

UNIVERSITY OF CALIFORNIA

Los Angeles

Policy Management and  
Interoperation through Negotiation  
in Ubiquitous Computing

A dissertation submitted in partial satisfaction of the  
requirements for the degree Doctor of Philosophy  
in Computer Science

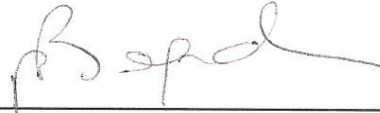
by

Venkatraman Ramakrishna

2008

© Copyright by  
Venkatraman Ramakrishna  
2008

The dissertation of Venkatraman Ramakrishna is approved.




---

Rajive Bagrodia



---

Rajit Gadhe



---

Leonard Kleinrock



---

Richard Muntz, Committee Co-chair



---

Peter Reiher, Committee Co-chair

University of California, Los Angeles

2008

To my family, for their love, support, and encouragement

## TABLE OF CONTENTS

<b>Chapter 1. Introduction and Motivation .....</b>	<b>1</b>
1.1. Ubiquitous Computing Environments: State-of-the-Art and Our Model .....	2
1.1.1 Physical Integration .....	3
1.1.2 Spontaneous Interoperation .....	3
1.2. Interoperability: Flexible and Secure Service Discovery and Access .....	7
1.3. Practical Motivational Scenarios .....	11
1.3.1 A Ubiquitous Conference Room.....	11
1.3.2 An Office/Lab Party.....	14
1.3.3 Security Perimeter Enforcement.....	16
1.3.4 Peer-to-Peer File Sharing.....	17
1.4. Assumptions.....	19
1.5. Thesis and Research Contributions.....	21
1.5.1 Thesis .....	22
1.5.2 Contributions.....	22
1.6. Dissertation Outline .....	25
<b>Chapter 2. Policy-Governed Domains and Cross-Domain Interoperation .....</b>	<b>27</b>
2.1. Devices and Groups as Administrative Domains with Scoped Policies.....	27
2.1.1 Interactions Among Domains .....	30
2.2. Policies and Their Role in UbiComp Domains .....	32
2.2.1 Policy Expression and Scope .....	33

2.2.2	Classification and Applications of Policies .....	35
2.2.3	Policy Management in a Domain.....	40
<b>Chapter 3. Negotiation Concepts.....</b>		<b>44</b>
3.1.	Interaction Model.....	44
3.2.	Agreement through Negotiation .....	51
3.3.	Negotiation Theories and Strategies .....	56
3.3.1	Characterization and Analysis of Negotiations.....	57
3.3.2	Factors that Affect Negotiation Strategies.....	60
<b>Chapter 4. Policy Language and Database.....</b>		<b>63</b>
4.1.	Design Requirements and Challenges .....	64
4.2.	Key Language Characteristics .....	68
4.3.	Database Semantics and Operations .....	79
4.3.1	Examination Operations.....	80
4.3.2	Modification Operations .....	82
4.4.	Policy Engine Implementation.....	84
4.5.	Comparison with Existing Policy Languages and Frameworks .....	86
<b>Chapter 5. Negotiation Protocol.....</b>		<b>90</b>
5.1.	Protocol Units .....	90
5.1.1	Speech Acts and Illocutionary Logic.....	91
5.1.2	Classification of Negotiation Message Contents .....	94
5.1.3	Negotiation Message Structure and Methods .....	96
5.2.	Protocol Semantics and State Machine.....	98

5.3.	Negotiation Query Resolution Algorithms .....	105
5.3.1	Generating Counter-Requests and Policies.....	105
5.3.2	Generating Matching Alternative Offers .....	114
5.3.3	Generation and Verification of Affirmative Offer Messages .....	115
<b>Chapter 6. Design and Implementation of a Policy Manager for a Ubiquitous</b>		
<b>Computing Environment .....</b>		
<b>117</b>		
6.1.	Device Communities and a Ubicomp Middleware Platform.....	117
6.1.1	Spheres of Influence and Panoply.....	118
6.1.2	Role of the Policy Manager Within a Panoply Sphere .....	122
6.2.	Panoply Policy Management Architecture .....	123
6.3.	Features Supported Through Policy Management.....	127
6.3.1	Mediated Interactions: Negotiation.....	128
6.3.2	Dynamic Access Control Through Event Filtering .....	130
6.3.3	Context-Sensitive Responses to Events .....	132
6.4.	Multi-Threaded Operation .....	134
6.4.1	Multiple Concurrent Negotiations .....	134
6.4.2	Dynamic Renegotiation .....	137
6.5.	External Helper Functions .....	139
6.6.	Protocol Reliability and Fault Tolerance .....	143
6.7.	Visualization and Administration Tools .....	149
6.7.1	Policy Browser.....	149
6.7.2	Negotiation Timeline .....	151

<b>Chapter 7. Demonstrative Applications</b> .....	153
7.1. Negotiation in Prominent Panoply Applications .....	154
7.1.1 Membership in a Home/Lab Environment (Sphere Joins) .....	154
7.1.2 Membership and Interaction in an Interactive Narrative .....	157
7.1.3 Membership and Interaction in a Smart Party .....	164
7.2. A Conference Room Environment.....	170
7.3. Peer-to-Peer File Sharing.....	176
7.4. Network Access Control Based on the QED Model.....	187
7.5. Opportunistic Configuration .....	194
7.5.1 Negotiation for Credential and Associated Key .....	196
7.5.2 Negotiation for Service and Associated Access Mechanism.....	197
<b>Chapter 8. Theoretical Analysis and Commentary</b> .....	200
8.1. System Analysis of the Negotiation Protocol .....	200
8.2. Theoretical Analysis of Negotiation .....	211
8.2.1 Policy Resolution .....	212
8.2.2 Centralized Policy Resolution Using an Oracle.....	214
8.2.3 Visualization of Negotiation as a Distributed (or Decentralized) Search Tree .....	216
8.2.4 Quality of a Negotiation.....	217
8.2.5 Metric Evaluation of Negotiation .....	221
8.3. Other Protocol Properties and Issues .....	231
8.3.1 Privacy Maintenance.....	231



8.3.2	Limitations of this Negotiation Procedure .....	234
8.3.3	Security Properties of the Negotiation Protocol .....	235
<b>Chapter 9. Performance .....</b>		<b>237</b>
9.1.	Overhead of Commonly Invoked Policy Engine Functions .....	237
9.2.	Negotiation Overhead .....	245
9.3.	Event Filtering through the Policy Manager .....	249
9.4.	Renegotiation Overhead .....	252
9.5.	Comparing Negotiation to Centralized Policy Resolution .....	255
9.5.1	Test Metrics .....	256
9.5.2	Characterization of Test Inputs .....	259
9.5.3	Test Parameters .....	261
9.5.4	Generation of Test Cases .....	263
9.5.5	Test Results .....	270
<b>Chapter 10. Related and Complementary Research .....</b>		<b>321</b>
10.1.	Negotiation Protocols .....	321
10.1.1	Automated Trust Negotiation .....	321
10.1.2	Automated Negotiation for Services .....	324
10.1.3	Speech Act-Inspired Negotiation Protocols .....	327
10.2.	Policy Languages and Models .....	327
10.2.1	Policy Languages .....	328
10.2.2	Ontology .....	332
10.2.3	Event-Triggered Condition-Action Policy Support .....	333

10.3.	Middleware .....	334
10.3.1	Smart Spaces and Ubiquitous Interoperation.....	334
10.3.2	Service Discovery and Access .....	337
10.3.3	Policy Management Frameworks .....	339
10.4.	Access Control and Trust in Distributed Systems .....	339
10.4.1	Access Control Models .....	340
10.4.2	Trust Models .....	344
<b>Chapter 11.</b>	<b>Future Work: Extensions and Enhancements .....</b>	<b>347</b>
11.1.	Negotiation Protocol .....	347
11.1.1	Heuristics and Strategies.....	348
11.1.2	Re-Modeling Negotiation .....	350
11.1.3	Collective Negotiation Among Multiple Parties.....	352
11.1.4	Framing Binding Legal Contracts.....	353
11.1.5	Security .....	354
11.2.	Policy Management Framework .....	354
11.2.1	Performance .....	355
11.2.2	Running the Policy Manager on Resource-Constrained Devices .....	356
11.2.3	Porting to Other Ubicomp Platforms .....	357
11.2.4	Large-Scale User Studies.....	359
11.2.5	Stress Testing.....	360
11.2.6	Controlling Event Flow.....	360
11.2.7	Integrating with Semantic Web Technologies.....	361

11.3.	Applications .....	362
11.3.1	An Interactive Museum Tour.....	362
11.3.2	E-Commerce: A Shopping Mall .....	363
11.4.	User Interaction.....	364
11.4.1	Negotiation: User Control and Feedback.....	365
11.4.2	Feedback and Analysis .....	367
11.4.3	Usable Policies.....	368
<b>Chapter 12.</b>	<b>Conclusion .....</b>	<b>372</b>
<b>Appendix A:</b>	<b>Policy Language Reference .....</b>	<b>379</b>
<b>Appendix B:</b>	<b>Sample Database Generated for Performance Testing .....</b>	<b>389</b>
<b>References</b> .....		<b>394</b>

## LIST OF FIGURES

Figure 1. Application-Level Interoperation in the Semantic Web.....	8
Figure 2. A Conference Room Scenario .....	12
Figure 3. A Policy-Guided Negotiation Framework as a Middleware .....	21
Figure 4. Policy Scope and Classification .....	34
Figure 5. Domains with Policy Management Middlewares that Mediate Interactions.....	43
Figure 6. Inter-Domain Interaction Model.....	48
Figure 7. Negotiation Protocol: High-Level State Machine .....	98
Figure 8a. Negotiation Protocol: Request Queue Processing: <i>Servicing Requests</i> .....	103
Figure 8b. Negotiation Protocol: Request Queue Processing: <i>Processing Offers</i> .....	104
Figure 9. Overall System Architecture .....	119
Figure 10. The Panoply Middleware (Borrowed from Eustice’s PhD Dissertation [Eustice2008b]).....	120
Figure 11. Policy Manager Functional Diagram.....	124
Figure 12. Four Concurrent Negotiations .....	135
Figure 13a. Voucher Data Structure .....	141
Figure 13b. Opaque Field Data Structure .....	142
Figure 14. Faults and Recovery in a Negotiation Protocol Instance.....	146
Figure 15. Policy Browser Snapshot.....	150
Figure 16. Negotiation Timeline Example.....	152

Figure 17. A Snapshot of the Interactive Narrative Client GUI (Note the Policy-Driven Hint Displayed in the Bottom Right-Hand Panel) .....	163
Figure 18. Smart Party GUI: Enables Manual Playlist Control.....	169
Figure 19. Snapshot of the <i>P2PClient</i> Application GUI.....	177
Figure 20. Centralized Search Tree Representing Policy Resolution by an Oracle .....	213
Figure 21. Distributed Search Tree Representing Policy Resolution through Negotiation.....	217
Figure 22. Query Processing Times along Various Dimensions .....	240
Figure 23. Clause Lookup Performance .....	242
Figure 24. Database Modification Overhead.....	244
Figure 25. Variation of Renegotiation Overhead with the Number of Existing Associations .....	254
Figure 26. Variation of Renegotiation Overhead with the Number of Attempted Renegotiations.....	255
Figure 27. Comparison of Actual and Optimal Negotiation Steps .....	273
Figure 28. $I_{min}$ : Number of Nodes Examined During Policy Resolution .....	275
Figure 29. $I_{min}$ : Number of Nodes Examined per Unit Database (Number of Nodes / Number of Policy Statements in the Database) .....	276
Figure 30. $I_{min}$ : Total Processing Time for Policy Resolution .....	277
Figure 31. $I_{min}$ : Average Processing Time for Policy Resolution per Negotiation Step ..	278
Figure 32. $I_{min}$ : Total Processing Time for Policy Resolution per Unit Database (Total Processing Time / Number of Policy Statements in the Database) .....	279

Figure 33. $l_{min}$ : Average Processing Time for Policy Resolution per Negotiation Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database).....	280
Figure 34. $l_{min}$ : Average Fraction of Intermediate Requests Granted .....	281
Figure 35. $l_{min}$ : Average Fraction of Alternatives Examined .....	282
Figure 36. $l_{min}$ : Average Position of the First Valid Alternative.....	284
Figure 37. $l_{neg}$ : Number of Nodes Examined During Policy Resolution .....	285
Figure 38. $l_{neg}$ : Number of Nodes Examined per Unit Database (Number of Nodes / Number of Policy Statements in the Database) .....	286
Figure 39. $l_{neg}$ : Total Processing Time for Policy Resolution.....	287
Figure 40. $l_{neg}$ : Average Processing Time for Policy Resolution per Negotiation Step ..	288
Figure 41. $l_{neg}$ : Total Processing Time for Policy Resolution per Unit Database (Total Processing Time / Number of Policy Statements in the Database) .....	289
Figure 42. $l_{neg}$ : Average Processing Time for Policy Resolution per Negotiation Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database).....	289
Figure 43. $l_{neg}$ : Average Fraction of Intermediate Requests Granted .....	290
Figure 44. $l_{neg}$ : Average Fraction of Alternatives Examined .....	292
Figure 45. $l_{neg}$ : Average Position of the First Valid Alternative .....	292
Figure 46. $b_{max}$ : Average Number of Policy Resolution Steps .....	294
Figure 47. $b_{max}$ : Policy Resolution Step Ratio (Number of Negotiation Steps / Optimal Number of Steps) .....	294

Figure 48. $b_{max}$ : Average Number of Nodes Examined .....	296
Figure 49. $b_{max}$ : Ratio of Number of Nodes Examined (Negotiation Tree Nodes / Oracular Tree Nodes).....	296
Figure 50. $b_{max}$ : Average Number of Nodes per Unit Database (Number of Nodes / Number of Policy Statements in the Database) .....	298
Figure 51. $b_{max}$ : Processing Time for Policy Resolution.....	299
Figure 52. $b_{max}$ : Ratio of Processing Times for Policy Resolution (Negotiation Time / Oracular Time).....	299
Figure 53. $b_{max}$ : Total Processing Time for Policy Resolution per Unit Database (Processing Time / Number of Policy Statements in the Database) .....	301
Figure 54. $b_{max}$ : Average Processing Time for Policy Resolution per Policy Resolution Step .....	301
Figure 55. $b_{max}$ : Ratio of Average Processing Time for Policy Resolution per Step (Negotiation Time / Oracular Time).....	302
Figure 56. $b_{max}$ : Average Processing Time for Policy Resolution Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database) ..	303
Figure 57. $b_{max}$ : Average Fraction of Intermediate Requests Granted.....	304
Figure 58. $b_{max}$ : Average Fraction of Alternatives Examined.....	305
Figure 59. $b_{max}$ : Average Position of the First Valid Alternative .....	306
Figure 60. $d_{max}$ : Average Number of Policy Resolution Steps .....	307
Figure 61. $d_{max}$ : Policy Resolution Step Ratio (Number of Negotiation Steps / Optimal Number of Steps) .....	308

Figure 62. $d_{max}$ : Average Number of Nodes Examined .....	309
Figure 63. $d_{max}$ : Ratio of Number of Nodes Examined (Negotiation Tree Nodes / Oracular Tree Nodes).....	309
Figure 64. $d_{max}$ : Average Number of Nodes per Unit Database (Number of Nodes / Number of Policy Statements in the Database) .....	311
Figure 65. $d_{max}$ : Processing Time.....	312
Figure 66. $d_{max}$ : Ratio of Processing Times (Negotiation Time / Oracular Time).....	312
Figure 67. $d_{max}$ : Total Processing Time per Unit Database (Processing Time / Number of Policy Statements in the Database) .....	313
Figure 68. $d_{max}$ : Average Processing Time per Policy Resolution Step .....	314
Figure 69. $d_{max}$ : Ratio of Average Processing Times per Step (Negotiation Time / Oracular Time).....	314
Figure 70. $d_{max}$ : Average Processing Time per Policy Resolution Step / Database Size.	316
Figure 71. $d_{max}$ : Average Fraction of Intermediate Requests Granted.....	317
Figure 72. $d_{max}$ : Average Fraction of Alternatives Examined.....	318
Figure 73. $d_{max}$ : Average Position of the First Valid Alternative .....	319
Figure 74. Negotiation Protocol Choice Point.....	351
Figure 75. GAIA Architecture (Borrowed from GAIA homepage: <a href="http://gaia.cs.uiuc.edu/">http://gaia.cs.uiuc.edu/</a> ).....	358
Figure 76. GAIA Architecture Modified to Incorporate the Policy Manager .....	358



## LIST OF TABLES

Table 1: Core Panoply Services (Borrowed from Eustice’s PhD Dissertation [Eustice2008b]).....	121
Table 2. Variation of Query Processing Times.....	241
Table 3. Clause Lookup Times .....	242
Table 4. Overhead of Database Modification Operations .....	245
Table 5. Sample Negotiation Performance Measurements (in milliseconds).....	247
Table 6. Sample Negotiation Performance Measurements (in milliseconds).....	248
Table 7. Event Filtering Performance on a Single Device (in milliseconds).....	250
Table 8. Event Filtering Performance when Two Devices Interact (in milliseconds).....	252
Table 9. Roadmap to Results of Optimality Measurements .....	272
Table 10. Comparison of Actual and Optimal Negotiation Steps .....	274

## ACKNOWLEDGMENTS

This dissertation would not have been possible without the guidance and support of my advisor, Dr. Peter Reiher, who helped me stay focused and motivated through my years of research. I am also very grateful to Dr. Leonard Kleinrock for his guidance, and the invaluable advice he gave me during our research meetings. Additionally, I would like to acknowledge the contributions of the other members of the Laboratory for Advanced Systems Research. I would like to thank Kevin Eustice, Nam Nguyen, and Matthew Schnaider for the countless discussions we participated in and their work on the Panoply project, which complemented my research and made my task easier. Many thanks are owed to Erik Kline, Chuck Fleming, Matt Beaumont, and Rafael Laufer, who provided me with many useful suggestions. I am grateful to Shane Markstrum and Martin Lukac for their work on the QED project and the Panoply proposal, both of which served to kick-start my research. Generous financial support was provided by the National Science Foundation (grant CNS-0427748). I owe a special thanks to Janice Martin for her expert editing of the dissertation. Last but not least, I would like to thank my parents for their continuous support and encouragement, which kept me motivated through my years as a graduate student.

## VITA

June 26, 1979	Born, Jhansi, India
2001	B. Tech. (Honors) Computer Science and Engineering Indian Institute of Technology Kharagpur, India
2003	MS, Computer Science University of California Los Angeles, California, U.S.A.
2001-2008	Graduate Student Researcher Laboratory for Advanced Systems Research (LASR) University of California Los Angeles, California, U.S.A.
January-March 2007	Teaching Assistant, Computer Science Department University of California Los Angeles, California, U.S.A.
1996	National Talent Search Examination (NTSE) Scholarship Awarded by the National Council of Educational Research and Training (NCERT) Government of India
2006	Chorafas Foundation Scholarship
2004	Graduate Technical Intern Intel Corporation Chandler, Arizona, U.S.A.
2008-	Software Development Engineer Microsoft Corporation Redmond, Washington, U.S.A.

## PUBLICATIONS

- Vincent Ferreria, Alexey Rudenko, Kevin Eustice, Richard Guy, V. Ramakrishna, and Peter Reiher, "Panda: Middleware to Provide the Benefits of Active Networks to Legacy Applications," *DANCE 2002*, May 2002.
- V. Ramakrishna, Max Robinson, Kevin Eustice and Peter Reiher, "An Active Self-Optimizing Multiplayer Gaming Architecture," *Proceedings of the 5<sup>th</sup> Annual International Workshop on Active Middleware Services (AMS 2003)*, 25 June 2003, Seattle, Washington.
- V. Ramakrishna, Rakesh Kumar and Anupam Basu, "Switching Activity Minimization by Efficient Instruction Set Architecture Design," *Proceedings of the 45<sup>th</sup> IEEE International Midwest Symposium on Circuits and Systems (MWSCAS 2002)*, August 4-7, 2002, Tulsa, Oklahoma.
- Kevin Eustice, Leonard Kleinrock, Shane Markstrum, Gerald Popek, Venkatraman Ramakrishna, Peter Reiher, "Enabling Secure Ubiquitous Interactions," *Proceedings of the 1<sup>st</sup> International Workshop on Middleware for Pervasive and Ad-Hoc Computing (Co-located with Middleware 2003)*, 17 June 2003, Rio de Janeiro, Brazil.
- Kevin Eustice, Leonard Kleinrock, Shane Markstrum, Gerald Popek, Venkatraman Ramakrishna, Peter Reiher, "Wi-Fi Nomads: The Case for Quarantine, Examination and Decontamination," *Proceedings of the New Security Paradigms Workshop 2003*, August 2003, Ascona, Switzerland.
- V. Ramakrishna, Max Robinson, Kevin Eustice and Peter Reiher, "An Active Self-Optimizing Multiplayer Gaming Architecture," *Cluster Computing Journal*, Publisher: Springer Netherlands, Issue: Volume 9, Number 2, "Special Issue: *Autonomic Computing*," Guest Editor: Manish Parashar, pp. 201-215, April 2006.
- Kevin Eustice, Leonard Kleinrock, Shane Markstrum, Gerald Popek, Venkatraman Ramakrishna, Peter Reiher, "QED: Securing the Mobile Masses," *UCLA Computer Science Department Technical Report #TR050042*, October 2005.
- V. Ramakrishna, Kevin Eustice and Matthew Schnaider, "Approaches for Ensuring Security and Privacy in Unplanned Ubiquitous Computing Interactions," *Proceedings of the International Workshop on Research Challenges in Security and Privacy for Mobile and Wireless Networks (WSPWN06)*, March 15-16, 2006, Miami, Florida.

- V. Ramakrishna, Kevin Eustice, and Peter Reiher, "Negotiating Agreements Using Policies in Ubiquitous Computing Scenarios," *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA'07)*, June 19-20, 2007, Newport Beach, California.
- Kevin Eustice, V. Ramakrishna, Alison Walker, Matthew Schnaider, Nam Nguyen and Peter Reiher, "nan0sphere: Location-Driven Fiction for Groups of Users," *Proceedings of the 12<sup>th</sup> International Conference on Human-Computer Interaction (HCII 2007)*, 22-27 July 2007, Beijing, P.R.China.
- V. Ramakrishna, Kevin Eustice and Matthew Schnaider, "Chapter 8: Approaches for Ensuring Security and Privacy in Unplanned Ubiquitous Computing Interactions," *Mobile and Wireless Networks Security and Privacy*," Edited Volume by Springer Science+Business Media, LLC, Ed: Makki et al., 2007.
- Peter Reiher, Kevin Eustice, and V. Ramakrishna, "Chapter: Security and Privacy in Pervasive Computing," *Security and Privacy in Mobile and Wireless Networking*, Troubadour Publishing, by Stefanos Gritzalis, Tom Karygiannis and Charalabos Skianis (editors), ISBN: 978-1905886-906, 1 June 2008.
- Kevin Eustice, V. Ramakrishna, Nam Nguyen, and Peter Reiher, "The Smart Party: A Personalized Location-aware Multimedia Experience," *Proceedings of the 5<sup>th</sup> IEEE Consumer Communications and Networking Conference (CCNC 2008)*, Las Vegas, NV, January 10-12, 2008.

## ABSTRACT OF THE DISSERTATION

Policy Management and  
Interoperation through Negotiation  
in Ubiquitous Computing

by

Venkatraman Ramakrishna

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2008

Professor Peter Reiher, Co-chair

Professor Richard Muntz, Co-chair

Ubiquitous computing environments consist of autonomous domains that are not administered centrally and have independent goals and policies. These domains, which could be single mobile devices or networks of devices, are capable of communicating through standardized protocols and identifying external services. But true spontaneous interoperation leading to policy-compliant resource and service access agreements across domains is yet to be realized. The large number

of possible interaction contexts, resource heterogeneity, and differing security policies of the domains make the use of application level protocols for every scenario impractical and non-scalable. Also, every domain cannot expect to identify or have a pre-arranged trust relationship with every other domain.

We have designed and implemented a generic *negotiation protocol* based on illocutionary speech acts that enables domains to reach resource and service access agreements. This protocol is guided by the local private policies of each domain, which specify system invariants, goals, resource usage, and security constraints, in a declarative logical language. Negotiation was achieved within a broader policy management and mediation framework, which was designed and implemented as part of the Panoply ubiquitous computing middleware. This framework also provides other services, including dynamic context-sensitive access control through message filtering and event-driven system responses.

In this dissertation, I will describe mobile and ubicomp applications that benefit from policy management and negotiation, and show that the protocol performance is adequate for practical scenarios. I will show how the negotiation protocol was modeled as a distributed policy resolution, where neither negotiator is privy to the other's policies, and analyze its theoretical correctness properties. I will describe how random test cases were generated for the purpose of comparing negotiation performance to centralized policy resolution that is optimal in the number of negotiation steps. The results indicate the feasibility of using the negotiation protocol to generate agreements in ubicomp scenarios.

# Chapter 1

## Introduction and Motivation

The past few years have seen rapid growth of technology, both in academia and in industry, slowly bringing the ubiquitous computing vision to reality. In our current transitory phase, we see a mixture of static desktop computing and mobile computing. Widespread deployment of WiFi hotspots, pioneered by cities, private businesses, and corporations like Google, and the large-scale adoption of mobile devices by users, enables an unprecedented amount of mobile computing. Such computers, and embedded systems performing sensory and actuator functions, are getting more powerful, *intelligent* and capable of running more useful applications without explicit user intervention. Wireless networks are easier to discover and connect to without requiring technical expertise. The services provided by our surrounding environments are not limited to connectivity, and a wide variety of *smart spaces* have been built and deployed, primarily in academic and controlled settings. A more widespread deployment would ideally result in our smarter devices being able to make use of available services at any time or place. Yet this requires a level of *spontaneous interoperation* among computing systems that is currently infeasible, primarily because existing solutions fail to provide optimal security, privacy and usability, or some combination of these features. Typical environments where users need service are either completely open or are extremely picky about the devices with which they interact, and have stringent security policies that infringe on the



privacy of the user devices (which must agree to the policies to obtain service). These devices and networks should not be, and do not have to be, so rigid in their interaction mechanisms. Nor do systems have to restrict interaction to familiar, trusted environments.

Every computing system, whether a single device or a networked group of devices, can specify all its constraints, requirements and system state in the form of policies, and these systems can interact with each other to achieve mutually satisfactory resource sharing and service access agreements through a process of *negotiation*. In this dissertation, we will describe our research in the management of policies within individual systems participating in a ubiquitous computing environment, and the necessary operations they support—prominent among which is a negotiation protocol. In the remainder of this section, we shed more light on current ubiquitous computing research and the challenges that motivate our research. We describe and justify our chosen ubicomp interaction models, and conclude by listing the research contributions, and an outline for the rest of this dissertation.

### **1.1. Ubiquitous Computing Environments: State-of-the-Art and Our Model**

In Mark Weiser’s original 1991 vision of ubiquitous computing (*ubicomp* for short) [Weiser1991], computers will pervade the physical space around us without being obvious to the human eye. Such an infrastructure will help users perform various tasks, offer useful services and allow information access anywhere and at any time. More recently, Kindberg and Fox [Kindberg2002] identified *physical integration* and *spontaneous interoperation* as the two fundamental characteristics of ubicomp systems.

### **1.1.1 Physical Integration**

Thanks to advances in embedded systems technology, our mobile accessories, like cell phones, PDAs and watches, run complex applications, offer services like GPS, and communicate wirelessly using embedded processors. Even our refrigerators, walls and clothing will get as smart in the near future by using sensors, actuators and interfaces. Ubiquitous networking is also rapidly becoming a reality. We already have established standards for wired and wireless LANs, cellular, satellite and personal area networks, and efforts are currently under way to produce standards for wireless MANs, or WiMax [IEEE802.16], and vehicular networking.

*Smart space* projects are manifestations of computing, networking and sensory elements physically integrated into office-like spaces. Examples include Oxygen [MIT-Oxygen], Gaia [Román2002], One.world [Grimm2004a][Grimm2004b] and Centaurus [Kagal2001a][Kagal2001b][Undercoffer2003]. These projects demonstrate infrastructure that can dynamically manage resources, enable seamless communication, and adapt to changing context using a mix of mobile devices and components integrated into the background (apart from the interfaces, these are oblivious to users).

### **1.1.2 Spontaneous Interoperation**

Smart spaces are designed to govern local *hot spots* and do not scale to a global distributed system. Successful ubiquitous computing can be achieved with widespread deployment of such spaces that are connected through the Internet, share resources with each other and offer services to mobile devices purely through local interactions. Users

will expect to obtain and make use of resources wherever they go and to run their applications and obtain private data and information, either through their personal devices or local clients. Unfortunately, the heterogeneity of hardware, software, resources, networking technologies and applications that each space or device will possess prevents these devices and systems from interoperating to the extent required. Smart spaces rely on standardization of these features for global interoperation, which is practically not enforceable. Even if certain mechanisms, hardware, system software and networking technologies become de facto standards through popularity or market forces (like TCP/IP networking culminating in the Internet and the World Wide Web), different people manage and use local domains, and have different needs and expectations. They might use the design and resource management principles proposed by Oxygen or Centaurus, but they will have unique resource requirements and expect certain functionality from their systems, in addition to having unique security and privacy requirements. The infrastructure deployed at physical spaces that will enable ubicomp is growing in a decentralized fashion through the efforts of research groups and companies. Though it will create the problems described above, bottom-up growth of the infrastructure is both inevitable and desirable as it promotes innovation and lets designers and administrators make independent choices. The problem is to enable a standard through which different devices and networks can interoperate although they are managed and used by different people, do not share resource capabilities, security and privacy constraints, or have pre-decided trust relationships. Such an interoperation standard would not replace the research done in building smart spaces, but would rather be complementary to it.

### 1.1.3 Security, Privacy and Usability in Ubicomp Environments

Ensuring security, privacy and usability in ubiquitous computing environments is a challenge, as these are often at cross purposes. Unfortunately, most systems treat security as an afterthought, as something to be retrofitted onto mechanisms or interfaces already implemented, an approach that does not work and is not extensible. This is especially true of open multi-user ubiquitous computing environments, where external entities (semi-trusted) are permitted to discover local resources, and such resources are dynamically allocated. The wide range of contexts and entities one comes in contact with makes it infeasible for any computing system to completely specify security policy, or use off-the-shelf security mechanisms. Still, both mobile devices and the networks they come in contact with (to give one mode of interaction) must guard against malicious attacks and resource abuse while preserving the fundamentally open nature of ubicomp. Therefore, autonomous systems must have the capability to make intelligent tradeoffs in particular contexts when deciding the nature and scope of interactions, based on a limited set of policies. These tradeoffs often involve the sacrifice of privacy in order to get higher security assurances, and obtain any kind of useful output.

Looking at this from a higher level, we advocate a top-down approach for ensuring security and privacy by looking at *ubiquitous interoperation in its entirety* while permitting a bottom-up, decentralized growth and deployment of ubiquitous services. Through this approach, systems could use a wide variety of security enforcement mechanisms like virus scanners, firewalls, tools like *nmap*, intrusion detection utilities, and QED [Eustice2003b], but these must be augmented with security policies that dictate

how and when such mechanisms should be used. More powerful security and access control frameworks like GRBAC [Covington2000], DRBAC [Freudenthal2002], and evolving trust models are inadequate in various ways; this will be addressed in Section 10. The top-down approach to security will be helped only through flexible interoperation of the kind that is enabled through our research.

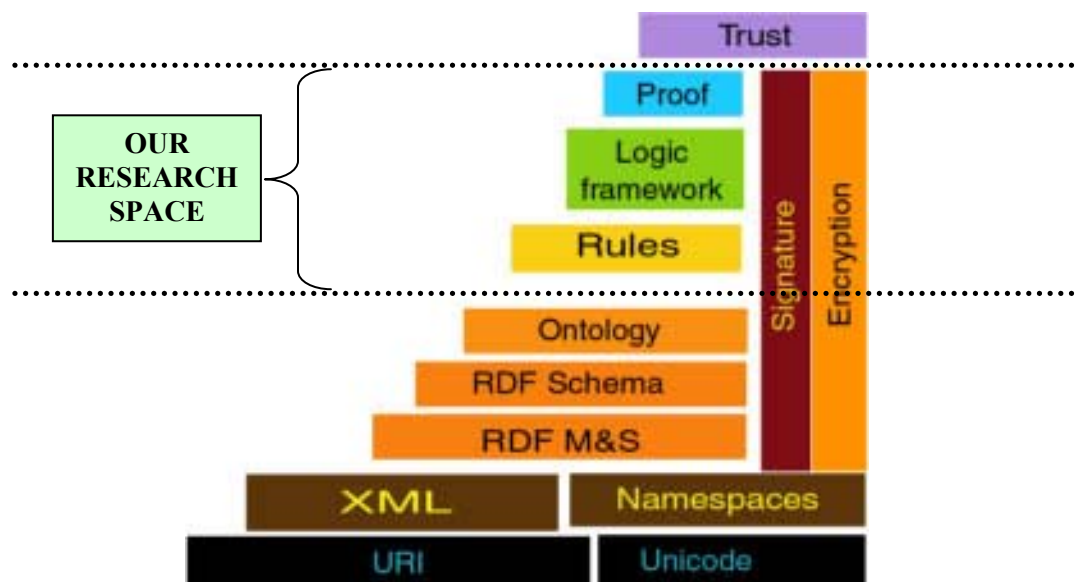
The more widely addressed challenge in ubicomp is system *usability*, and the building of interfaces that provide *intuitive* ways for non-technical users to interact with their personal devices and with the invisible infrastructure components in the background. It would be impractical to expect such users to change configuration settings in devices to establish network connectivity, change security and privacy levels, or explicitly run commands to discover and obtain resources. Users and system administrators should be able to set policy and expect their devices to figure out which lower-level resources (such as network connectivity, display and audio devices, and file systems) are required and then obtain these from any environment they happen to be in. Devices should also adjust service and information provided to users with context. Tools like DHCP and Zeroconf [Guttman2001] allow a measure of automated connectivity at lower levels, but usually do a proper job only through prearranged configuration scripts and when devices and networks *know* each other, unless neither has a security policy. Therefore, in most practical systems, the needs of security and usability often clash. A system that enables more ease of use invariably enables more ease of abuse. More flexibility is needed to balance security and resource needs, without requiring users to make arbitrary decisions whenever default configurations fail. Frameworks deployed on

devices and networks should be context-aware and not require users to constantly change settings when such parameters change. Our research addresses this issue, with a caveat that it will not be possible to let devices make 100% of the decisions by themselves without significant advances in AI. There will be situations when feedback must be provided to users, and this needs to be done through suitable interfaces. Though our research does not address this issue in depth, we do comment on this in the future work section (see Chapter 11).

## **1.2. Interoperability: Flexible and Secure Service Discovery and Access**

In a ubicomp environment, not every device or system of interconnected devices will have or require every possible resource or capability, as long as necessary services can be obtained through interoperation with the surrounding environment. In the service-oriented view of computing [Huhns2005], services are available and are offered by autonomous entities. The discovery and the setting of terms for obtaining these services are left to the runtime environment. Interoperation therefore consists of two processes: discovery of external services and resources, and obtaining access (setting terms) to them. In addition, information about objects and offered services can also be communicated through suitable interfaces. Current systems and prior research have provided separate solutions for discovery and access control. (*Note:* We are concerned here with application and middleware layer interoperation.) In open systems and those offering ubiquitous services, we will see the functions of discovery and access merge more and more [Zhu2005b], and existing solutions won't work. This is because the *resources* possessed

could include private information that domains don't necessarily want to advertise to the wrong parties. Service providers and consumers also need a common semantic framework to describe all kinds of information, resources and protocols; suitable standards may result from ongoing *Semantic Web* [SemWeb] research (see Figure 1). These entities also need to be able to reconcile their resource needs, capabilities and security policies in order to interoperate. We are not concerned with lower layer networking, and significant research has gone into standardizing interoperation mechanisms at those layers, such as mobile IP [Bhagwat1996]. Research in application session handoff, like IMASH [Bagrodia2003] and the *grid*, which enables resource sharing in a large-scale distributed environment, are valuable but often do not consider mobility or trust issues, or both.



**Figure 1.** Application-Level Interoperation in the Semantic Web

Ensuring spontaneous interoperation among sets of devices and administrative domains, typically between user devices and wireless networks, is hard because of the following characteristics of ubicomp:

- Heterogeneity of devices and communication features
- Differences in the kinds of resources (and services) possessed and offered
- Differences in capabilities possessed for interacting with external entities
- Contexts and context-sensitive constraints that cannot be anticipated in advance
- Diversity of security and privacy policies, and trust relationships

To elaborate, heterogeneity poses the following problem: how does a computing system match requests posed by an interacting system to its local resources while maintaining its security and resource management policies? It is impractical to both enforce standard application layer protocols and interfaces as well as a uniform standard of trust on every administrative domain. Conceptually, multiple protocols and mechanisms for service access could be deployed and used on an ad hoc basis. But every unique context consists of a service consumer with different characteristics requiring a different variation of the service. Creating separate protocol instances for each unique context will lead to a combinatorial explosion. To circumvent this problem, designs and configurations have been proposed that are inflexible and which adopt an all-or-nothing approach. One approach advocates rigid policies governing interactions, which typically limit such interactions to devices or domains that have prearranged trust relationships, and/or predeployed mechanisms to make use of external services when two devices/networks come into contact. Such solutions do not scale globally. At the other



extreme, systems use standard open interfaces which allow free interoperation at the expense of security and privacy (more about this in Section 1.3). This is the design approach of projects like Oxygen's Metaglove [Coen1999] and Sun's JINI [Waldo1999] which enable resource discovery and access. These and other ubicomp infrastructures, until recently, have not considered user and device mobility. Neither do they consider the dynamism of the environment, where interaction mechanisms, goals and constraints must vary with context.

The need for interoperation often forces a device to expose more private information and allow access to more resources than it needs or wants. This is far from ideal, but it is the approach used by various systems for which security or wide-scale interoperability is not a priority. Fallback or alternative agreements can be reached by exchanging information that allows interacting entities to determine all alternatives that could result in a compatible resource sharing agreement; in many cases, constraints could be relaxed by determining exactly how much security can be risked or how much privacy can be given up. Most systems leave it to users to make these decisions upon failure. This not only detracts from usability, but also circumvents a problem for which an automated solution can be worked out. Moreover, rigid policies or tailor-made applications for domain-specific needs cannot work in an environment where context changes dynamically, and all situations cannot be anticipated beforehand. Unanticipated results and security holes will invariably occur. The solution is a middleware for policy management, based on the use of flexible and domain-dependent policies that can be easily specified in a high-level language, and can be easily modified during runtime.

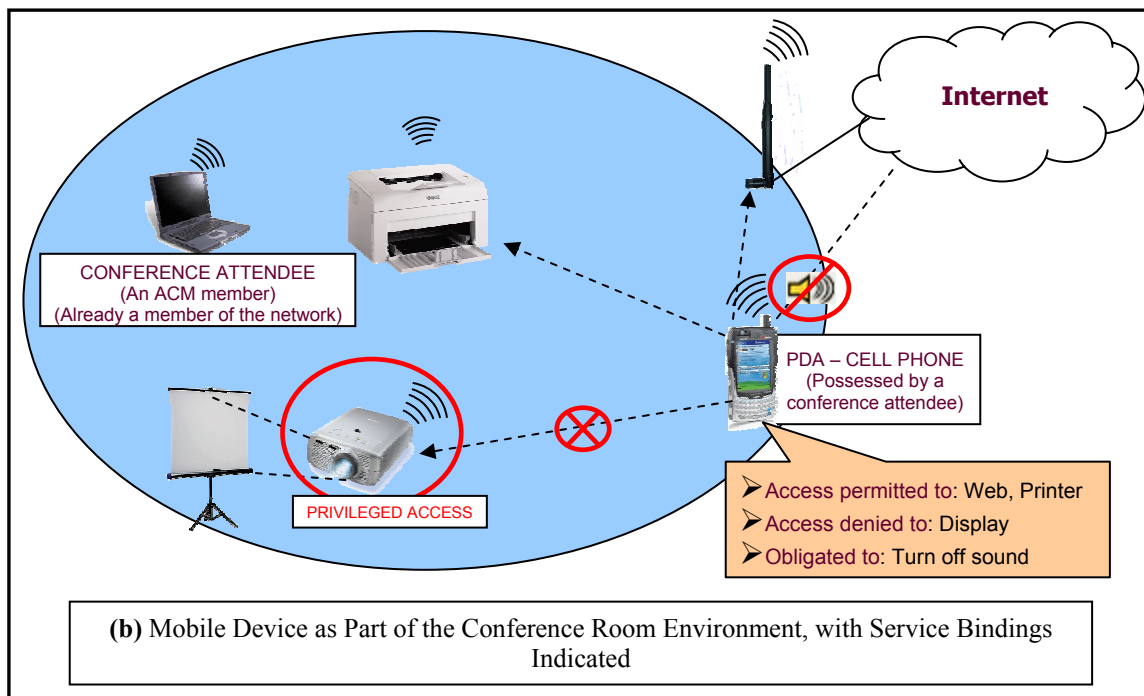
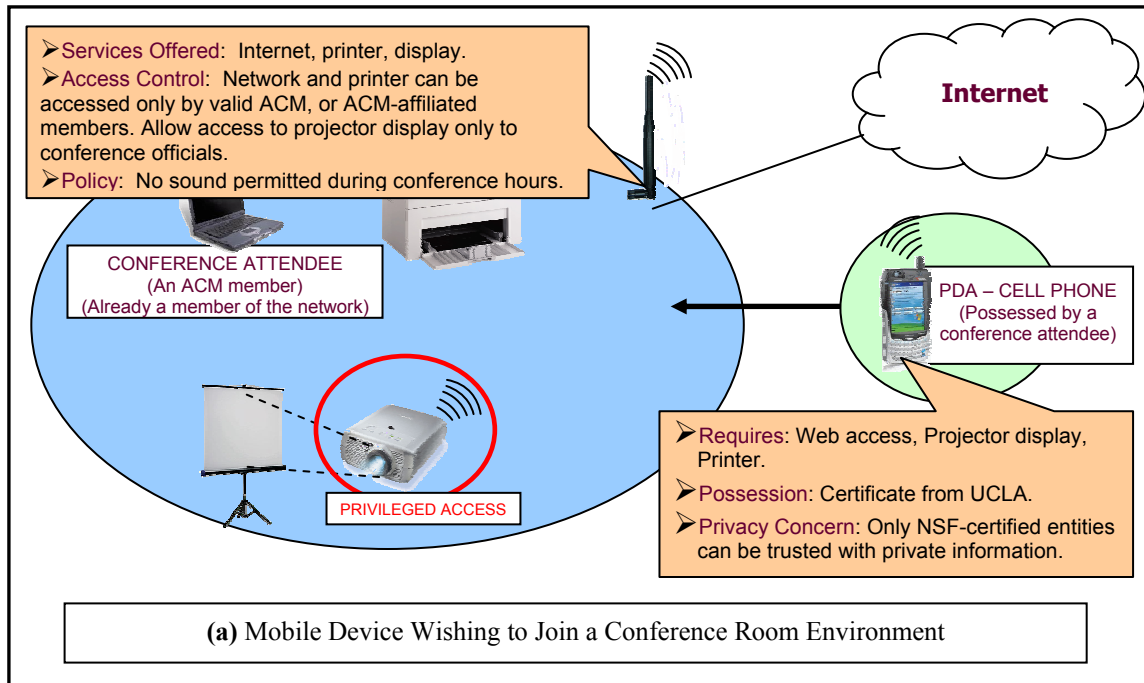
In the following subsection, we will show how interoperation occurs, or sometimes doesn't occur, in contemporary mobile and ubiquitous computing scenarios, and contrast it with the ideal or optimal corresponding scenario. The rest of this dissertation will then describe how our research enables transformations from the conventional to the optimal scenario.

### **1.3. Practical Motivational Scenarios**

There is no one killer application that demonstrates how and why our research is valuable in a ubiquitous computing environment. Indeed, the very promise of ubicomp is that it can support a potentially unbounded set of applications that make users' lives easier without requiring them to become more tech savvy. Successful ubicomp research projects over the past decade have demonstrated the worth of their research in terms of the versatility of their frameworks. We make a similar attempt here. We describe specific scenarios that motivate the need for our research, and indicate how these can be generalized. These scenarios face problems ranging from misconfiguration to security and privacy violations that occur due to the lack of spontaneous interoperation-enabling technology.

#### **1.3.1 A Ubiquitous Conference Room**

Consider a motivating scenario where interoperation fails in the absence of prior configuration. This is due to stringent security policies and a lack of preestablished trust.



**Figure 2.** A Conference Room Scenario

An ACM-conducted conference (see Figure 2) is being held in a room containing a wireless network, a display device, a projector and a printer. Though this room is accessible to valid attendees and conference officials, as well as non-attendees, it may offer services only to valid attendees and officials, with varying degrees of access. The room network (managed by a server) allows any ACM-certified computer access to these services, though the projector display can be accessed only by ACM officials. A conference attendee could access these services through his personal mobile devices, like the laptop depicted in Figure 2a. A prospective attendee now tries to join the network and access these services through his smart PDA-phone. This device possesses a certificate from UCLA, which is an ACM-affiliated school, and also has policies restricting release of sensitive credentials to a trusted entity, in this case, an NSF-certified entity. As it turns out, the conference is NSF-certified, and the server can prove this. The two interacting entities, the conference room server and the attendee's PDA-phone, have compatible constraints, which should lead to a service binding of the form indicated in Figure 2b in an automated manner. But today, the attendee's device could, at best, discover the wireless network and associate with it if permitted. Accessing the other services would require manual configuration of the device. What is lacking is an effective procedure that allows the conference room server and the PDA-phone to exchange policy and accreditation information, verify that information, and agree on service access. In addition, the server could impose policy constraints on the supplicant device, such as an audio-silent request during the conference, and grant service access only upon compliance. The conference system might choose to offer further privileges (such as

copies of presentations or papers) to users who subscribe to a journal related to the conference. These tasks can be achieved through a step-by-step progression of trading information and making agreement decisions.

The dynamics of the above scenario are similar to how web services are accessed today. Clients and servers interact through rigid protocols, often with user input, because of the lack of a viable procedure for dynamic agreement. Standards like P3P (Platform for Privacy Preferences) [P3P] have been promoted to make web service access more flexible, but these have achieved very limited success or adoption (we will discuss this further in the Related Work section). Therefore, a spontaneously interoperating conference room could be the model for generic web service access as well.

### **1.3.2 An Office/Lab Party**

This scenario builds on the *Smart Party* concept [Eustice2008a] introduced by Eustice et al. as a ubiquitous computing application supported by the Panoply middleware (which is the subject of a PhD dissertation by Kevin Eustice) [Eustice2008b]. In brief, a smart party provides media-based entertainment to guests at a party, being sensitive to the preferences of the guests present in the various party locations. For example, our lab at UCLA is subdivided into multiple rooms, each room containing a speaker and a wireless access point capable of media storage and playback. In the ideal scenario, guests are invited to the party and given invitations in the form of digital *vouchers* (see Chapter 6). The guests arrive at the lab, are let in, and move around from one room to another. They carry personal mobile devices, such as cell phones, PDAs or MicroPCs, which store their

digital credentials and their media preferences. The devices can associate with the party environment only upon production of a valid guest voucher. Based on the preferences of the guests within a room, a dynamic song playlist is generated, relevant songs are obtained from the guests' devices, and the songs are played on the local speaker.

The party host H adjusts the playlists from time-to-time. Ordinary guests are not permitted to control the playlists, but when H leaves the party for a while, he provides a voucher with delegated permissions to one of his trusted guests G through a mobile device-to-device transaction, so that G can control playlists in H's absence. The voucher is usable only if G's device lies within the bounds of the lab. When G's device attempts to modify a playlist, the lab network dynamically allows access on the basis of G possessing appropriate credentials (in the voucher) and being able to prove its presence within the room where the playlist is being modified.

In addition to the media entertainment, the scenario also encompasses the door that lets guests into the lab. The opening and closing of the door can be controlled by commands from computers present within the lab. The lab network has a policy not to let more than a specified number of guests (say 7) in at a time unless explicitly allowed by the host within the lab. In addition, no visitors are let in unless the host (or someone with delegated host permissions) is present within the lab. The door can be opened through acoustic signals, such as knocks, as well. Therefore, the first 15 guests are allowed entry upon knocking, but subsequent visitors are turned away (this scenario assumes that more than 15 invitations are sent) as their knocks are unable to open the door, even though they

can associate with the network. Guests who are turned away can enter if earlier guests choose to leave, thereby reducing the instantaneous party population.

The above lab party scenario demonstrates the variety of policy management functions that are necessary for the smooth running of a ubicomp application. Allowing guest devices to participate in the party network requires a form of negotiation. Preventing unauthorized guests from modifying playlists, as well as allowing temporary “hosts” to control the playlists, requires dynamic access control using negotiation or some form of challenge-response, which is not easily provided by existing frameworks. Context-sensitive control of the smart door requires dynamic event monitoring and action triggers. This scenario thus establishes the need for a policy management framework.

### **1.3.3 Security Perimeter Enforcement**

Professor B takes his personal laptop to a panel meeting conducted under the auspices of the IEEE and attended by PIs and graduate students working on the new 802.21 standard. The meeting takes place in a secure environment, the network being guarded by the latest firewall and DDoS defense technology. Direct access is allowed only to the local LAN, and limited Internet access is provided through a proxy. Even with these guards, there is always the danger of an insecure or infected device, such as Prof. B’s laptop, bypassing the firewall and potentially compromising the IEEE network from within. To mitigate these security risks, the IEEE network could deploy more sophisticated guards like Cisco’s NAC (Network Access Control) [Cisco2003] or UCLA’s QED (Quarantine, Examination and Decontamination) [Eustice2003b] that would prevent the laptop from

accessing network services until it has been scanned and certified as patched and clean of malware. The problem with this scenario is its one-sidedness and the all-or-nothing nature of the interactions. The entering laptop is forced to subject itself to a complete examination, or it is not allowed to join the network at all. The network is allowed to impose its policies on the laptop, and it need not consider the laptop's privacy constraints. For example, Prof. P may not care for access to any conference room services beyond being able to access email using limited bandwidth. In exchange for the permission to perform a much less intrusive search on his laptop, he could agree not to run certain classes of applications, such as instant messaging and file sharing. The laptop would be prevented from accessing all computing resources within the network and prevented from communication with other computers on the network. Unfortunately, this result would require an IEEE system administrator to manually reconfigure both the network and the laptop. Currently available tools do not enable an automated bilateral agreement.

This scenario illustrates the three-way security, privacy and usability tradeoff discussed earlier. A happy medium can be achieved in this scenario, and our research yields a solution that produces this in an automated manner.

#### **1.3.4 Peer-to-Peer File Sharing**

File sharing using free or commercial P2P software is largely client-centric and ungoverned. Most research in this area has also focused on efficient data and network structures, and fast search and data retrieval. There are a number of meta issues such as bandwidth and disk usage, as well as security and access control, which are not given due



importance. The reality is that most file sharing goes on within administrative domains that would like to control how and with whom file sharing occurs, in situations where an outright ban would be undesirable or infeasible.

For example, Bob carries his PDA to his friend John's house. The PDA attempts to run a file-sharing application whenever it can obtain Internet access. John's network will allow Bob's PDA to join and access the Internet, but it has limited bandwidth. John considers P2P to be a frivolous activity, and is not willing to provide more than  $x$  units of bandwidth to any device wishing to run such an application. On the other hand, if the device chooses not to run P2P software, no bandwidth limit is imposed. If a P2P application is found to run on Bob's device, it would immediately be disassociated from the network. John is also unwilling to risk lawsuits in case Bob's PDA downloads and shares illicit or copyright material. His network has a blacklist consisting of a set of known web addresses that are known vendors of copyrighted media. Also, he does not want Bob to download media files that include songs and videos larger than 5 MB. In turn, Bob has a list of preferred web addresses he would absolutely like to share files with. John's network and Bob's PDA could then exchange this information and reach an agreement whereby the P2P application works under the constraints that the network imposes, and the network in turn permits the kind of file sharing Bob prefers, as long as it does not violate its policies. If Bob's device is found to violate these policies at any time (through any available monitoring mechanism), its privileges are immediately revoked. John's network may also choose to provide a proxy P2P service, whereby it could check the veracity of the files that are uploaded or downloaded. Such a range of flexible

agreements could be achieved in an automated manner, as in the other scenarios discussed previously. Our research addresses this type of scenario as well.

#### **1.4. Assumptions**

Certain assumptions about ubiquitous computing environments and the state of technology predated our research, and our implementation was predicated upon them. These are briefly described as follows.

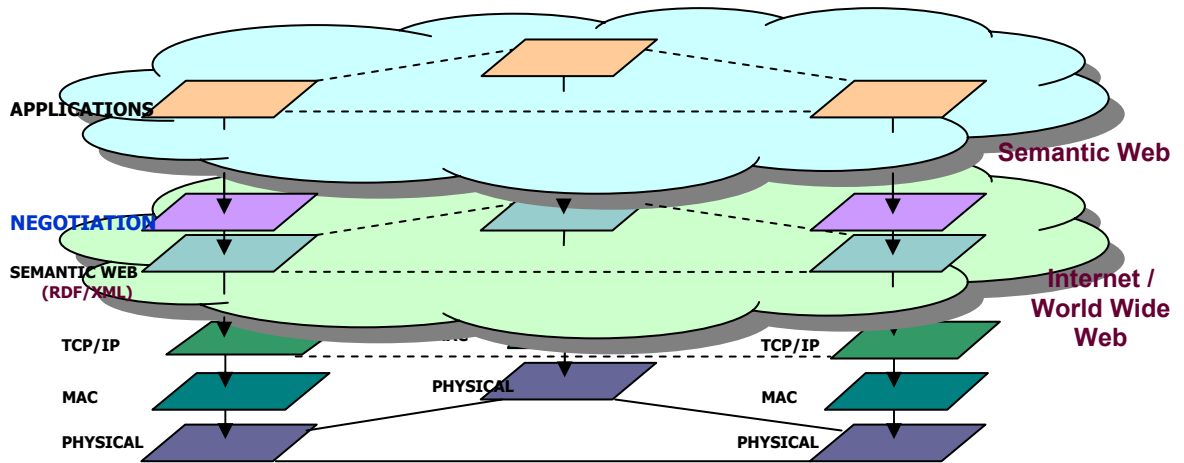
A global ubiquitous computing environment governed by a centralized administrator is infeasible and impractical both from a technical and logistical standpoint. Even though we are seeing companies like Google tie up with cities to provide ubiquitous WiFi access [Google2005], these will most likely be restricted to network connectivity, and local businesses will independently offer different services and have different trust relationships. A large number of scenarios involve resources and services that are only usable in a local area, e.g., printers and displays, so the standardization that cellular providers have achieved for data communication services is unlikely to be translated to the more general ubiquitous computing arena.

We assume that the global ubiquitous computing system of the future will be a vast inter-network of local domains (a term we will define more precisely in Section 2) that are completely autonomous and are able to specify policies that govern all activities within the scope of that domain. What we envision is that local environments will choose to configure themselves in the mold of the smart spaces described earlier, such as Oxygen or Gaia, while connecting to the outside world through a network that will be built on our

contemporary Internet. Our implementation, for example, is built on and is a part of the Panoply middleware [Eustice2008b], as Panoply's model most closely corresponds to our notion of administrative domains.

We also assume that such domains keep the nature of these policies private by default, since exposure of the policy constraints to untrusted entities may leave their resources open to abuse.

We assume that low-level networking compatibility exists. De facto standards such as TCP/IP- and 802.11-based MAC protocols already enable mobile devices to connect and obtain services for their users. Protocols for secure communication, such as TLS, are also fairly ubiquitous. We note that our implementation is a prototype, and it could be augmented by adding other communication mechanisms, such as Bluetooth. Our implementation deals entirely with middleware and application layer compatibility and is independent of the underlying data communication mechanisms. In this respect, we also assume that some standardization of Semantic Web technologies is also inevitable, which would enable diverse systems to share a common description ontology. We believe that the widespread use of XML, a Semantic Web technology, justifies our assumption. Our assumptions, and the position of our negotiation framework, are illustrated in Figure 3.



**Figure 3.** A Policy-Guided Negotiation Framework as a Middleware

Lastly, our framework could be justifiably criticized on the grounds that it adds something more that devices need to agree on, or have deployed, before they can interact. Our claim is that the set of functions we have identified (described in Chapters 2 and 3) as being part of a policy negotiation framework is probably the bare minimum needed for any kind of ubiquitous interaction, and therefore some such framework will be sorely needed in the near future. We will also justify this assumption in Chapters 2 and 3.

## 1.5. Thesis and Research Contributions

In this section I state my thesis on which this research is predicated. Following that, I list the contributions that support this thesis.

### 1.5.1 Thesis

My thesis can be described in three parts, as follows:

- 1) Interoperation among devices and domains in the absence of a preestablished trust relationship or a common set of service (or application) level protocols can be achieved through a generic negotiation protocol.
- 2) All devices and domains have established local policies that constrain the way they can use and export their services. Negotiation only requires that participants have a common understanding of resource semantics, and that these policies be specified in a declarative logical language.
- 3) A framework for policy management that monitors internal changes as well as mediates interactions through negotiation can be designed and implemented to work for a generic ubicomp unit, be it a single device, a group, or a distributed system.

### 1.5.2 Contributions

- **General Purpose Negotiation Protocol for Exchange of Information through Speech Acts**—The most significant contribution of our research is a minimal and lightweight protocol that computers and groups of devices can run, enabling them to interoperate, or as described earlier, query each other for information, exchange data, offer and consume resources. The high-level message types are generic speech acts, such as requests, offers, queries, commands and policy obligations, which is a general enough set sufficient for all the varieties of ubicomp applications and scenarios we could think of. The protocol itself is completely independent of the type and number

of resources and credentials possessed by each negotiator, and the nature of the security and resource usage constraints represented by the policies. The interacting entities don't need to have much in common, apart from sharing a common ontology and an understanding of how to process speech acts in the form of negotiation messages. A negotiation could result in failure, but that would only be because the negotiators' policies are completely incompatible. Our negotiation protocol is sensitive to runtime context and policy changes, and supports simultaneous negotiation for multiple goals. It goes much further than traditional fixed goal and domain-specific negotiations such as DHCP or QoS-guaranteeing network protocols.

- **Decentralized Policy Resolution for Agreement/Contract Generation—**  
Autonomy of computing systems and differences in policies could prevent agreements from being reached in the absence of a procedure through which both parties find out more about the each other. Were the parties to surrender their privacy to some centralized entity, it would have sufficient knowledge to generate a satisfactory agreement by resolving the sets of policies. But this may be unwise in most scenarios for privacy reasons, and because none of the interacting parties can agree on a trusted third party. Our negotiation framework processes negotiation messages and decides what proposals can be made and what information can be revealed; we model this as a decentralized and distributed policy resolution procedure. It works on partial knowledge, and is thus less efficient than centralized policy resolution, but offers more privacy protection. Also, given suitable restrictions on the policy language (as we will see in Sections 3, 4 and 8) the results may be

identical in quality to the centralized version. We also show, through large-scale testing, how negotiation performs in comparison to centralized policy resolution along a number of dimensions; these are described extensively in Chapter 9. To the best of our knowledge, such a performance study has not been attempted before.

- **Design and Implementation of a Policy Management Subsystem for Groups of Devices in a Ubiquitous Computing Environment**—We designed and implemented a policy management framework as a core feature of Panoply [Eustice2008b], a generic group-management middleware for ubiquitous computing. Our design principles were faithful to our top-down, holistic approach to thinking about ubicomp interactions, as indeed, were the principles that inspired Panoply. A policy manager in a Panoply *sphere of influence* (we will define this term in Section 6) is effectively the security manager for a Panoply-enabled computing system. It maintains system state and policies in a database, is sensitive to context changes by monitoring *events*, triggers suitable actions as specified by relevant policies, mediates both intra- and inter-sphere interactions, and filters information flow to applications. Lastly, policy management promotes a *non-intrusive* paradigm whereby computers, given suitably specified policies, can do users' work without requiring constant manual intervention.
- **Dynamic Context-Sensitive Access Control through Negotiation**—Our negotiation framework also inspired a procedure for access control which is more dynamic than the traditional procedure of local access control policy lookup and verification. The implementation is tied to the Panoply model of *event* communication (we define Panoply events in Section 6). The Panoply policy manager filters events to test

whether or not events are permitted to flow to the destination application, which is conceptually equivalent to testing access control permissions, as Panoply resource accesses occur through events. If the event is found to violate policy, a challenge-response (through a negotiation) could ensue. Resource accesses mediated by the policy manager will not violate access policies even though they may change dynamically. As we show later, negotiation results could be cached for a limited amount of time to ensure better performance, though revocation and integrity issues arise (discussed in Chapter 6). Our access control framework is, as we justify later, more dynamic and effective than recent advanced models like GRBAC [Covington2000] and DRBAC [Freudenthal2002].

## 1.6. Dissertation Outline

The remainder of this dissertation is organized as follows:

**Chapter 2** describes our model of local ubiquitous computing environments, or domains, consisting of groups of devices, resources and networks, in more detail. The role that policies play in the management of such domains is also described, as well as why such policies are needed, and what the characteristics of such policies should be.

**Chapter 3** describes our model of interaction among two or more domains. Following that, the concept of *negotiation* as a way of enabling such interaction is introduced. The results of negotiation, as well as the theories backing such negotiation, are discussed.



**Chapter 4** describes our policy language and the semantics of a policy database, or collections of policies.

**Chapter 5** describes our negotiation protocol as implemented in a real-world system on the basis of local policies. Protocol messaging units, semantics, and the algorithms at the message processing back-end are described.

**Chapter 6** describes the implementation of a full-scale policy manager within Panoply, a real-world ubiquitous computing middleware. The negotiation support provided by the policy manager is emphasized, and the larger roles of policy management are also described. These include action triggers upon events, and a dynamic form of access control through mediation of information flow. The systems issues in making our protocol fault tolerant are described.

**Chapter 7** describes various Panoply applications, some of which benefited from the use of policy management functions, and others that were built to demonstrate versatility.

We discuss the characteristics of the negotiation framework from a theoretical standpoint in **Chapter 8**, and from a practical performance standpoint in **Chapter 9**.

**Chapter 10** discusses the related work in the areas of negotiation and policy management, and also discusses complementary research in policy languages, the Semantic Web, and security and access control frameworks.

**Chapter 11** describes suggestions for future work, and our conclusions are described in **Chapter 12**.

## Chapter 2

# Policy-Governed Domains and Cross-Domain Interoperation

In Section 1.4 we showed why it was reasonable to assume some level of standardization and compatibility at the lower networking layers and in the lower Semantic Web layers. The purpose of networking standardization is to enable two computers to communicate raw data, and the ostensible purpose of Semantic Web standardization is to enable agents and applications to communicate. The purpose of our research, which involves policy management and negotiation, is to enable interoperation among ubicomp *domains*, a word used multiple times in the previous section. Below, we describe by example what we mean by the notion of a domain, and the role that policies play in a domain.

### 2.1. Devices and Groups as Administrative Domains with Scoped Policies

We refer to any computing element that can act autonomously and has a fixed and defined administrative boundary as a *domain*. These can come in a variety of shapes and sizes, yet have basic common characteristics and interaction semantics. The following seemingly diverse computing systems fit in our definition of a domain:

- A single device, which could be a desktop PC or a mobile device like a PDA or cell phone. Such devices possess resources that range from computing power, disk space,

and networking bandwidth to media files, digital credentials and agent code. They run applications like instant messengers and services (by default, usable only by that device) like location sensors and VoIP. The owner of the device subjects it to his needs and constraints.

- A group of computing devices connected by a local network. These can range from tightly coupled device clusters to distributed systems to independent devices connected temporarily through a LAN. Examples include: enterprise networks in offices, department and lab networks within a university; server farms, clusters of computers offering a service or performing a task in a collaborative manner. Coffee shop wireless networks, and a personal area network connecting a user's laptop, cell phone and e-watch, are also real examples of such domains in a ubicomp world. Such domains consist of computers as well as resources (such as printers, speakers and displays) that are not tied to a computer. They could offer a wider range of services such as Internet connectivity, caching and proxy services. The conference room described in Section 1.3 describes such a domain offering display, print and Internet services. The constraints that dictate the behavior of devices within and the rules of resource usage are typically set by a system administrator.
- Organizations or social networks that are bound by a shared agreement or contract, where the set of participant devices is neither static nor confined to a physical region. Examples include organizations like ACM and IEEE, schools like UCLA and companies like Intel. The human members of these organizations are more relevant to such domains, rather than the computing devices themselves. The use of computing

devices enters the picture as a storehouse of credentials that certify affiliation and the procedures for accessing group services through web protocols. The kinds of services provided (such as access to online documents) and the policies that govern organizational membership are shared characteristics of these kinds of domains.

We can observe from the above examples that these different types of domains have well-defined boundaries, and serve as containers of resources and services that are accessible within the domain but are by default inaccessible to any computing entity outside it. Centralized control exists within a domain with all behavior being constrained by a set of policies, whose scope extends only to the boundary of the domain. In addition, each domain maintains state or contextual information that is relevant to its needs. As the examples indicate, domains are not mutually exclusive; they may intersect, or lie within other domains. Intuitively, computing domains are analogous to administrative domains in the real world, consisting of local self-governing communities bound by a law whose effect extends to the communities' geographical boundaries. A related project, *Law-Governed Interactions* [Minsky2000], uses a similar definition of domains, and we will discuss this further in the Related Work chapter. Our notion of domains corresponds to, and is borrowed from, the Spheres of Influence concept, which is supported by a middleware called Panoply. An introduction and study of this concept is presented in a PhD thesis by Kevin Eustice [Eustice2003b], who collaborated on the research presented in this dissertation.

### 2.1.1 Interactions Among Domains

In the scenarios described in Section 1.3, we encountered a number of domains that fit into our above definition. The conference room, the party network, the IEEE network, the home network, and each mobile device, had clearly demarcated boundaries, services and policies. Often, a mobile user might carry multiple devices like a laptop, cell phone and smart watch connected through a personal area network. The scenarios therefore describe what happens (and what should happen) when two domains try to interact. Modeling spontaneous interoperation [Kindberg2002] is easier with our definition of a domain. In this model, we treat each interacting domain as a single virtual device. The virtualization mechanism is beyond the scope of this dissertation, though Eustice describes it in his thesis [Eustice2008b]. Examples of interactions include the following:

- The simplest case: two computers communicating (e.g., to share files)
- A mobile device interacting with a local network in proximity; for example, a conference attendee's device interacting with a conference room network
- Two networks interacting; for example, a party guest's personal area network interacting with the party network
- Two organizations interacting for the purpose of creating and changing agreements, which would be relevant to entities who are members of both organizations. For example, ACM might make an agreement with UCLA to allow UCLA terminals to be used to read ACM conference papers; any change in the agreement will trigger a change in the relationship and constrain the ways in which UCLA students can access ACM resources.

The nature and purpose of interaction among domains, though of a diverse nature, remains identical at a high level: they all involve service discovery and service (and resource) access. The domain-dependent variable is policy, which every domain is free to set as it chooses, and which guides the decision making of the infrastructure during interactions and also local resource management. As an extreme case, a completely open system without any security or privacy issues would allow interactions based on a *null* policy. Our premise, one which we justified in Chapter 1, is that different domains in a ubiquitous environment are not likely to fall under a common security and trust umbrella, and so making them obey similar guidelines and policies is impractical for the same reason that centralized management is. Still, ad hoc associations must be supported and users carrying their personal devices need to access computing resources and be able to network wherever the presence of such infrastructure makes it possible.

An alternative way of handling dynamic interactions exists. We could treat every ubicomp application on its own merit, and have each domain that wishes to support that application configured with relevant policies and mechanisms. Unfortunately, this solution cannot then be ported to a different application, and all the common features would have to be re-engineered whenever one needs to create and deploy a new application. Also, this would not be an effective way of handling situations that warrant dynamic formation or changes of agreements. Our domain-oriented approach provides a more tractable and scalable solution compared to an application-oriented approach, not least because of the huge number of combinations of possible policies and contexts.

## **2.2. Policies and Their Role in Ubicomp Domains**

Most domains, including single devices, will have certain common features such as networking, display, and audio capabilities. In fact, given a basic networking and processing capability, all resources could, in theory, be obtained and accessed across domain boundaries. Most devices that mobile users carry around will perform a small number of specialized functions through off-the-shelf components. There are mechanisms available to do almost anything that is computationally tractable; the number of ways to make use of resources and system capabilities is huge and is increasing. Ubicomp interaction will be based on the principle of being able to find and use such mechanisms wherever available, because user devices will have neither the hardware capabilities to perform every conceivable task nor the storage capacity necessary for all the services and data that their owners will find useful.

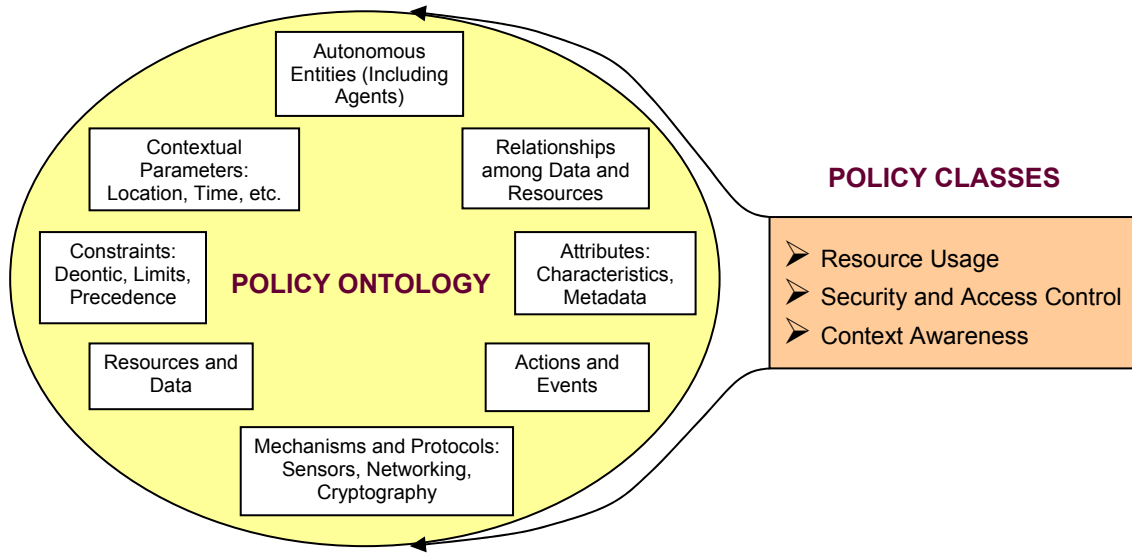
So why cannot such domains, with their shared characteristics and needs (as described in Section 2.1), freely interoperate? It is precisely because two domains that wish to interact do not know if the other possesses relevant resources or resource access mechanisms or if their policies are compatible. Potentially, any required mechanism can be discovered from the external world and used, but how those mechanisms can be used and offered is not a trivial issue, given trust and system integrity concerns that every domain will have. These concerns fall under domain policy, and so we see that policies, though a shared common feature of domains, are an obstacle to interoperation. To aid in both internal management and interoperation with external domains, policies should be stated in a proper way and must provide an adequate description of domain constraints.

This premise both inspired and drives our research; it will be validated through our design and the results will be discussed in later chapters.

### **2.2.1 Policy Expression and Scope**

Policy is essentially an abstraction, or a set of rules that constrain how a system can behave and how it ought to behave. It is a set of factual and behavioral specifications that are binding on every computing element and resource within a domain. Policy must specify entities (as represented by computing devices) and their attributes, security and privacy constraints, trust relationships, security credentials, network types, resources and protocols, cryptography-based objects and protocols, data and content types, and contextual parameters like time and space (see Figure 4 below). This list, which is fairly comprehensive, must be supported in a general-purpose ubicomp middleware. Some of the more general requirements for policy are the ability to deal with groups and classes of objects, and specification of *general* behavior and *exception* conditions. Deontic concepts like obligations and permissions [Kagal2003a], meta-constraints for priorities, resolution of modality conflicts, and setting up precedence ordering among different policy rules are other general requirements.





**Figure 4.** Policy Scope and Classification

Not every domain needs to specify policies describing all sorts of resources, contextual parameters, and groups of entities that exist in a ubiquitous computing environment, though it should have the ability to do so if required. For example, a domain could have policy rules that describe its knowledge about objects that it is aware of (such as computer identities as IP addresses, location and time parameters, resources like storage capacity, printers and displays) and the relationships among such objects.

Still, different domains need to agree on some issues in order to interoperate, unless their administrators and users wish otherwise. The bare minimum that must necessarily be common, while leaving individual users the maximum independence, is a shared policy description framework (a common syntax for a high-level description of resources of shared interest), a way of describing common resources (a common global ontology), and a communication protocol for transaction based on those policies. Domain-specific data and objects do not need to be understandable across domains, but

the policy language should support description of both domain-specific and global objects and constraints. In the future, some resource types, protocols or policies may get standardized on a global scale because of wide usage. Our research will still be useful in that situation because it would help in settling the terms concerning use of resources as constrained by policy rules, which may still differ from one domain to another.

### **2.2.2 Classification and Applications of Policies**

As illustrated in Figure 4, we classify the use and applications of policies in three broad, high-level areas: resource management, security and access control, and context-awareness. In our view, based on a study of ubicomp literature and by observing the uses that ubiquitous systems and middleware have been put to, this is a fairly exhaustive classification. More relevant to our research is the fact that all three classes of policies impact the nature and result of an inter-domain interaction.

This classification serves only to illustrate the aspects of a domain that users and administrators might be interested in controlling, and for which policies would be enforced in an automated manner. Individual policy rules may not fall strictly under one class or another; for example, it would be easy to frame a composite policy that specifies *“how much of a resource can be used by a user accessing it through an insecure channel in a particular location.”* Our ontology (see Figure 4) provides a more basic classification that is used to compose policies. No policy framework has rules for resources, security and context awareness written in complete isolation from each other. Every domain will maintain a database of policy statements, including facts and rules of behavior;

statements can be added to and removed from this database by the users or by the system. The interplay of these different functions can get very complex in a dynamic and heterogeneous environment.

#### **2.2.2.1 Resource Management**

Every domain possesses resources ranging from high-level ones like printers, displays, audio devices, sensors, actuators, and network connections to more basic resources like amount of bandwidth, disk space, files, data, and even memory blocks. It must export interfaces for such resources to users and applications. A single device manages resources through an operating system, and a network or cluster can manage its resources in many ways, ranging from a distributed operating system with tightly coupled devices to a loose federation of devices that share a single server or gateway to the outside world. Policy can be used to describe and constrain the way each of these resources can be used, and to perform resource allocation when multiple clients or applications have similar requests. It can describe how the usage of one resource is dependent on (or constrained by) another. High-level resources usually are dependent on lower-level resources, and any actions that are requested by clients could impact the behavior of multiple resources at different levels. Also, requests for resources at a high level, common when neither the requester nor the owner have knowledge of each other's possessions, could be translated into requests for access to lower-level resources at the owner's end. For example, Bob carries a PDA with him to a coffee shop expecting to get Internet connectivity and be able to use a particular network protocol through the shop network's gateway. The shop

network's policy for connectivity and protocols impacts low-level resources like network bandwidth and the amount of buffer space it has available. Therefore, what is a simple requirement from the PDA's point of view involves a more complex interplay of policy rules governing resources at the network's end. In a highly dynamic environment, with the number and nature of clients in constant flux, a policy-based framework would be necessary to enable interoperation and to make sure that desired system behavior is exhibited. Such a system could be used to monitor conflicts and either solve them using meta-policies, or report them through appropriate interfaces. Systems like Keynote [Blaze1999] do such conflict checking, though in a static manner and using a restricted policy language. Our research has dealt mainly with variable high-level policies (set up by users) and their interplay with more static low-level policies in different contexts.

#### **2.2.2.2 Security and Access Control**

Security and access control policies often define the boundary of a domain, and are based on a local measure of trust and an idea of what it could take to compromise a system and misuse its resources. Examples of security policies include filtering remote service access based on identity and port (this is done using firewalls), and memory and file access restrictions to prevent buffer overflow attacks and mitigate the threat of viruses. Access control and privacy policy rules are used to answer questions like “*who is allowed to access a particular resource?*” and “*what kind of authentication is necessary?*” In ubiquitous computing, mutually unknown computing entities need to have a certain measure of *trust* in one another before they are ready to interact and share resources.

Access control policies will be specified using a local idea of trust, since it is impossible to establish trust relationships between every possible pair of devices in the world. As with resource management, it should be possible to describe security and access control rules at a high level, with the policy manager deciding what mechanisms to use in an ad hoc manner. This approach is highly flexible compared to many existing security frameworks that would require re-engineering when faced with new requirements. Different resources may have different access policies, and it is impossible to anticipate all permutations and side effects of these beforehand. Also, higher-level security policies (like setting security levels in the Internet Explorer browser) could impact behavior at a lower level (for example, which ports must be opened). Meta-policies, such as those pertaining to security/privacy conflicts, could help a policy manager with decision making. For a system in which new modules could be added dynamically, policy can be used to monitor the security impacts and prevent violations. Often, it might be important to have local policy rules kept private, since the exposure of these may inadvertently release private and sensitive information. This could enable malicious entities to discover security holes or make use of the nature of the policy itself in order to mount attacks. For example, domain A may have stringent policies for access to its resources (requiring intrusive checks) but may also provide such access to entities that are certified by domain B (which is more lax than A) and present in the vicinity. The latter is a much lower bar for entities to pass, and it is not in A's interest to let everyone know that. In another scenario, if A has a policy of allowing resource access to entities affiliated with organization X between 8 pm to 9 pm, an attacker could attempt denial-of-service attacks

on A and X within that timeframe; knowledge of A's policy would result in a more effective attack with fewer resources expended by the attacker.

### **2.2.2.3 Context-Awareness**

Context-aware computing makes computer systems more intelligent from the point of view of a user since they provide customized service to users based on users' *context*. In a ubiquitous computing world, the same application or resource provider would behave differently based on its perceived context. A context-specific application of a general policy would resolve to a set of lower-level policies. Learning a user's behavior and anticipating his needs is an artificial intelligence problem, but there are systems issues in determining exactly what the context is and what is of relevance to the current context. Here, context can be used but as an added dimension to resource management and security policies. Location and time are the most common and widely used contextual parameters. For example, if I am in my car and need to find a gas station, I would like the application on my PDA to find the closest gas stations, rather than give me a complete list or prompt me to specify my location. Here the policy simply states that a gas station be located when gas runs out; the device has some way of sensing gas level, or could get such information explicitly from the user through some interface. I might like my home TV to change maximum volume levels gradually based on time, or prevent R-rated content from being screened during certain hours. Other types of contextual parameters could be considered, like applications behaving differently on my PDA based on whether I am in a public bus or in my car. Of course, for a truly intelligent environment, any

system would require sensors that provide context information; various sensors are in use today, and innovative ways of using them is the focus of a lot of ongoing research. Lastly, any policy framework that supports context awareness must allow generalizations over objects and context to be specified as well as exceptions, which can be used by systems to make quick and appropriate decisions. In practice, it is usually not necessary to *completely* specify context-adaptive behavior, which may be an impossible problem; partial specification and probabilistic reasoning could be used for decision making. Policies should be extrapolated to a context when direct rule lookup is not possible or cannot be inferred. Our expansive view of a policy management framework, which includes observation and recording of state changes, enables such extrapolation, as will be seen later in Chapter 6.

### **2.2.3 Policy Management in a Domain**

We have seen how policies can be indispensable in a domain's internal and external functions. To harness their power in an automated manner, policies need to be specified in particular ways and should have certain defined characteristics. Automated handling and manipulation of policies requires a policy management subsystem or infrastructure for every domain, which maintains a database consisting of all the state information and behavioral rules that comprise that domain's policy.

**Policy Characteristics:** Any system that we use to describe and reason about policy must have an ontology that defines *what* policy can specify, and also *how* it can specify

the elements of that ontology. We just described the “what” part, which is also illustrated in Figure 4. With respect to the “how” part, we assert that such policies should be *declarative*, and that such policies can be specified independently by users and administrators alike, leaving dependencies and conflict management to the subsystem. This also requires that policies have a logical underpinning, because logical reasoning frameworks provide formal and well-researched tools for the analysis of a collection of policies and the management of their inter-dependencies.

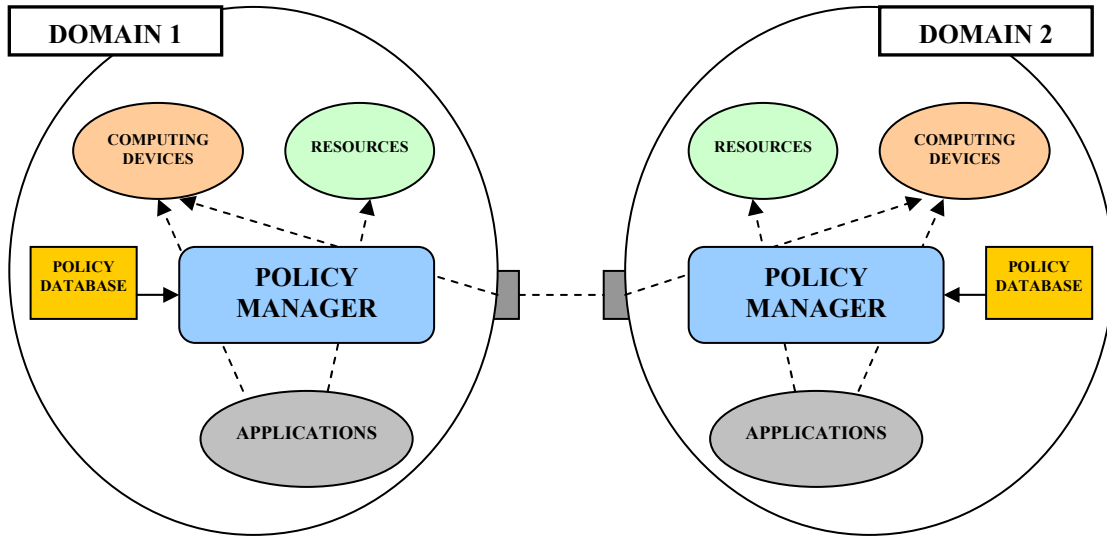
Let us look at the alternative to declarative, logical policies, and our reasons for rejecting it. A domain administrator with sufficient programming skill could write up an application that implements the collective system policy. But this approach is obviously less flexible, as it ties policies to mechanisms; an application is written based on a particular specification of what is to be done (*policy*) **and** how to do it (*mechanism*). Using a declarative policy language, both online (i.e., while the system is running) and offline changes, and the addition and removal of policies are possible; in contrast, an application (whose implementation combines policy with mechanism) will have to be examined, modified, recompiled and redeployed. A number of policies will be specified by naïve users; neither will such users be able to modify system applications nor can designers anticipate all possible preferences that users might have, especially since such preferences may have a cross-application impact on the system [Zhu2005b; Toninelli2006; Agrawal2005]. A policy manager also allows a modular approach, so that resources can be added or removed easily, and the manager takes care of resolving newly added policies with existing ones. Policies must therefore be easy to write at a high level,



and can be ambiguous in certain ways, notably related to context. It is the policy manager's task to clear ambiguities and make context-dependent decisions.

**Policy Engineering:** From an engineering perspective, a policy management subsystem must deal with domain-specific requirements while targeting interaction from a global perspective. Since every domain will possess policy, but not possess every possible mechanism to deal with every possible situation, this subsystem must necessarily favor *policy* over *mechanism* for flexibility and extensibility. It should have the ability to reason with a set of policy rules and provide some guarantee of correct results when action decisions are made on the basis of local policy. The presence of such a framework precludes the need to invent specialized protocols and mechanisms for diverse security and resource access requirements.

Structurally, a policy manager would lie as a middleware between the operating system layer and the application layer, being independent of both. It would mediate applications' access to a domain's computing resources and interactions between two or more domains. Our conception of a policy manager within a domain is illustrated in Figure 5.



**Figure 5.** Domains with Policy Management Middlewares that Mediate Interactions

The variety of policies that such a middleware might need to handle and the range of issues involved in managing a collection of declarative policies looks like a huge burden for a research project. But we can leverage a large body of research in formal models or schemas for policy description and inference as well as formal logics, databases, and Semantic Web tools like RDF/XML. For implementation and evaluation purposes, we will select certain representative scenarios and ubicomp applications, and will also illustrate how these can be extended to other environments for which new policies can be written without having to reimplement the infrastructure.

Using policy to control system behavior is not a novel concept introduced by us. Most systems use policy for flexibility and extensibility and also to impose constraints on the usage of system resources. While traditional uses of policy have been domain-specific and meant for local interpretation, I propose to use policy as a tool for ubiquitous bidirectional interoperation.

# Chapter 3

## Negotiation Concepts

We have seen how seemingly incompatible policies of interacting domains could prevent them from reaching mutually beneficial resource sharing agreements. In this chapter, we model such interactions in terms of the resources, services and policies possessed by domains. We describe how to develop interactions around the concepts of negotiation, thereby leading to the desired agreements illustrated in the scenarios in Chapter 1.

### 3.1. Interaction Model

Based on the example scenarios described in Chapter 1, and our definition of interoperating domains described in Chapter 2, we defined interoperation as a dynamic procedure discovering services and resources as well as obtaining access to them. We formalize the process of inter-domain interaction in this section. Each interaction starts off with domains possessing resources, policies and goals, and ends with a working agreement or relationship, consistent with the participants' policies, that involves cross-domain service and resource access. In practice, such an interaction is initiated by the need to obtain resources not available within the local domain but which are offered by the other. The set of resources and services that are ultimately shared is expected to be larger than the original goal set, including pieces of data and credentials that are necessary for fulfillment of the policy constraints. An agreement that results in a

satisfaction of the original goals may or may not be possible; the latter is an acceptable result if satisfaction would entail the violation of the policies of one domain or the other.

Interaction is not restricted to two domains. Multiple domains may interact simultaneously, but the dynamics remain the same. We outline below the two-party model, and then briefly mention what an  $n$ -party case would look like.

### **Pre-Interaction:**

- We have two computing domains,  $D_1$  and  $D_2$ , which could be devices, groups of devices, or small networks that can operate autonomously (See Figure 6).
- $D_1$  possesses a triple  $\langle S_1, P_1, G_1 \rangle$ 
  - A collection of resources, services and data,  $S_1$ .  
 $S_1$  is general enough to incorporate high-level services like printing or display, and also low-level items like individual data items, memory space.
  - A set of policies,  $P_1 = \{P_{11}, P_{12}, \dots, P_{1m}\}$ ,  $m$  = number of policy facts and rules.  
 $P_1$  is general enough to describe both the state of a device or a network, values of relevant context parameters like time and location, operating rules that describe access control of resources and data objects, resource allocation, use of security mechanisms, context-sensitive behavior, and deontic concepts such as obligations.  
If  $C_1$  is a network,  $P_1$  would include information about resources allocated to or accessible to individual computers within the network.
  - A set of goals or requirements,  $G_1$ .

$G_1$  describes a set of resources, data or services that are not available locally at  $D_1$  but which are necessary for the performance of local tasks or the running of user applications. These can be obtained from the external environment, which consists of other domains (though  $D_1$  is only interacting with  $D_2$ ).

- Likewise,  $D_2$  possesses a triple  $\langle S_2, P_2, G_2 \rangle$ , where  $P_2 = \{P_{21}, P_{22}, \dots, P_{2k}\}$ ,  $k =$  number of policy facts and rules.
- Local private policy assumption:
  - By default,  $D_2$  is assumed to have no knowledge of  $S_1$ ,  $P_1$  or  $G_1$ .
  - Likewise,  $D_1$  is assumed to have no knowledge of  $S_2$ ,  $P_2$  or  $G_2$ .

### **Interaction Procedure**

- $D_1$  attempts to find and obtain all or part of the resources specified in  $G_1$  from the set of resources  $S_2$  possessed by  $D_2$ .
- Likewise,  $D_2$  attempts to find and obtain all or part of the resources specified in  $G_2$  from the set of resources  $S_1$  possessed by  $D_1$ .
- Both attempts (discovery leading to obtaining of access) occur concurrently, with no imbalance of power or influence between  $D_1$  and  $D_2$ .

### **Post-Interaction: Outcome**

- The result of  $D_1$  interacting with  $D_2$  is a relationship or agreement as follows:
  - $D_1$  gains access to a set of resources  $Q_1 \subseteq G_1 \cap S_2$ .

From  $D_2$ 's point of view,  $D_1$  gaining access to  $Q_1$  (we can represent this as '*grant-*

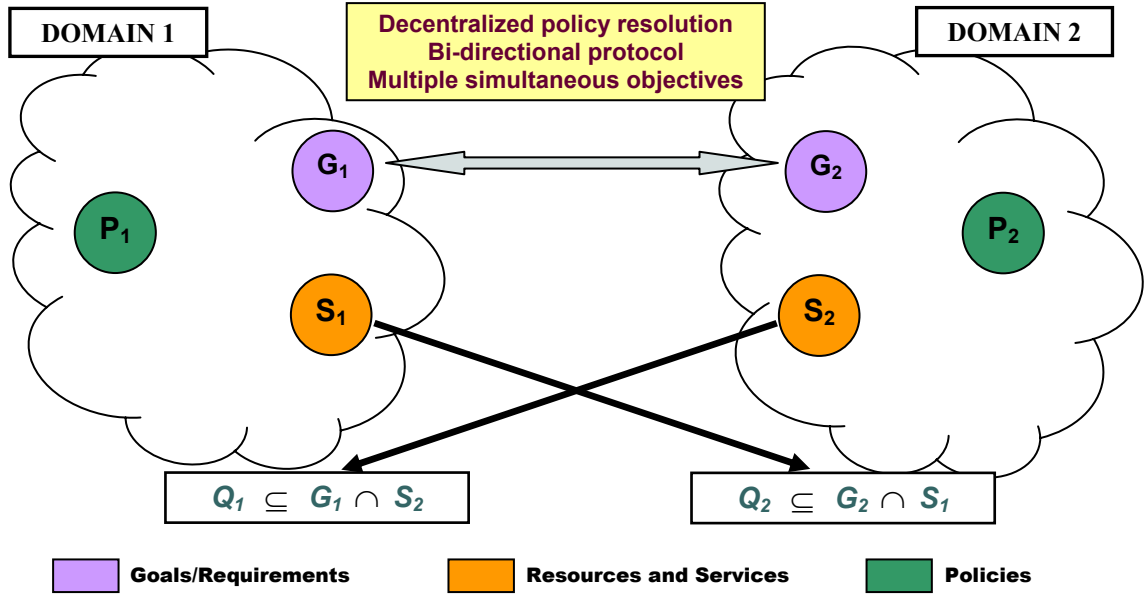
$access(D_1, Q_1)$ ' is consistent with  $P_2$ ; i.e., when  $grant-access(D_1, Q_1)$  is suitably framed as a policy statement,  $grant-access(D_1, Q_1) \wedge P_{21} \wedge P_{22} \wedge \dots \wedge P_{2k}$  is not a contradiction.

$Q_1$  is a subset of  $G_1 \cap S_2$  because some elements of the goal set  $G_1$  will be granted and others not; also, some goals will be satisfied partially or at a degraded level (e.g., the amount of bandwidth granted is lower than the requested level).

- $D_2$  gains access to a set of resources  $Q_2 \subseteq G_2 \cap S_1$ .

From  $D_1$ 's point of view,  $D_2$  gaining access to  $Q_2$  (we can represent this as ' $grant-access(D_2, Q_2)$ ') is consistent with  $P_1$ , i.e., when  $grant-access(D_2, Q_2)$  is suitably framed as a policy statement,  $grant-access(D_2, Q_2) \wedge P_{11} \wedge P_{12} \wedge \dots \wedge P_{1m}$  is not a contradiction.

- $Q_1$  and  $Q_2$  are sets of resources obtainable within the constraints imposed by  $P_1$  at  $D_1$  and  $P_2$  at  $D_2$ , and the partial knowledge possessed by either entity. It may not be the case that all the goals of either domain can be satisfied by the other based on what it possesses, which is why the maximal resource binding is the intersection of the sets  $G_i$  and  $S_i$ .



**Figure 6.** Inter-Domain Interaction Model

- In an ideal (or optimal) interaction outcome,  $Q_1$  and  $Q_2$  are maximal sets within the constraints imposed by the respective policy sets, i.e., any larger set would be inconsistent with the policies. An oracle that has knowledge of both sets of  $S$ ,  $P$  and  $G$ , and which can run a centralized policy resolution procedure, could generate such an outcome. But it is not obvious that such optimality can be achieved through a decentralized procedure and with partial information. A deeper analysis and comparison (through practical measurements) are presented in Chapters 8 and 9.

Figure 6 illustrates our ubicomp interaction model, clearly showing the inputs and the outputs. In a large class of scenarios, only one party (e.g., a mobile device discovering and joining a network) starts off with goals. Let us map our conference room scenario

from Chapter 1 onto this model. We add some features not present in Figure 1 just to make the example richer.

➤ Interacting domains:

- D<sub>1</sub>: PDA-Phone carried by the conference attendee
- D<sub>2</sub>: Conference room network that offers various services

➤ Interaction goals:

- G<sub>1</sub>: {*Web access, Projector display service, Print service*}
- G<sub>2</sub>: {*Expand organization (IEEE) subscription base*}

➤ Domain resources and services:

- S<sub>1</sub>: {*UCLA-certified voucher, IEEE-membership Voucher, Credit card info.*}
- S<sub>2</sub>: {*Network connectivity and Internet access, Print service, Projector display service, Journal subscription service, NSF-granted certificate*}

➤ Domain policies:

- P<sub>1</sub>: {*I will release my vouchers only to NSF-certified domains;*  
*I will block services on vulnerable ports in return for network access*}
- P<sub>2</sub>: {*I will release my NSF certificate to whoever requests it;*  
*I will allow printing access to any valid conference attendee;*  
*I will allow projector display access only to conference officials;*  
*I will allow network and Internet access to conference attendees who provide either a valid ACM voucher or a voucher from an ACM-affiliated school (UCLA being one of them), is willing to block incoming traffic on ports 25 and 110, and is willing to provide specification information about its OS*}



➤ Final Outcome:

- $Q_1$ : {*Network connectivity and Internet access, Print service, NSF-granted certificate presented by  $D_1$ , Journal subscription*}
- $Q_2$ : {*UCLA-certified voucher, Credit card info.*}

### n-Party Interaction

Modeling an interaction among  $n$  ( $n \geq 2$ ) domains can be done in a straightforward manner by extending the 2-party model.

➤ We have a set of domains  $\{D_1, D_2, \dots, D_n\}$ .

➤ Each domain  $D_i$  :

- Possesses a triple  $\langle S_i, P_i, G_i \rangle$ , where  $P_i = \{P_{i1}, P_{i2}, \dots, P_{ip_i}\}$ ,  $p_i$  = number of policy facts and rules.
- Is assumed, by default, to have no knowledge of  $S_j, P_j$  or  $G_j$ , ( $j$  varies from 1 to  $n$ , excluding  $i$ ).
- Attempts to find and obtain all or part of the resources specified in  $G_i$  from the set of resources  $\bigcup_{j=1}^n S_j - S_i$ . All attempts occur concurrently, with no imbalance of power or influence among the  $D_i$ s.
- Gains access to a set of resources  $Q_i \subseteq G_i \cap (\bigcup_{j=1}^n S_j - S_i)$ , so that from the point of view of each  $D_j$  ( $j$  varies from 1 to  $n$ , excluding  $i$ ),  $D_i$  gaining access to  $Q_i$  (we can represent this as '*grant-access*( $D_i, Q_i$ )') is consistent with  $P_j$ ; i.e., when *grant-*

$access(D_i, Q_i)$  is suitably framed as a policy statement,  $grant-access(D_i, Q_i) \wedge P_{j1} \wedge P_{j2} \wedge \dots \wedge P_{jp_j}$  is not a contradiction.

A full (multi-party) interaction involving more than two domains is easy to model but possesses very complex dynamics, because each bilateral interaction affects others in unpredictable ways. A strictly less complex interaction among  $n$  domains has multiple concurrent bilateral sessions, where each interaction proceeds as if it were the only one. We refer to this as a multi-thread interaction, and our framework supports this in addition to the basic 2-party interaction.

### 3.2. Agreement through Negotiation

In the previous section we modeled an interaction as a constraint-satisfaction problem. Disregarding the complexity of solving such a problem for the moment, obtaining a solution is made much harder simply because the inputs (resource and policy sets) for each interacting domain are private by default, and not available to the other domain. Therefore, reaching an agreement through an interaction necessarily requires some exchange of information between the two sides. We provide a procedure for agreement using an application-level protocol, which takes the form of a ***negotiation*** (an *interaction of influences*). Inter-domain interactions have similar characteristics to negotiations:

- The domains have competing influences or interests.
- A dispute arises because of policies that prevent one domain from immediately granting access to all requested resources.

- The final resource-access agreement is the course of action the domains settle on, the set of resources granted access to, and the level of access granted depends on the interests of each domain.
- It is advantageous to gain as much as possible while conceding as little as possible (including the nature of the policies), thereby necessitating bargaining.

At a practical level, we follow Zartman's definition of negotiation as *a process of combining conflicting positions into a common position under a decision rule of unanimity, a phenomenon in which the outcome is determined by the process* [Zartman1993]. Conceptually, our negotiation protocol (described in technical detail in Chapter 5) achieves a tradeoff through a process of give-and-take. What must be given and taken is inferred from the policy rules. Once the other negotiator's goals are known, local policies governing (and relevant to) those goals can be checked for compliance. It is obviously not in the interest of a domain to concede to a request at the expense of violating its policy. Alternatively, obstacles to goal satisfaction can be inferred from the policy statement, and these can be restated in the form of different goals that can be considered in turn. In some cases, the incompatible policy rule could be communicated as a goal; the other negotiator could examine and attempt to satisfy this rule. Negotiation is perpetuated by such trading of information, guided by the examination of policies and the evaluation of competing interests at every step. Satisfaction of a goal (e.g., obtaining resource access) might involve some sacrifice on a negotiator's part, or some behavioral *obligations* could be imposed upon it; in the larger picture, one is willing to make such a sacrifice because a larger interest (of obtaining resource access) is served.

Sometimes, the nature of resources and policies may require a re-evaluation of goals, and the negotiation would result in a compromise. For example, resource access goals may be compromised upon in three ways: (i) negotiation for a quantitatively measured resource (e.g., bandwidth) could eventually settle on a point where the quantity granted is less than or equal to the originally requested level; (ii) negotiation for a resource that can only be granted whole or not at all (e.g., access to a display device) could eventually result in access to an alternate resource that serves the purpose but is less *desirable* in qualitative terms than the originally requested one; (iii) the original request could be granted with certain restrictions on the use of the resource (e.g., it cannot be accessed during the night).

The salient features of our negotiation protocol are listed below:

- It is a **policy-guided operation** through which domains can make a request to each other for access to locally available data and resources in a sequential manner, and decide whether to grant such access. Each domain starts with a goal set, though only one initiates a protocol session. Each party's interests, framed as meta-policies and heuristics, guide the negotiation towards a compromise agreement.
- The final result must be consistent with the policies of each domain, and a number of policies of different domains may conflict. Therefore, negotiation is a **decentralized process of policy resolution and conflict management**, where none of the domains have complete knowledge of the others' policies, state and goals. It is a **best effort** procedure, or theoretically suboptimal compared to centralized policy resolution using an oracle with full knowledge. We compare centralized policy resolution (a

non-trivial operation) with negotiation in Chapter 8.

- In the most general case, each participant's **local policies are private** and unknown to the other. Keeping policy private is not a random assumption; exposing policies, especially access control rules, could open up a system to abuse and have serious security implications. A malicious agent may potentially take advantage of such knowledge to compromise systems; this has been demonstrated in widely used network security protocols. For example, the knowledge that resource R can be accessed only between 8 pm and 9 pm, and only by X, could invite denial-of-service attacks targeted both at the resource host and at X.
- Unlike many other interoperation protocols, ours is **bidirectional**, without any negotiator constrained to be a client or a server, since both entities may possess objects that the other desires. The model in Section 3.1 makes no distinction between any of the interacting domains. For example, a patron's PDA and a coffee shop network could derive mutual benefit from interaction; the former obtains network access, while the latter could expand its customer base through incentives that include network access. Our protocol enforces an ordering: only one domain may initiate a negotiation, and each party must send a message in turn. These constraints are necessary to avoid race conditions and other synchronization problems.
- Negotiation is **dynamic**. For a negotiator, each step results in the discovery of the other negotiators' characteristics, services and policy constraints. Intermediate goal satisfactions (through requests and responses) result in change of state and policies of the domains. The original goals are therefore continuously re-evaluated. Such re-

evaluation is guided not just by local policies, but may also involve meta-issues like the time taken and the level of trust gained in the other negotiators. A negotiation does not follow a script (in other words, a template that specifies a limited set of defined states and actions that is known to all negotiators); rather, the steps vary depending on the nature of the negotiators' policies.

- The negotiation protocol is **not domain-specific**, but is designed to be independent of the nature of the resources and policies. Two domains can negotiate as long as they share a common semantic framework for the description of their resources and policies. An infrastructure like the Semantic Web and an expressive policy language are sufficient for such a negotiation scheme to be applicable ubiquitously. For example, negotiation is going to be difficult, if not impossible, between entities that have a totally different idea of what the resource “*display*” signifies. (Note: this does not preclude security domains from using diverse strategies and heuristics, which need not be understandable or familiar to other domains.)

**Protocol Engineering:** To negotiate, two domains must know about (or discover) each other's presence and establish a low-level data communication channel [Eustice2008b]. Different interaction scenarios use different communication channels. Ubicomp interactions typically involve wireless communication between mobile devices and network access points. Software agents on the Web interact through the application-layer HTTP protocol, and are not device-based. Policy-guided negotiation protocols are applicable to these seemingly diverse sets of negotiators and communication media, as

they all involve private data and policies. Related research in web interactions includes the P3P standard [P3P] and automated trust negotiation [Winslett2003].

### **3.3. Negotiation Theories and Strategies**

Certain dynamics are common to all negotiation instances, namely that they involve proposals and counter-proposals. Proposal message contents are determined by the input (primarily goals and policies). A number of other extraneous factors determine the priorities of the negotiators, guide the actual process, and consequently determine the outcome; we will discuss these at the end of this section. But even though these factors vary with the target application, the negotiation protocols have the following in common:

- They support arbitrary starting points and inputs (equivalent to our resource and goal sets) and generate outcomes that have unanimous agreement, i.e., the outcomes are acceptable to all negotiators based on their constraints (policies in our framework) but may not necessarily yield the *best* result for all.
- They have the notion of a strategy, which can be loosely defined as an algorithm or a function that determines the path a negotiation instance follows, given that there could be many different paths leading to different outcomes.
- The participants cannot coerce one another to bend their rules and do something against their will; every participant is completely autonomous and can withdraw from a negotiation whenever it chooses to.

Our negotiation protocol, as we will see in Chapter 5, has all the above characteristics.

Now we briefly survey the dimensions along which negotiations could be categorized, and which affect the choice of negotiation protocol for a given application. Then we discuss various factors that affect negotiation strategies, and consequently, the outcomes.

### **3.3.1 Characterization and Analysis of Negotiations**

Negotiations can be categorized based on the following criteria:

- *Types of expected outcomes and propensity to compromise:* Broadly, one could classify negotiations as being *rigid* or *flexible*. In a rigid negotiation, the purpose is simply to determine whether goal satisfaction is possible through the process of proposals and counter-proposals; the participants are not willing to back down from their maximal demands, resulting in a largely binary (yes/no) agreement. A flexible negotiation has participants willing to settle for a degraded level of goal satisfaction (the logic being that something is better than nothing) based on the others' willingness to compromise as well. A scenario involving sensitive resources and private information, the compromise of which could be disastrous, would be an example of a rigid negotiation. A bargaining process, where the participants' resources are known but their compromise levels are not, would be flexible, resulting in a compromise agreement.
- *Purpose of negotiation:* Some negotiations are done purely for the purpose of discovering and matching the appropriate resource to a prospective consumer. The process results in a semi-free information exchange among participants, resulting in



everyone having collectively more information in order to make the right decision. DHCP and Jini service discovery protocols fall under this category. Negotiations for resources such as disk space on the Grid, or network bandwidth for better QoS, can be considered as distributed resource allocation protocols. Both of these classes of negotiations have minimal or no associated security risk, and take the form of negotiation simply because the necessary information to solve the overall problem is distributed among the participants. Distinguishable from this category are negotiations among service owners and consumers, where security and validation play an important role in reaching an agreement. The KeyNote trust management system [Blaze1999] is an example.

- *Relationships among the negotiators*: Negotiators may not have identical powers and roles. Broadly, one could classify negotiations as being of the client-server type or the peer-to-peer type. In the former case, services are owned by one of the negotiators and the others are forced to compromise by virtue of their supplicant status. The service provider could terminate negotiations without losing anything. In the latter case, participants have an equal interest (or disincentive) in compromising and guiding the process towards a mutually beneficial conclusion.
- *Negotiation process*: Drawing from game theory, one could broadly classify negotiations as being either cooperative or competitive. In a cooperative negotiation, participants are more willing to share information with each other and help each other, as such cooperation is likely to lead to better results for all. A competitive model is used either when the payoff is zero-sum or each participant feels that it

stands to gain more by denying certain information to the other side.

These categories are not mutually exclusive, and most scenarios would be drawn from combinations of categories of each criterion. Our negotiation model is not restricted to a specific category, as it could be used for different purposes, could be competitive or cooperative, and accommodates negotiators with different levels of influence. The flexibility of the protocol could be varied by framing suitable policies that indicate how strict a domain must be in satisfying others' resource requests, how sensitive it is to its privacy, where its interests and priorities lie, and its trust and influence over others. In practice, we concentrate on scenarios that involve secure guarded resources and private information because that is the primary motivating factor driving our research, and one that will be most beneficial to the advance of ubicomp.

There are various perspectives through which we can view and analyze negotiations. The purpose of such analysis is to model negotiations with more specificity and predict the outcomes. Briefly, the modes of analysis prominent in literature are the following [Zartman1988]:

- *Structural Analysis*: In this mode, elements of power or leverage are distributed among negotiators, resulting in a power structure or power relationship among them. This power differential can be used to predict the outcome, though it may not be straightforward in all cases. Power typically represents the ability of a negotiator to enforce contracts, and to make its choices prevail.
- *Strategic Analysis*: Strategies are forms of heuristics that determine what move to make in a given configuration; this concept is drawn from game theory. Utilities or

payoffs are assigned to possible outcomes, and the estimations of the payoffs of each available move typically guide the strategy. Cooperative strategies are used to reach *equilibria*, or outcomes that are mutually beneficial, though these are hard to enforce in practice because of lack of trust and because moves are made nearly simultaneously without full knowledge of the game state.

- *Process Analysis*: Negotiation is considered a form of haggling, where participants start from fixed points and make concessions, eventually converging to a common point, or backing out if the cost of concession becomes prohibitive. In this mode of analysis, the dynamics of processes, or the behavior of parties in finding the rate of concession, are studied.
- *Integrative Analysis*: Negotiations are divided into multiple stages, which consist of pre-negotiations, where parties make first contact, finding a formula for a positive-sum outcome, and the settlement.

These analyses are not exclusive to computing systems, but describe real-world negotiations among humans and groups, especially the structural and integrative modes. For the same reason, these modes are somewhat difficult to translate to a computing or mathematical model. *We prefer and use the strategic mode of analysis, as it maps well onto our negotiation model.*

### **3.3.2 Factors that Affect Negotiation Strategies**

For a given input, a unique outcome may not exist. Instead, multiple outcomes may be possible, depending on the strategies followed by the negotiators. It is generally (though

not always) possible to enumerate all possible outcomes and also generate a partial ordering of the net benefit of the outcome for each negotiator. A negotiator achieves an outcome of maximum benefit when all of its goals are satisfied and it does not have to make any concessions. In practice, one can only estimate the best outcome at every negotiation step and make an appropriate move. Below we list some factors and priorities (besides negotiation goals and policies) that affect the negotiation process and the determination of expected outcome:

- *Gain*: A negotiator wants to maximize his goal satisfaction irrespective of other factors, even if he incurs significant loss in other areas.
- *Loss*: A negotiator wants to minimize his loss (things he has to give up in order to get something), or minimize the compromises he is willing to make.
- *Net Utility (Gain-Loss)*: This is the difference between the expected gain and loss. This is, in general, a more useful metric than pure gain or loss, and is used in most bargaining situations. Utility is a well-studied concept in microeconomics.
- *Security*: Most negotiations involving sensitive resources and data take into account the potential security violations resulting from granting a request during negotiation. The objective is to minimize the probability of security compromise. (The only reason why a negotiator would be willing to consider having its security compromised is because his net benefit from having some of his goals satisfied overrides the security costs).
- *Privacy*: Negotiators may also be mindful of the potential privacy loss they incur by revealing, for example, identity-related data, or their policies. A negotiator may be

- willing to compromise on its goals in order to safeguard its privacy.
- *Time*: The total time taken to complete a negotiation is important in many scenarios that have real-time constraints. A negotiator could offer certain compromises during the process in order to reach a quicker result.
  - *Number of Steps*: The number of steps is related to the time a negotiation takes, but it also incorporates other factors, such as the cost of sending, receiving and processing messages, and a network bandwidth cost. Sometimes, negotiators may not have real-time constraints, but might want to terminate a negotiation in as few steps as possible to save networking resources.

Negotiation strategies commonly consider combinations of the above factors. The factors are also not isolated from one another. Probability of privacy loss or security compromise, or a long negotiation could be modeled as estimated losses. Utility models could be used to estimate gains, and heuristic functions could be designed that would estimate a net benefit of doing a negotiation step, where multiple action choices are available. In game theory parlance, these would be equivalent to payoff functions. These heuristics or payoff functions can also be framed as meta-policies in the same language as input policies to a negotiation. In our protocol implementation and demonstrative applications, we focused on system-building and performance comparisons. We did not experiment with different strategies and heuristics based on the above criteria, but we provide guidelines for these in our future works chapter (see Chapter 11).

## Chapter 4

### Policy Language and Database

As summarized in the introduction, policies are sets of rules that describe the behavioral constraints and ideals of a ubiquitous computing domain. We take an expansive view of what a *policy* is in this context. System state, knowledge, beliefs, invariants, and intentions, in addition to behavior specification rules, are included in our definition of a policy. Also, policies cannot be treated in isolation as they impact each other, and are impacted by changes in domain state. Therefore, we provide both a language for writing individual policies and also establish clear rules that govern the management of a database of policy statements. Where do these policies come from? System administrators, with knowledge of the resources, devices and applications that comprise a domain, will write a large number of policies that are unlikely to change frequently. These include both behavioral rules and targets, but also the more static limitations and constraints inherent to the domain components. Events and dynamic changes in context during regular operation, and the resultant changes in the state of the components, would be registered as policies in an automated manner, as programmed by the designers and system administrators. A large number of policies, and goals or requirements, will be specified by the users themselves.

Our interest in conceiving and building a policy language was limited to the purpose it would serve in our policy management and negotiation framework. We start

off by listing both the challenges inherent in designing a policy language for a ubiquitous system and the key principles underlying such a design. We survey existing policy languages ranging from those designed for system administration purposes to those that were explicitly designed for use in dynamic ubiquitous computing applications and for Semantic Web applications. Following that, we describe the characteristics and limitations of our policy language and database. We conclude with a description of the design and implementation.

#### **4.1. Design Requirements and Challenges**

Our language and database management procedures must not only allow the expression of the objects and constraints described in Chapter 2 but also enable the goals outlined in Section 1.6. The syntax and semantics of our policy language must enable the following:

- Information can be extracted from policies in an unambiguous way so that a negotiation can proceed in an automated manner towards a conclusion. It should be possible to infer what is permissible and what is not through logical or mathematical means by examining those policies.
- It should be possible to pose queries and extract answers pertaining to current system state or goals by examining policies.
- One should be able to tell what action ought to be performed in a given situation (and consequently, when a situation changes owing to an event) to maintain inviolable rules and invariants.

**Nature of Policies:** Policies can assume different levels of stringency depending on the applications they are used in. They could be treated as mere guidelines, or default constraints that could be overridden as the situation warrants it. Systems that allow deviations from specified policies can be incorporated into a formal logic model, such as default logic [Antoniou1999]. When policies are meant to be examined and updated in real-time, they must be specified in a form that is computationally tractable. Therefore, we chose to state our policies as being statements with hard truth constraints; i.e., each statement must unambiguously hold a true/false value at every instant, and if it can be evaluated to be false, action must be taken immediately to rectify the situation and bring back consistency.

There are different views of what policies could be. They could be intentions or actions that are performed upon some event (e.g., the light must be turned off at 8 pm); goals or requirements that may have no immediate relevance but must be achieved in a suitable context (e.g., running a peer-to-peer application requires obtaining ‘x’ amount of bandwidth); system configuration, state or invariants that must always be maintained (e.g., no more than five occupants may remain in the lab at any instant); result or constraint policies that when queried, return a response (e.g., an access control policy that specifies whether or not Bob can access resource R). Our challenge is to have a language that provides a unified syntax and semantics to represent each of these forms, because a policy management framework of our scope should support these. These different views of policies are tied together through events, such as devices demanding admission into a domain. Such an event could trigger querying of a policy database to infer whether it is



permissible, examination of invariants to determine whether such admission would result in an inconsistency, and the performance of certain actions if admission is granted (which results in a change of state).

We discussed policy expressivity in Chapter 2, and gave an informal classification that was relevant to our application targets. One of the requirements of our policy language is that the syntax and semantics should not inherently limit expression of policy constraints when presented with a new scenario. We were less interested in providing ready-to-use policy constructs for a particular class of applications than a more general framework that would require some extra syntactic definitions for each application but no re-engineering of the semantics. The following is a list of requirements that our policy language needed to fulfill. This list is inspired to some extent by research in the area of policy languages for automated trust negotiation [Seamons2002].

1. *Well-defined logical semantics*: Statements in the language have truth values, and a policy management framework should be able to extract unambiguous meaning from a collection of policies. Therefore, our policy language should be backed by a formal logical reasoning framework.
  - a. This makes the policy language usable in domains with diverse resource capabilities and constraints, without being tied closely to any one domain in particular [Kagal2003a].
  - b. Decision-making on the basis of policies described in a logic-based language ensures correct and consistent behavior.

A logic-based language also enables formal analysis of the negotiation protocol that makes use of it, and one could prove properties like soundness, completeness and optimality using that logic.

2. *Declarative semantics*: As we discussed in Chapter 2, the language should allow easy specification of intent and goals, leaving the enforcement procedure to the runtime system. Thus, while the vocabulary describing objects could be specific to a domain, the semantics of dealing with the specifications will be common across domains.
3. *Individuals and groups*: The language must allow expression of attributes and constraints on individual elements, as well as classes (or groups) of individuals.
4. *Generality and specificity*: The language must allow expression of inter-policy constraints so that a general policy can be synthesized to a more specific policy in a given context.
5. *High- and low-level policies*: The language must allow specification of high-level policies that express user preferences in the same language as low-level policies that express system invariants, and the relationships between them.
6. *Local and global constructs*: The language must allow the writing of rules pertinent to local objects and constraints in the same way as global objects and constraints, and have the ability to separate the two for the purpose of external interactions.
7. *External functions*: The language must have the ability to specify external or helper functions for interfacing, mainly with the operating system.
8. *Causality*: The language must have the ability to express causality or consequences.
9. *Exceptions*: The language must have the ability to specify or add on exception

conditions to a general rule.

10. *Monotonicity*: The language must allow a negotiation to proceed in a monotonic, non-contradictory way. That is, promises made during negotiation should not be contradicted at a later stage owing to defects in language syntax or semantics.
11. *Constraint extraction*: The language must enable extraction of unsatisfied constraints in order to drive forward a negotiation.

Designing a multi-layer policy language for different policy classes, such as high- and low-level policies, or local and global policies, would make design and implementation too cumbersome. Since the different policies must coexist in a database and be consistent with each other, it is advantageous to have a unified policy language; there could be some syntactic separation between the different policy classes but the semantics would be identical, which would serve our purposes.

Being compatible with legacy frameworks of policy managements or with domains that use other policy languages was not a priority for us. Still, if a legacy system used a policy language based on the same logical semantics as our language, interoperation would be possible, and relatively minimal work would be involved in gluing together the constructs of the local and global policy languages.

## **4.2. Key Language Characteristics**

As mentioned above, we wanted a policy language that would be independent of the domain or applications it was used in, and still be usable by the negotiation framework in diverse scenarios. The constructs or syntactic features provided by the language were

important only to the extent that scenarios could be constructed. To minimize our initial effort, we chose to adapt a policy language from an existing language that would provide a syntactic base and basic reasoning semantics, retaining the option of modifying or augmenting it later. (*Note:* We did not need to exercise this option while implementing the policy manager and our demonstrative applications.) Though a number of desirable features of a policy language can be found in existing languages (more in Section 4.5), the combination of features needed for a negotiation framework was not available in any existing policy language.

Our policy language is built on Prolog, which is based on first-order logic. Recent implementations [Roy1990] have significantly advanced its computational efficiency, making Prolog satisfactory for use in a real-world framework, especially as mobile users are unlikely to perceive appreciable response time lag (see Section 4.4). The structure of predicates, variables and other terms in Prolog allows us to specify categories and instances of entities, objects and contextual parameters in policy rules. The semantic nature of a logic-based policy language also enables specification of high- and low-level policies, and the specification of relations between these. We use the SWI-Prolog code base and API [SWIProlog], which offer important features that will be discussed later. System state and policy rules are defined in the form of Prolog facts and rules (clauses).

Using Prolog as our language base, the first two requirements in Section 4.1 are satisfied. First-order logic has well-defined and sound semantics, and those properties transfer to Prolog, as Prolog is based on a more restrictive logic, and our language restricts it even further (as we will see later on in this section.) Policy statements in

Prolog are also declared in the form of intents and goals, leaving the enforcement procedure to the runtime system. Below, we describe the salient features of our language, and show how they satisfy the requirements from Section 4.1.

**Ontology:** In Chapter 2, we discussed the fact that most system information must be understood only within a domain, and need not be part of a global specification language. Interoperating devices still must understand and interpret the objects that they trade. Such specification issues have been researched in the context of the Semantic Web and other open frameworks, examples being RDF/XML (which have been widely adopted), DAML+OIL [DAML] and OWL [OWL]. Our policy ontology is inspired by SOUPA [Chen2004], which defines a set of *core* components and optional *extensions* that can be used to model ubiquitous computing applications. Our ontology is described in Figure 4, and to reiterate, includes the following: *entities* and *agents*, *resources* and *content*, *properties* and *metadata*, *mechanisms* (e.g., *sensory*, *networking*, *cryptography*), *context*, *relationships* between entities and resources, *quantitative limits*, *precedence rules*, *deontic constraints* (e.g., *permission*, *obligation*), *actions* and *events*. Our interest in ontology here was limited to making sure that our language would have the capability to describe the various elements in Figure 4, and not be inherently limited in some respect. Our motivation was similar to that of the designers of XML in this respect. Therefore, we do not describe each and every possible keyword that represents each element here, but rather give examples indicating that it would be possible in a given domain.

**Syntax and Semantics:** Our language uses almost all syntactic features of Prolog. Each statement in the language is either a *fact* or a *rule*. Each fact (or term) has the form “predicateName(argument<sub>1</sub>, argument<sub>2</sub>, ..... argument<sub>n</sub>)” where  $n \geq 0$ . Each argument is either a constant (includes random strings, arithmetic numerals, lists, Boolean values, or functions) or a variable (a string beginning with a capital letter). Each rule has the form “term<sub>1</sub> :- term<sub>11</sub>, term<sub>12</sub>, ..... term<sub>1n</sub>” where  $n \geq 1$ . The *head* is the term on the left of the ‘:-’ operator and the *body* consists of the terms on the right.

A rule is a Horn clause, or a conjunction of terms implying another term (the ‘:-’ operator denotes reverse implication). The comma implicitly denotes the conjunction, or AND, operator ( $\wedge$ ). In addition, the terms could contain the disjunction operator ( $\vee$ ) denoted by the semicolon (;), the implication operator ( $\rightarrow$ ), and the cut (!) operator. Negatives are denoted by a ‘not’ operator, which takes the form of a predicate name.

The semantics of these operators are identical to those in first-order logic. Each term has a binary truth value. In the rule written above, term<sub>1</sub> evaluates to TRUE if and only if each element in the set {term<sub>11</sub>, term<sub>12</sub>, ..... term<sub>1n</sub>} evaluates to TRUE. A term in the body could be a head of another rule. Given a set of Prolog statements, one can pose queries, which are answered by a process of search for all possible proofs of the query statement. Evaluation takes the form of a depth-first search of a tree of rules, with facts being the leaves or terminal points. *Backward chaining*, a well-known first-order logic algorithm, is the evaluation technique followed in Prolog derivations. This procedure uses *unification*, or the matching of two terms to return a valid variable-to-constant binding, to process queries. Our collection of policy statements is thus a *logic*

*program* that consists of certain provable facts, and the semantics of the language are the semantics of query processing.

Two features of Prolog need to be noted by policy writers; these features result in deviations from first-order logic, though a large number of Prolog statements can be written in such a way that they are faithful to the logic. First, evaluation of the conjuncts in a rule body is strictly left to right, which implies that the way one writes policies impacts query processing. As we will see later, this has an impact both on the completeness property of negotiation as well as its performance. The other feature is the omission of the *occur-check* [Marriott1989] from the Prolog unification procedure. This operation comes into play if one of the arguments of a term is a function whose arguments might contain a variable that the function itself is being matched against. This is a heavy operation and is commonly omitted for performance reasons, but the result is that the Prolog inference procedure is unsound.

In our language, as in Prolog, negatives cannot be asserted directly. The way to prove negatives is through *negation-by-failure*; i.e., the absence of a proof of an assertion is considered to be a proof of its negation.

Due to soundness issues, we generally prohibit functions in the statements in our language. We don't lose any expressive power this way, since a function can be specified in the form of a relation (a predicate having a truth value). For example: the function `fileType('song.mp3')=audio` and the relation `fileType('song.mp3',audio)` (which can be asserted in Prolog) are equivalent. We allow functions only in a couple of places

and ensure that the functions are never matched with variables they contain, thereby preserving soundness (we will see examples of this later).

A depth-first search procedure is incomplete, and may not terminate when cycles (directly or transitively self-referential rules) exist in a set of policies. To avoid this completeness problem, we prohibit cycles in our language. This does not lead to a loss of expressivity, since a set of statements with a cycle can be transformed into an equivalent set of statements without a cycle [Han1991]. In practice, especially when naïve users are allowed to write policies, it may not be possible to avoid cycles. But our inference procedure could be augmented with additional checks that prevent a rule from being examined more than once for a given query. We don't currently do this because the space and time complexity of such checks are prohibitive.

Some examples of policies in our language are given below, with explanations in English. 1-3 indicate facts, whereas 4-5 indicate rules or *if-then* clauses.

1) `fileType('song.mp3', audio).`

**['song.mp3' is an audio file]**

2) `relation(alice, bob).`

**['alice' and 'bob' are relations]**

3) `certificate('UCLA').possess(john, 'UCLA').`

**['UCLA' is a certificate, and is possessed by 'john']**

4) `member(X) :- teamMember(X), numChildren(N), maxChildren(M), N < M.`

**[X is a member if it is a team member and the current number of children is less than the maximum]**



5) `access(S,V) :- candidate(S), teamMember(S), voucher(location,V).`

**[S can be granted access to voucher V if S is a ‘candidate’ and a team member,  
and if V is a ‘location’ voucher]**

Variables allow the expression of groups or classes. Every statement containing a variable has the implicit for-all qualifier ( $\forall$  *variable\_name*) attached to it. For example, statement 4 above defines a class of ‘members’, each ‘member’ being denoted by the variable ‘X’. Thus our language satisfies requirement 3 from Section 4.1.

There are a number of designated predicate names which we use in our policy language to denote special kinds of objects, actions or states that are necessary for negotiation. Negotiation requires the expression of speech acts that can be translated into messages of give-and-take and communicated to the opposite negotiator during the course of a protocol session. We will describe speech acts and messages in detail in Chapter 5, but briefly give examples of the predicates that express these below.

- Possessions of resources and data are denoted by `possess(arg0)` or `possess(arg0, arg1)`. Correspondingly, access to a resource is denoted by `access(arg0, arg1)`.
- Actions that can be performed are denoted by `action(arg0, ..... argk)`, with a variable number of arguments. Correspondingly, rules governing agreement to an action request by another entity are denoted by the predicate `obey(arg0, arg1)`.
- A number of scenarios might involve simple information transfer. For example, a domain could have a predicate `location(loc)` asserted, indicating that its current location is ‘loc’. Predicates that govern the release of such information are denoted by `accessInfo(X, Pred)`; e.g., `accessInfo(X, location(loc))`. The predicate

`'location(loc)'` is used as a function here. Other cases where functions are permitted are described in Chapter 7 and in Appendix I.

Certain keywords, namely constants and predicate names, are asserted as global through the `global` predicate. Predicates that denote 'request types' (describing elements that can be requested by one negotiator to another) are asserted using the `requestable` predicate. Not all predicates can be `global` or `requestable`, thereby providing a simple way of satisfying the local vs. global requirement in Section 4.1.

The following examples illustrate low-level rules: in (1), a low-level JPL (see Section 4.4) query (`jpl_call`) is required to call a Java method to translate a group member name to a player name; and in (2), the action of closing a port 'Po' requires the execution of the 'iptables' shell command.

```
1) teamMember(X) :- groupMember(X), playerName(Y),  
    jpl_call('panoply.policy.Helper','sphereName',[X],Y).  
2) action(closePort,Po) :- atom_concat('iptables -A INPUT -j DROP -p  
    tcp --dport ',Po,C1), atom_concat(C1,' -i lo',C), shell(C,0).
```

These rules describe static configurations of a domain, and are usually set by people with knowledge of the system. Such rules also indicate how external functions (calling a Java method in rule 1, and running an operating system command in rule 2) can be invoked; thereby we satisfy both requirements 5 and 7 from Section 4.1.

Causality can be specified in our language using implications in two ways. The clause construct by itself denotes causality, whereby the head of a clause happens to be

the consequence of the terms in the body. We also use the implication operator ( $\rightarrow$ ) to specify causality constraints.

In Chapter 5, we will show how the monotonicity and constraint extraction properties are satisfied.

**Inter-Domain Translation:** All domains do not have the same ontological range, and this range will be fairly limited for most domains. For instance, expressing concepts such as a display device is relevant only to domains that either possess or would require the use of such a device. A vast number of credentials and pieces of data would be useful only for selected domains and entities. Therefore, building a comprehensive database of syntactic elements, or keywords for predicates that represent all of these elements, is unnecessary and practically difficult to enforce as a standard. Therefore, we take a similar approach to this problem as that adopted in Semantic Web technologies, namely XML and RDF. Using standard (and extremely minimal) ways of describing XML schemas through Document Type Definitions (DTD), an XML document could be parsed by entities other than that which created it. Thereby, one can specify different keywords in different domains, but the core structure and semantics of the language are platform independent. In our framework, when different domains interact, or when diverse applications, resources, or groups are brought under the same policy domain, one could easily add a small number of policy statements that would aid in cross-translation. For example, if a display device is represented using the predicate `disp(D)` in domain  $D_1$ ,

and `display(D)` in domain  $D_2$ , the following Horn clauses could be added to the two domains for a common understanding of what a display device would be:

$$D_1:\text{display}(D) \text{ :- disp}(D).$$
$$D_2:\text{disp}(D) \text{ :- display}(D).$$

If a legacy domain were made part of a bigger domain (e.g., a network is expanded to include more computers), one could make their policy databases compatible by adding suitably written translation policies like these. Such compatibility can only be ensured manually through out-of-band channels. If two domains with different keywords representing the same objects were to attempt a negotiation, it would likely fail.

**Limitations of the Policy Language:** As mentioned before, Prolog, and consequently our language, is a restriction of first-order logic, whereby we can talk about functions and relationships between predicates, constants and variables, and use existential and universal quantifiers. What we cannot express are policies about policies. This would be a form of second-order logic, whose reasoning semantics are not so clearly understood as those of first-order logic. We therefore deal with policies in a non-logical manner, through manipulation by the external policy management system, rather than through the reasoning engine. This means that the expression of precedence or priority is somewhat limited. One could express something like the following: a color display is preferred to a black-and-white display for application A; but one could not express the following: policy rule  $R_1$  is preferable to policy rule  $R_2$  in context C.

Another area that we have not investigated rigorously is the writing of temporal policies. There is a large branch of temporal logic that deals with constraints involving time, events in the past and future, and so on. We did not feel the need to actively pursue research in this area. There is no inherent limitation in our policy language that prevents the expression of temporal constraints; rather, when we deal with events in the future, using timers and alarms, our policy engine may incur high overhead, or stored state information may increase fast over a period of time. This occurs because of the continuous nature of time, which can be divided into arbitrarily fine granularities. Consider an example involving a policy that governs when an automatic door can be opened or closed. We could specify a constraint that the door must not be open longer than 5 seconds. This would entail writing a policy that schedules a door closing event within 5 seconds of the door opening event. But what happens when someone else opens the door within that 5 second window? If we were to retain the originally scheduled event, the door might close too early for intermediate arrivals. So our policy engine would have to examine the entire database and invalidate the scheduled close events. Since time can be divided into arbitrarily fine granularities, the number of such events (however small the time window) are potentially unlimited. There is no straightforward logical way of handling such situations. We could use an external scheduler that would be invoked as needed but would not be part of the policy engine. We avoided this issue while designing our policy language because it impacted our research tangentially at best. The language can express policies relevant to the door closing example, but does not go

far enough to resolve the state or the event scheduling problem. The language does allow expression of event triggering actions, which we will see in the following section.

### 4.3. Database Semantics and Operations

Every domain maintains a policy database, consisting of the facts and rules written in our language. Such a database has collective logical properties, whereby each statement cannot be treated in isolation. One of the requirements stated earlier was that a policy management framework must return an unambiguous answer to a posed query. This implies that all database statements must be consistent with each other, i.e., if the policy database =  $\{statement_1, statement_2, \dots, statement_n\}$ , then  $statement_1 \wedge statement_2 \wedge \dots \wedge statement_n$  must not be a logical contradiction.

A policy database supports various operations that can be roughly classified into examination functions and modification functions. Examination functions include database querying, which does not result in the change of any statement in the database. Modification functions are intended to cause a change in the database contents, and these must be performed in such a way that the above consistency property is maintained. These consistency issues have been studied in logic-based AI systems as *truth maintenance* [Doyle1979]. We list the operations supported by a policy engine database manager, and provide specifications, examples and intended uses in a real-world domain.

### 4.3.1 Examination Operations

Each function below uses the default database (having been initialized prior to operation) unless specified otherwise.

***QueryResult:***

*Input:* Query string in Prolog; Database file containing Prolog statements

*Output:* Solution as a set of variable bindings

At its core, this method runs a Prolog query on the input string. If a null database argument is passed, the default database is examined; otherwise the passed database file is examined. The Prolog query procedure uses backward chaining and unification.

***Substitute:***

*Input:* Prolog term (string), Variable binding (*map*: variable\_name  $\rightarrow$  constant)

*Output:* String with variable occurrences from binding substituted with the corresponding constants in the given term

***IdenticalTerms:***

*Input:* Two Prolog strings, Variable mapping

*Output:* Success indicated by returning a non-null variable mapping; failure indicated by returning a null mapping

This operation can be performed using standard unification. It is not a trivial string matching operation or a unification because it should succeed only if the constants, including predicate names, of both strings match, and variables at corresponding positions match each other's modulo names. For example: `pred(A,B,t)` should match `pred(C,D,t)` but not match `pred(C,C,t)`.

***DominatingTerms:***

*Input:* Two Prolog strings, Variable mapping

*Output:* Success indicated by returning a non-null variable mapping; failure indicated by returning a null mapping

This operation can be performed using standard unification. Term  $T_1$  dominates  $T_2$  if  $T_1$  is more *general* than  $T_2$ . This can happen in two ways: (i) if  $T_1$  and  $T_2$  are conjunctions, and the conjuncts in  $T_1$  are a subset of the conjuncts in  $T_2$ ; (ii) if both  $T_1$  and  $T_2$  are single predicates that can be unified, and if all constants in the resulting variable binding occur in  $T_2$ , then  $T_1$  dominates  $T_2$ . For example: `pred(A,B,t)` dominates “`pred(A,B,t), pred1(K)`” because the former is less restrictive; `pred(C,D,t)` dominates `pred(C,d,t)` because the former is less restrictive, the resulting variable binding being  $\{D=d\}$ , where the variable occurs only in  $T_1$ .

By default, if  $T_1$  is identical to  $T_2$ , it is said to dominate  $T_2$ .

***PresentStatement:***

*Input:* Prolog statement (fact/rule)

*Output:* Success if the input is either directly or indirectly asserted in the database

A given statement can be asserted, or present, in two ways: i) it is directly asserted as a fact or rule and its presence can be inferred just by examining the database; or ii) a statement is asserted that dominates (is more general than) the input statement. A fact  $T_1$  is more general than a fact  $T_2$  if the function *dominatingTerms* succeeds. A rule  $T_1$  :-  $T_{11}$  dominates a rule  $T_2$  :-  $T_{21}$  if  $T_1$  dominates  $T_2$  and  $T_{11}$  dominates  $T_{21}$ .



### 4.3.2 Modification Operations

#### ***AddStatement:***

*Input:* Prolog statement (fact/rule)

*Output:* Returns success (if addition was successful) or failure

*Task:* First, we check to see if the given statement is asserted either directly or indirectly within the database using the *presentStatement* function. If it is, the function returns with success. Otherwise, the statement is added to the database.

Conceptually, this function should not return a failure; if the input (say  $S_1$ ) is contradictory to a statement (say  $S_2$ ) already present in the database (i.e., has the opposite effect),  $S_1$ 's addition simply complements  $S_2$ . Since Prolog reasons through negation-by-failure, it perceives no contradiction. In practice, *addStatement* returns a failure only if the input is invalid (e.g., a SWI-Prolog meta-predicate that should not be modified).

#### ***RemoveStatement***

*Input:* Prolog statement (fact/rule)

*Output:* Returns success (if removal was successful) or failure

*Task:* First, we check to see if the given statement is asserted either directly or indirectly within the database using the *presentStatement* function. If it is not, the function returns with success. Otherwise, the statement is removed from the database. The effect of this removal should be the following: after the function returns, if *presentStatement* is called with this statement as input, it will return false.

### ***ChainEventAction:***

This function serves to accomplish a form of forward chaining, though not exactly the way it is classically defined.

We define a special category of *event-action-trigger* rules that respond to events. An example of such *update* rules is given below:

- `update :- (((numRelatives(X,N), door(X), closeD(X)), doorOpen(X),  
N>0) → (retract(doorOpen(X)), retract(closeD(X)))).`

The function `chainEventAction` is called whenever a change is made to the state of the database through the `addStatement` or the `removeStatement` functions. It in turn makes a Prolog query ‘update’, and all such update rules are evaluated as a result. If all conditions on the antecedent of the body of the clause are proved true, the consequents are also evaluated. In the above rule, ‘update’ results in a change to the state of a ubiquitous door service. For every known door (asserted as ‘`door(X)`’) that is open (asserted as ‘`doorOpen(X)`’), if a close event has been initiated (asserted as ‘`closeD(X)`’) and the number of occupants (asserted as ‘`numRelatives(X,N)`’) is greater than 0 (‘`N>0`’), the door’s open state is retracted from the database, using the statement ‘`retract(doorOpen(X))`’ (implicitly indicating that the door is closed.) The event-initiating predicate is also retracted, using the statement ‘`retract(closeD(X))`’. In a real scenario, if `x='door3564'`, an assertion of ‘`closeD(door3564)`’ may cause the antecedent to be true (if `N>0`), resulting in the consequent (the *retract* statements) being evaluated. This function results in appropriate state changes being made in response to policy-specified events.

#### **4.4. Policy Engine Implementation**

A *policy engine* is the back-end of any policy management framework that uses policies written in our language for the purpose of negotiation or state/action monitoring. It abstracts the policy database from the programmer as well as the user, and provides ways of accessing and manipulating the policies through suitable interfaces. Such an engine must implement and offer the functions listed in Section 4.3, which then can be called by the higher layers of a policy management framework (which can be implemented in a variety of ways for a variety of purposes). The declarative model of writing policies makes such a policy engine model (whose role is analogous to that of an interpreter for a regular programming language) most appropriate for use with our language.

Though we go into further details about the implementation of a policy management framework, we recommend here that irrespective of the framework, the policy engine should be designed and deployed as a static, or persistent, module, providing hooks or functions into the database for any domain. A single policy engine should be maintained per domain. It is not necessary that there be a single unified database managed by such an engine; there could be multiple such databases that are accessed in isolation for different purposes. This might be useful in situations like the following. A domain could choose to apply different, often contradictory, policies in different contexts. Obviously, combining the different policies into one database would introduce logical inconsistencies. A higher-level policy collection that simply consists of contextual assertions could then be invoked, first by the policy engine, and the relevant policy database invoked based on inferred context. The policy engine would require

additional internal functions to abstract away these multiple databases and present a unified view to the higher layers. Therefore, in addition to offering basic manipulation and query functions, a policy engine must support importing and reconciliation of whole databases into the existing repository.

To be able to perform the tasks listed in Section 4.3, the policy engine must be built using a Prolog subsystem that offers ways of manipulating individual logic statements. There is a large body of free software available for this purpose, and many such frameworks are open source, licensed under GPL. These include SWI-Prolog [SWIProlog], CiaoProlog [CiaoProlog], TuProlog [TuProlog], and Minerva [MinervaProlog]. We selected SWI-Prolog, which offers various policy examination and manipulation features in the same syntax (predicate and arguments) as regular ISO-Prolog. These include `assert`, `retract`, `clause`, `term_variables` and `functor`, just to name a few. In addition, arithmetic and string operations are also supported. The biggest advantage to using SWI-Prolog is that it offers bidirectional APIs to programming languages like C and Java (the Java library is called JPL), offers operating system and networking features, and supports Semantic Web technologies like RDF and XML. The implementation and query processing is based on the Warren abstract machine technique [Aït-Kaci1991], which makes performance reasonably efficient on modern computers. This ensures that the policy engine implementation does not have to worry about inefficient indexing and retrieval of predicates in, say, Java.

#### **4.5. Comparison with Existing Policy Languages and Frameworks**

Here we describe how our policy language and database management framework differs from prominent related frameworks that support some form of negotiation, ubiquitous applications or Semantic Web applications. (For a more comprehensive list, see Section 10.2.) We will see why adopting those frameworks will not work for our stated aims, and also to what extent our framework can support those applications.

A large part of our work was inspired by research in trust negotiation [Winslett2003], and Seamons et. al. have provided a comprehensive list of requirements and challenges in designing a policy language for trust negotiation [Seamons2002]. Our framework was intended to be more general than trust negotiation, but these requirements provided a start for our research. As in trust negotiation, our language has well-defined semantics, is monotonic, and provides support for external functions. The former requires some very specific support for credentials and operations on credentials, whereas our system deals with more than just credentials. It requires support for expressing credential attributes, dependencies and chains, expression of transitive trust, and support for local credential variables. As our ontology and language specification indicate, all of these things can be expressed in the language of Prolog. In addition, our language has the capability of describing actions, states, events and meta-constraints, none of which are required in trust negotiation. Our language does not directly support authentication of credentials and policy rules as core language constructs, but these could be handled through external functions added as constraints to relevant policies (policies governing credentials could have added validation constraints that would succeed only upon

execution of say, a Java method, that verifies digital signatures; see policy rule 1 from page 78, for example). Thus, our language, with some straightforward extensions, could facilitate trust negotiations. On the other hand, a trust negotiation language will not be general enough for our purposes. Seamons et. al. [Seamons2002] also show that languages like PSPL [Bonatti2000] and KeyNote [Blaze1999] do not satisfy all the requirements for trust negotiation, making them too restrictive for our purposes as well.

One of the goals of this research is to support a more flexible form of access control for open systems. Advanced access control frameworks like Generalized Role-Based Access Control (GRBAC) [Covington2000] and dynamic Role-Based Access Control (dRBAC) [Freudenthal2002] have been proposed for applications in ubiquitous environments. Our policy language and framework can be easily used to express GRBAC and dRBAC policies as well. The former defines roles for subjects or entities, objects or resources, and environments or contexts. Roles effectively classify subjects, objects and contexts. As we have shown, we can express classes and groups in our language through the use of constraints involving variables. Role names could be defined as predicates, and subject, object and contextual names can be specified as constants and variables. Dynamic RBAC enhances RBAC by allowing policies specifying varying levels of service access and delegation of permissions. We can specify varying both of these using our policy language. Through a combination of digital credentials called *vouchers* (which we will encounter in Chapters 6 and 7) and credential validation through external functions, our policy management framework can actualize the concept of delegation. Similar to the proof tree generation in an application of dRBAC, our framework also

enables dynamic proof generation involving multiple entities for the purpose of access control. Thus our policy language has the capability of supporting applications that are based on GRBAC and dRBAC.

Negotiations for resources on the grid take the form of proposals and counter-proposals consisting of resource levels that one side is willing to grant and the other side is willing to take. These are evaluated against resource utility functions, specified in terms of maximum and minimum permissible values. Our policy language can express resources, utility values of resources, and preferences between discrete levels. Therefore, we can support grid negotiation. We will discuss a related example in Chapter 7.

Policy languages for the Semantic Web, Rei being one example, must be domain-independent with logical semantics [Kagal2003a]. Rei is written in Prolog syntax and is based on deontic logic, a subset of first-order logic that deals with permissions and obligations. Our language, like Rei, supports descriptions of actions, action classes and speech acts (see next chapter). Though our language does not directly support specification of policy precedence, it does support inter-resource constraints, contextual policies, external functions, and a host of other things that Rei does not support. But our language could potentially be used for Semantic Web applications, many of which will deal with negotiation-like scenarios, as it is designed on the same basis as Rei. Ponder [Damianou2001] is another policy language used to specify distributed systems security policy, but it is an object-oriented language rather than a logical language.

WS-Agreement [Andrieux2007] and WS-Policy [WS\_Policy] are Web services standards intended to enable XML-based services to negotiate requirements. These

standards are purely XML-based, and by implication, very likely to acquire widespread adoption; but the policy rules that can be specified using these standards refer to particular services and applications that make use of those services. In a ubicomp domain, there will be dependencies among different resources, services and contexts. It is not directly apparent (in the absence of a logical framework underlying WS-Policy) how such dependencies can be efficiently declared and captured for the purposes of ubicomp negotiation. This is why we did not choose these standards as the basis for our research.



## **Chapter 5**

### **Negotiation Protocol**

In its simplest form, a protocol can be defined as the set of rules governing the syntax, semantics, and synchronization of communication. Our negotiation protocol uses the interaction model described in Chapter 3 as its basis. The discussion in this chapter restricts itself to a protocol between two negotiators, though multiple parties could perform pair-wise negotiations. We first discuss what the basic units of the protocol (the messages) consist of; following that, we describe the semantics at three different levels. At the highest level, we can describe the semantics through a fairly simple state machine that is run by both negotiators, and has high-level messages as its units. In an intermediate level of processing, message contents can be examined (based on message type) and a suitable output message with different contents generated; this processing can be described through a lower-level state machine. Finally, we describe the logical operations on the database offered by the policy engine back-end, which ultimately determine exactly what goes into the contents of each message.

#### **5.1. Protocol Units**

Negotiation messages, or units of the protocol, are defined and classified to allow the most general expression of a negotiation. Our model in Chapter 3 was based on logic, as was our policy language, the sentences being first-order propositions. But since we

assume that there is no omniscient entity with knowledge of both sets of policies, first-order logic is not enough to deal with negotiations. *Illocutionary logic* [Searle1981], a branch of linguistics describing the logic of *speech acts*, provides the additional glue that ties together first-order propositions and negotiation messages. Such *speech acts* are a suitable basis for a general-purpose and closed set of negotiation message types, as we determined with some inspiration from the Rei policy language [Kagal2003a]. The special predicates describing actions, possessions, release and agreement (see Chapter 4) provide the context that enables appropriate interpretation of their contents, which could vary arbitrarily. These contexts can be captured and classified formally using illocutionary logic. An entire subject in itself, we won't go into the details of this logic, but briefly describe how and why it enables a better understanding of negotiation and to what extent our protocol is based on its principles.

### **5.1.1 Speech Acts and Illocutionary Logic**

In their book “Foundations of Illocutionary Logic,” Searle and Vanderveken define a speech act as an utterance that expresses intent to perform an action. An illocutionary speech act consists of an illocutionary force and a propositional context. The force of an act describes the purpose (or point) of the act and the strength of the intention conveyed by the utterer. The propositional context of an act describes the nature and context of the intended action itself [Searle1981]. As it turns out, all acts can be constructed through operations on a small closed set of primitive speech acts, consisting of the following: *assertives* (‘assert’ or ‘state’, denegation of which is ‘deny’), *commissives* (‘commit’,

denegation of which is ‘refuse’), *directive* (‘direct’, denegation of which is ‘prohibit’), *declarative* (‘declare’), and *expressive* (e.g., ‘apologize’, ‘complain’, ‘protest’, ‘compliment’). All speech acts are variations of these basic ones, which vary based on the following properties: degree of strength of the act, mode of achievement (authority of the speaker), propositional content conditions, preparatory conditions, and the nature and degree of the speaker’s sincerity. The purpose of illocutionary logic is twofold: (i) to enable inference of consistency of speech acts and to derive commitments (one act entails the performance of another); and (ii) to describe conversations, or sequences of speech acts, where an act is constrained by the nature of its predecessor, and such constraints can be established through the rules of illocutionary commitment.

A negotiation is effectively an illocutionary conversation, and can be engineered based on what would make sense in a given state. That is, a message from A to B constrains the kinds of messages that B can subsequently send to A. Our negotiations are targeted towards transactions, or something that results in the transfer of information, objects, and change of state at the ends. Therefore, we can exclude expressive acts, since apologies, protests and compliments are irrelevant to computers. Our basic message types do cover the other four categories:

1. *Requests*: These encompass the directive illocutionary forces. Depending on the negotiation context, a request could have the force of a command, order, supplication, query, demand, etc. Examples include requests for access rights, permissions to perform actions, and queries for information.
2. *Policies*: These encompass both assertive and some directive illocutionary forces.

They include permission, prohibition, obligation, notification, and statement, among others. Examples are rules or obligation statements that a listener must comply with.

3. *Offers*: These encompass the commissive illocutionary forces, including commitment, refusal, promise, and acceptance, among others. These are replies to requests, with supporting objects, proofs, information; e.g., data files, certificates, etc.
4. *Termination*: This has the illocutionary force of a declaration, indicating that goals have been met, or nothing further can be achieved.

These message types cover the range of illocutionary points that are relevant to transactional negotiations. Therefore, we found them to be necessary and sufficient to support negotiation scenarios, as well as being domain-independent. Illocutionary logic can also be used to build compound (or complex) speech acts, and to establish rules for consistency and commitment. A negotiation is a sequence (or conversation) of speech acts; for our purposes, we needed to construct a protocol that ensured:

- *Success*: negotiator has requisite authority to make a request or propose an offer, and
- *Non-defectiveness*: conditions necessary for performance of the proposed act are met.

Our criterion for a successful and non-defective negotiation is that a result consistent with both sides' policies be achieved, as described in Chapter 3. Therefore, a request message containing a proposition *P* has the following illocutionary force and point: *I request that you perform P if you can do so in a way that is consistent with your policy*. A policy message has the following force and point: *I assert that my policy constraint is P and I would like you to comply with P if it is consistent with your policy*. Since policy dependence is implied, both these messages implicitly assert that the speaker

would like to receive an affirmative or a negative reply. An offer message has the following force and point: *I agree/refuse to 'perform P'/'comply with P', and I provide supporting evidence E*. A counter-request can follow a request, which has the following point: *I will comply with your request/policy if you agree to perform 'proposition' (and will not comply if you do not perform 'proposition')*. Every speech act is accompanied by conditions that implicitly express the sincerity of the speaker. Lack of sincerity results in a defective negotiation; e.g., a negotiator makes an offer with no intention of keeping the promise. It is then incumbent upon the hearer to verify (if at all possible) the speaker's sincerity. An example of this is: A agrees to apply firewall rules requested by B (using *iptables*); B can verify this by probing A (using *nmap*).

### 5.1.2 Classification of Negotiation Message Contents

The internals of messages can vary, and we classify the contents further based on the propositional contents of the speech acts and their modes of achievements, and action modes (affirmatives and negatives).

A request can refer to one of the following:

- **Action** <Do A>: This has the force of an order or a command. *Example*: I command you to run a piece of code, such as anti-virus software.
- **Action** <Permit me to do A>: This has the force of a supplication. *Example*: I would like to run a P2P application.
- **Possession** <Give me P>: This has the force of a demand or a supplication. *Example*: Give me access to a particular resource; show me your credentials (e.g., certificate)

- **State Change** *<Let me change to state S>*: *Example*: I would like permission to become a member of your network/domain.
- **Question** *<Tell me T>*: *Example*: Please tell me what your current location is; tell me what kinds of display devices you have, if any.

A policy refers to the following:

- **Obligation** *<Promise to abide by O>*: This has the force of a command. *Example*: You must not play any sound on your device during the presentation hours (say, from 3 to 5 pm).

An offer can be one of the following:

- **Agreement** *<Yes, I agree to do what you ask>*: *Example*: I agree to run the code you have provided me; I will give you the certificate you demanded.
- **Refusal** *<No, I will not do what you ask>*: These are essentially negations of the above.
- **Rejection** *<I cannot accept your offer>*: An “Agreement” offer must be accompanied by verifiable proof. If such proof is not offered, or the recipient is unable to verify the integrity of the offer, it will send a rejection in response to the sender. The latter implicitly understands that it must attempt to resend the offer.
- **Answer** *<Here is what you asked/inquired>*: *Example*: My current location is ‘3564 Boelter Hall’.

With offers indicating agreement, proof consisting of objects could be supplied. For example, a ticket indicating permission to access a resource could be provided, or an

encrypted piece of code could be provided. Validation could be done if the receiver has the relevant key(s).

### 5.1.3 Negotiation Message Structure and Methods

Every negotiation message is identified by its high-level type, and consists of a set of policy-derived statements. The fields of a message are as follows:

- Source (or sender) name: In our implementation, this is typically the name of a Panoply Sphere (to be explained in Chapter 6).
- Destination (or receiver) name: Same as above.
- Message Type: Set of values = {REQUEST, OFFER, POLICY, TERMINATE}.
- Message Subtype: Set of values = {ACCEPT, REJECT}. This field is used only with an OFFER message. In practice, we end up using only the REJECT subtype, since acceptance is implied by the absence of a REJECT message.
- Array of entries: Each entry indicates a particular speech act relevant to the message type. A request message contains entries referring to demands for objects (`possess(printer_HP)`) and commands to perform various actions (`action(closePort,25)`); likewise for policy messages. An offer message contains affirmatives, negatives, or answers to the posed requests. A termination message does not contain any entries.
- Array of unique IDs corresponding to the entries: Each entry has a unique ID. A request or policy message entry has an ID that is set by the source. Entries in an offer message are sent in response to requests or policies; the ID of an offer is identical to

the ID of the corresponding request or policy. Certain offers could be gratuitous (not sent in response to a request but just because that particular negotiation context requires it); the IDs of these offers do not overlap with the request/policy IDs.

- Array of entry referrals: If request  $R_1$  is sent in response to received request  $R_2$  (counter-request), the referral ID for  $R_1$  is the ID of  $R_2$ .
- Extra (support) information describing the entries: These are lists of predicate statements or objects that support, or describe, the main entry predicates. For example, an entry referring to an object `possess(V)` could have an extra predicate `certificate(V)`, which indicates that the requester wants a voucher. An offer message could contain the objects requested, or proof of compliance (e.g., certificate).
- Optional information: This field is used in offer messages to send information in the form of variable bindings. For example, a request for information pertaining to location could be posed as `location(L)`, and the option field in the offer reply would contain the answer in the form of `{L='BoelterHall'}`.
- Timestamp: This field contains the time (as monitored by the message source) at which the message was sent. It is used for fault tolerance. The role of this field will be described in Chapter 6 when we discuss fault tolerance.

The only reason (but an important one) for bunching several entries in one message is to make communication more efficient, otherwise a negotiation will take many more steps (and consume a lot more bandwidth) than is necessary.



## 5.2. Protocol Semantics and State Machine

At the highest level, the negotiation protocol is a lightweight procedure that involves exchange of request, policy and offer messages, culminating in a terminate message. The protocol state machine is illustrated in Figure 7. It is deterministic and facilitates peer-to-peer communication between two peers. After one entity initiates the protocol, all ensuing communication is completely symmetrical. The machine consists of five states, though the *stop* and *start* states are identical from the point of view of a policy manager (they are different from the point of view of the middleware that incorporates the policy manager, and must perform certain cleanup actions after the *stop* state).

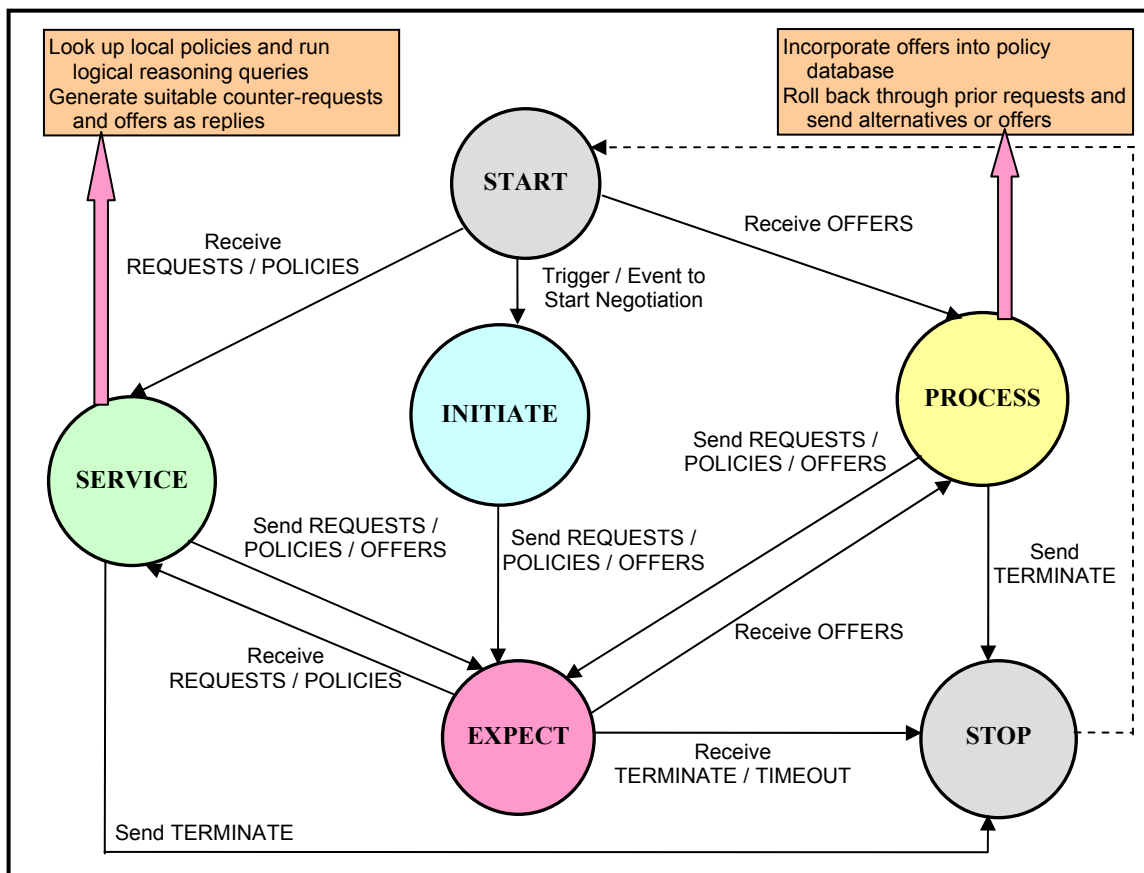


Figure 7. Negotiation Protocol: High-Level State Machine

Only the entity that initiates negotiation goes into the *initiate* state. Once negotiation is under way, either entity can go into the *service/process* or the *expecting* states. In the *expecting* state, a reply could be received from the remote policy manager and nothing needs to be done locally; local considerations might indicate whether a timeout is required, in case the remote entity can't communicate due to failure or network partition. In the *service* state, a message consisting of *directive* speech acts (requests and policies) is received from the remote policy manager; the local policy engine makes a decision on whether or not to satisfy them, and sends a suitable reply. In the *process* state, a message consisting of *commissive* speech acts (offers) is received; the policy engine decides whether the offer is satisfactory and sends a suitable reply. The processing actions are different in these two states, and are described in more detail in Figure 8. This state machine also handles faults (see Section 6.6), as indicated by the timeout link from the *expect* state to the *stop* state.

The message processing and reply procedure is described through the flowchart in Figure 8, divided into two parts for better understanding; the color code is identical to that in Figure 7. Negotiation starts with one entity sending request(s) to the other; these requests are derived from the initiator's goals and requirements. Each negotiator maintains two lists, one consisting of directives (requests and policies) posed to the negotiator, and the other consisting of directives received from the negotiator. A received request can be 'satisfied' by sending either an affirmative or a negative offer. Both of these are definitive, indicating whether the posed request can be obeyed under current circumstances. When an offer is received at the other end, the corresponding request is

removed from the list. The list is like a stack in some ways, since later requests must be satisfied prior to earlier requests; the exception is that requests sent in a single message have no ordering, and therefore violate the last-in-first-out property of a stack. If definitive offers cannot be returned, counter requests and policies can be sent in response to a request, causing the list of posed requests to grow at the sending end and the list of received requests to grow at the receiving end.

We classified the negotiation messages of the directive type broadly into requests and policies. There is no particular reason to separate the two in an implementation of the negotiation protocol, because the ‘satisfaction’ of a policy results in an offer that is similar to an offer that ‘satisfies’ a request. (In our original implementation, the two were separated because we assumed that requests and policies would be processed in different ways, but further research and application development unearthed no substantive reasons to do so. Hence the flowchart in Figure 8 uses two lists, one for the received requests and another for the posed requests. This also makes the flowchart simpler to understand.)

Offer messages contain either logical truth values (*true* or *false* to indicate satisfaction or dissatisfaction of requests) or query answers in the form of variable bindings in the ‘options’ field. Objects (such as requested credentials, files, or proofs in any form) may also be appended to an offer message. A negative offer results in the removal of the corresponding posed request from the list that contains it. If this request (say  $R_1$ ) is a counter in response to a prior request received (say  $R_2$ ), and there are no alternatives to  $R_1$  (we will discuss the concept of alternatives later),  $R_2$  is invalidated and the lists are rolled back through the removal of ‘satisfied’ requests at the top. On the other

hand, a positive offer results in certain policy assertions being made to the database. But before assertions are made, the objects sent as attachments are verified for integrity, if valid verification procedures exist. If verification fails, an OFFER-REJECT is sent; the other side now has an opportunity to resend a valid offer (for example, if network errors caused an integrity check to fail). It is not absolutely necessary to have this feature in the negotiation protocol, since it has a couple of serious drawbacks: (i) it results in extra steps and wasted bandwidth, and (ii) a malicious negotiator could keep resending a different invalid offer in multiple rounds, thereby effecting a denial-of-service attack. One could limit the number of offers received for the same request to mitigate such an attack, but the resulting bandwidth usage would still be suboptimal.

The procedure that generates counter-requests is explained in detail in Section 5.3. For our current purposes, it is sufficient to say that if  $N_1$  receives a request  $R$  from  $N_2$ , the former generates multiple sets of alternative counter-requests by running this procedure.  $R$  is satisfied by  $N_1$  only if every counter-request in a set is satisfied by  $N_2$ . One alternative is selected at a time, and formatted into a set of requests; these requests are then sent in a single negotiation message.  $N_1$  saves the remaining alternatives in a hash table indexed by the unique ID of  $R$ . If any one of the counter-requests results in a negative offer, a different alternative is selected and sent. If no more alternatives remain,  $R$  is removed from the received request list and rollback occurs.

The negotiation protocol terminates when both the posed request and received request lists of either side become empty. If a negotiator  $N$  receives an offer, cannot find any alternatives, and also finds both of its request lists empty, it will send a terminate

message. As Figure 8b illustrates, as long as the list of received requests contains a single entry, an offer is sent (which, in the absence of alternatives, will be a negative offer). Therefore, since every received request has a corresponding posed request at the other end, it is enough to check if the received request list is empty. If N not the initiator, then its posed requests list must be empty, as its received requests list is populated first and any requests posed are counters to those. (By the nature of the algorithm, counters to request R must be satisfied and removed from the list before R can be removed.) If N is the initiator, it must have already received offers for all the requests it posed; hence its posed requests list must also be empty. Therefore, we can prove that the algorithm terminates only when the request lists on both sides are empty. We prove this termination property formally in Chapter 8.

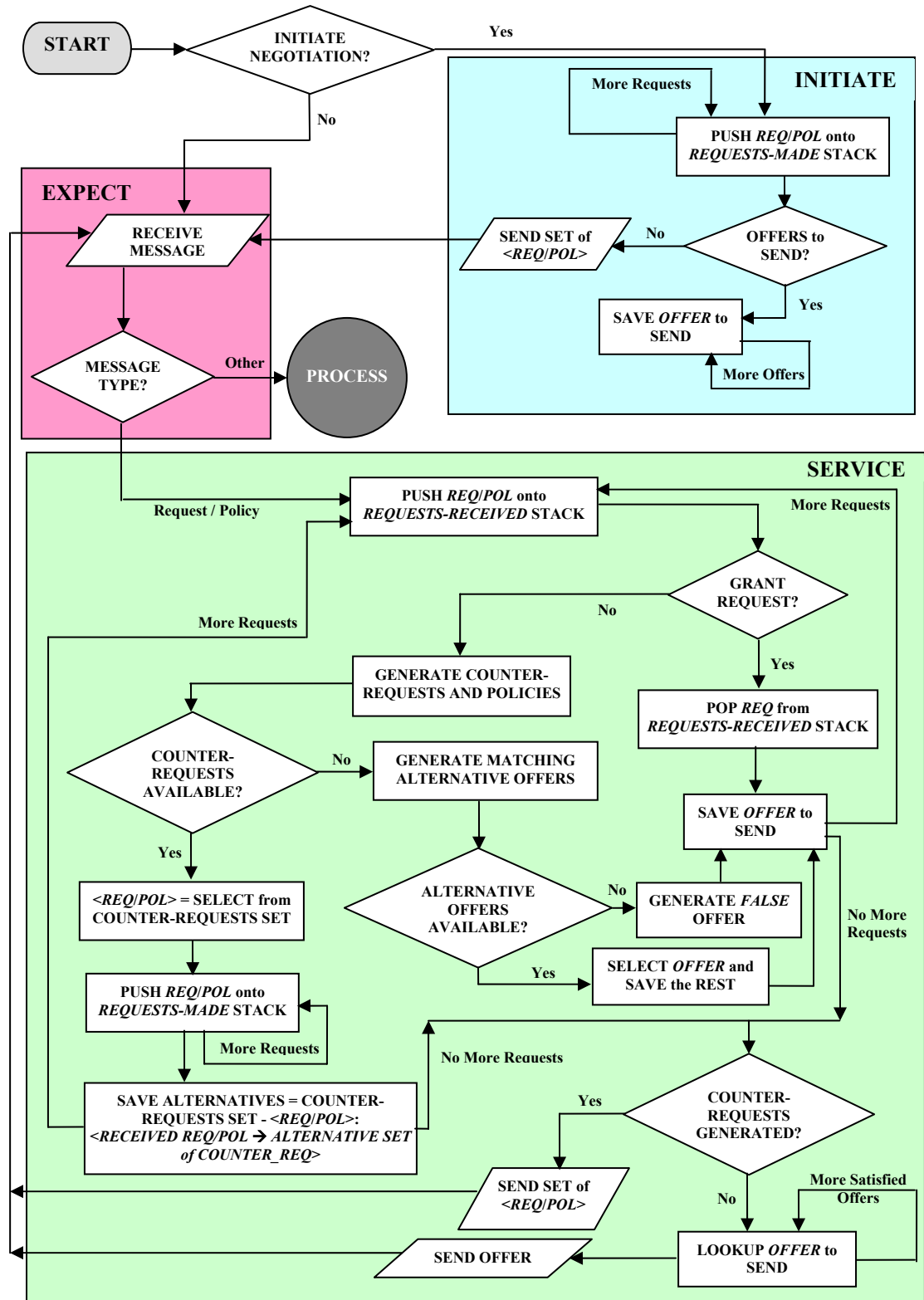


Figure 8a. Negotiation Protocol: Request Queue Processing: *Servicing Requests*

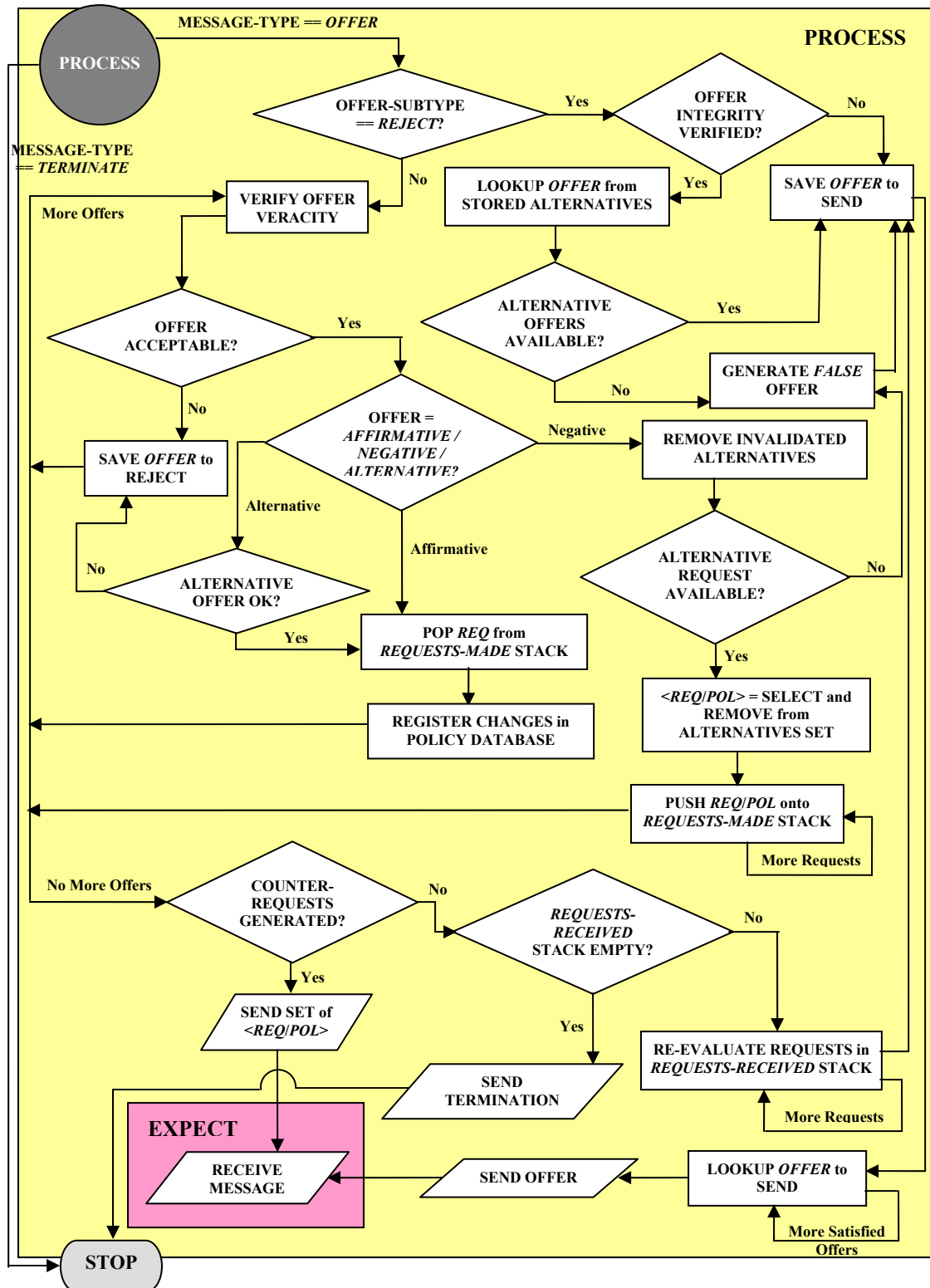


Figure 8b. Negotiation Protocol: Request Queue Processing: *Processing Offers*

### 5.3. Negotiation Query Resolution Algorithms

As Figure 7 indicates, message processing takes place in the *service* and *process* states. The flowchart in Figure 8 illustrates the parsing of request and response messages. Before appropriate actions are taken, the policy database must be examined and manipulated. We describe the procedure exported by the policy engine to generate a set of counter-requests and policies when a request cannot be granted right away. We also describe the procedure to find matching alternative offers if satisfactory counter-requests cannot also be found.

#### 5.3.1 Generating Counter-Requests and Policies

In Chapters 3 and 4, we stated that policies in our language have the syntax and semantics of Horn clauses (i.e., a conjunction of predicates that implies another predicate) by virtue of being written in Prolog. For example, a service access predicate could take the following form:

$$\text{access}(\text{ServiceName}) \text{ :- } P_{11} \wedge P_{12} \wedge \dots \wedge P_{1m}.$$

A query run on the policy database would check whether *access (ServiceName)* returns a *true* result or a set of variable bindings  $\{\text{ServiceName}='S_1, \dots, \text{ServiceName}='S_n'\}$  if and only if none of the conjuncts  $P_{ij}$  evaluate to *false*. In any such evaluation, some of these conjuncts will be satisfiable while the rest will not. The counter-request generation procedure is run when a request posed as a query evaluates to false. The procedure tries to infer which of the conjuncts are satisfiable and which are not, and candidate sets of counter-requests are drawn from the pool of currently unsatisfiable conjuncts. For example, if  $m=5$  in the above *access* policy,  $\{P_{11}, P_{12}, P_{14}\}$



is satisfiable while  $\{P_{13}, P_{15}\}$  is not, then counter-requests could be generated from  $P_{13}$  and  $P_{15}$  and sent to the negotiator. The latter would run queries to see whether it could satisfy these predicates (suitably formatted); if it can, it sends back an offer, and if not, it runs a counter-request generation procedure on its own database.

The high-level procedure that generates counter-requests is given below. We implicitly assume that the query predicate provided as argument is not currently satisfiable. If it is satisfiable, the procedure will return the null set. We outline a full example later, though we relate each step in the algorithm below to a relevant portion of the example for better understanding.

1. Given a query, all the matching (or relevant) policies are looked up. A relevant policy is a clause in the database whose *head* matches both the predicate name and the set of arguments of the query. This step involves generating a meta-query (provided by SWI-Prolog, our implementation framework); this procedure is quite efficient, taking only a few milliseconds (see Chapter 8), as it is performed completely in the Prolog subsystem.
2. Conjuncts in the body of each matching clause are separated into two sets. One set consists of all the predicates  $P_{ij}$  whose satisfiability depends on properties and policies of the opposite party (negotiator). The remaining predicates should (if at all possible) be satisfiable locally. This separation is performed by matching the variable referring to an autonomous (and in this context, negotiating) entity in the clause head with the variable arguments in the other predicates. For example, consider the clause:

```

access(S,F)    :-    file(F),    location(S,L),    groupSize(G),    G<=4,
                    closedPort(S,25).

```

This policy governs access to a particular file (most real policies would have additional constraints on the file, but let us consider a small policy for simplicity).

The argument that refers to a negotiating entity is  $S$  in the clause head `access(S,F)`.

Therefore, we classify the conjuncts in the body in the following way:

- Predicates that must be locally satisfiable (*local set*) =  $\{\text{file}(F), \text{groupSize}(G), G \leq 4\}$
- Predicates that depend on the state of the negotiator (*remote set*) =  $\{\text{location}(S,L), \text{closedPort}(S,25)\}$

Predicates in the latter set contain the variable  $S$  as an argument.

The request that would trigger the lookup of such a policy would be of the form  $\{\text{possess}(F), \text{file}(F)\}$ . At the receiving end, a query “`possess(F), file(F)`” would be run to verify that a *file* does exist in the database and could conceivably be sent to the requestor. After that, the request would be reformatted into the form `access(S,F)` with suitable variable bindings  $\{S=\langle \text{negotiator\_ID} \rangle, F=\langle \text{filename} \rangle\}$  so that the appropriate policy clauses can be looked up.

3. The conjunction of the predicates in the *local set* is evaluated to obtain suitable variable bindings. If the result is *false* (for example, if `groupSize=5`), no counter-requests can be generated that could satisfy this policy, and the procedure returns the null set. Otherwise, a set of variable bindings is generated.
4. For each such set, the predicates in the *remote set* are unified with the binding.

5. Each unified predicate is now examined, first to see if it is currently satisfiable (based on known state of the negotiator, as asserted in the policy database). If not, the predicate name is examined to see whether it is part of the globally understood vocabulary. For example, the predicate *location* is understood across domains, and so can be saved and sent as a counter-request. On the other hand, a predicate *closedPort* is only part of the local vocabulary. So now, the predicate `closedPort(S,25)` is examined recursively (starting from step 1 of this algorithm) to generate a set of counter-requests. For example, the following clause would be examined:

`closedPort(S,P) :- pred1, pred2, . . . . . , predk, action(S,closePort,P) .`

- Assuming that the *remote set* for this clause is  $\{\text{action}(S, \text{closePort}, P)\}$ , and the *local set* is satisfiable, and the predicate *action* is part of the global vocabulary, this set will be returned. This process proceeds recursively until global predicates are reached, or *leaves* (or database *facts*) are reached.
6. The contents of this set will be added to the ones gathered through examination of other predicates at the higher level (in this case, the *access* clause). In our example, we have seen only one clause being examined, but in practice multiple relevant clauses of the form `access(S,F)` and `closedPort(S,P)` will be present in the database, leading to multiple sets of counter-request predicates being generated. At any recursion level, if a null set is returned (indicating failure of a conjunct), our counter-request generation procedure returns a failure.
  7. At the end, the collection of request predicate sets generated is *minimized*; i.e., any request set that is *dominated* (see Section 4.3: *DominatingTerms*) by another is

eliminated in order to generate the most general and fewest possible alternatives. This step results in fewer negotiation steps, but in practice increases the running time of this algorithm; hence a tradeoff is involved here.

The above algorithm is a simplified version, and depending on the policy rule, additional inference steps would be required. Policy rules could also contain dependencies between predicates in what we refer to as the *remote* and *local* sets. For example, let us augment the *access* clause above to get a new policy:

```
access(S,F) :- file(F), location(S,L), groupSize(G), G<=4,
               closedPort(S,25), possess(S,V), voucher(V).
```

When `possess(S,V)` is selected to be a counter-request, `voucher(V)` is also selected and associated with it as it describes `possess(S,V)` with more precision. The dependency between the two predicates is inferred from the fact that they share the variable `v`. If recursion on a selected predicate is required, the associated predicates are passed down as arguments. As indicated in Figure 8a, if multiple counter-request sets are generated, one is immediately sent, and the others are saved as alternatives indexed by the ID of the original request received.

***Example:***

- Let the request posed by a negotiator (whose ID is 'neg') take the form `{possess(F), file(F)}`. This request is reformatted to the following query: `<possess(F), file(F), access(S,F)>`; the binding is `{S='neg'}`. This query is

posed to the policy database, which contains the following statements, among others.

We only list the statements that are relevant to the procedure.

Facts:     `{possess('SomeFile'),       file('SomeFile'),       groupSize(5),  
trustedDomain('ACM'), trustedDomain('UCLA')}`.

Relevant policies:

- (I)   `access(S,F) :- file(F), location(S,L), groupSize(G), G<=4,  
                  possess(S,V), voucher(V,M), trustedDomain(M).`
- (II)   `access(S,F) :- file(F), groupSize(G), G>4, closedPort(S,25),  
                  possess(S,V), voucher(V,M), trustedDomain(M).`
- (III)   `access(S,F) :- file(F), location(S,L), groupSize(G), G>4,  
                  closedPort(S,25), possess(S,V), voucher(V,M), trustedDomain(M).`
- (IV)   `closedPort(S,P) :- pred1,   pred2, ..... action(S, order,  
                  closePort, P), ..... , predk. (All predicates apart from the action  
                  predicate are irrelevant to this example).`

- Policy statements indicating what the global vocabulary is are listed below:

`{global(closePort), global(file), global(voucher),  
requestable(possess), requestable(location), requestable(action)}.`

- The generated query fails, because evaluating every clause with the access predicate in the head results in failure. The sub-query `<possess(F), file(F)>` is first tested and found to result in a solution `{F='SomeFile'}`.
- Now a counter-request generation is triggered. First, every clause corresponding to `access(S,F)` as the head is looked up using the following query

$\langle \text{clause}(\text{'access}(S, F)', \text{Body}, \text{Reference}) \rangle$ . The bodies of policies I, II, and III are returned, bound to the variable `Body`. From each body, the local and remote sets are computed as follows:

(I) Local Set =  $\{\text{file}(F), \text{groupSize}(G), G \leq 4, \text{voucher}(V)\}$ .

Remote Set =  $\{\text{location}(S, L), \text{possess}(S, V)\}$ .

(II) Local Set =  $\{\text{file}(F), \text{groupSize}(G), G > 4, \text{voucher}(V, M), \text{trustedDomain}(M)\}$ .

Remote Set =  $\{\text{closedPort}(S, 25), \text{possess}(S, V)\}$ .

(III) Local Set =  $\{\text{file}(F), \text{groupSize}(G), G > 4, \text{voucher}(V, M), \text{trustedDomain}(M)\}$ .

Remote Set =  $\{\text{location}(S, L), \text{closedPort}(S, 25), \text{possess}(S, V)\}$ .

- Queries are constructed by the conjunction of predicates in each remote set. The query fails if the body of clause I is evaluated, because `groupSize(5)` is asserted as a fact. The query partially succeeds for clauses II and III (`voucher(V, M)` is not satisfiable). But in these cases, we still continue with the examination by binding the unsatisfied variables to free variables, rather than constants. For example, `M` is bound to `'ACM'` or `'UCLA'` in turn, whereas `V` is bound simply to `V`. This lets us generate requests that, if accepted, would also result in additional predicates like `voucher(V, M)` being satisfied. Here, if this negotiator poses a request  $\{\text{possess}(S, V), \text{voucher}(V, \text{'ACM'})\}$  and receives a voucher named `'voucher'` in return, it will be able to assert the predicate `voucher('voucher', 'ACM')`, thereby resulting in the satisfaction of the query derived from the local set. Such predictive

binding results in negotiation success wherever possible. The bindings resulting from our query are  $\{F='SomeFile', G=5, V=V, M='ACM'\}$  and  $\{F='SomeFile', G=5, V=V, M='UCLA'\}$ .

- Each predicate in the remote set is unified with each binding generated above in turn. In our scenario, this does not result in any change, since predicates in the remote set do not share any variable with those in the local set, apart from  $V$ . Each unified predicate is queried in turn, using the binding  $\{S='neg'\}$ . We ignore predicates whose queries result in success. In this scenario, each unified remote predicate  $location('neg',L), closedPort('neg',25), possess('neg',V)\}$  results in failure. Therefore, we may add these predicates to our candidate counter-request sets as long as they are part of the requestable vocabulary. But the asserted facts  $\{requestable(possess), requestable(location), requestable(action)\}$  do not indicate `closedPort` as being a candidate request. We therefore recursively run the counter-request generation algorithm on  $closedPort(S,25)$ , with  $S='neg'$  as a binding. This results in the examination of clause (IV) and the extraction of  $action(S,order,closePort,25)$  as a counter-request. The predicates  $pred_1, pred_2, \dots, pred_k$  are low-level JPL predicates [SWIProlog] that always evaluate to a true value.
- The arguments of each extracted request predicate are examined to see whether they are global arguments (by running the query  $\langle global('argument') \rangle$ ). In the predicate  $action(S,order,closePort,25)$ , `closePort` and `25` are global, because the former is asserted through the  $global(closePort)$  fact and the latter is a

number. Any argument that is not found to be part of the global vocabulary is replaced by a variable (i.e., it is obfuscated).

- For each request predicate, support predicates are extracted from the corresponding local set. `location(S,L)` and `action(S,order,closePort,25)` do not have any support predicates. `possess(S,V)` has a support predicate, which is `voucher(V,M)`, because they share a common variable V.

- Examination of clause I thus results in no candidate counter-requests.

Examination of clause II results in two alternative sets:  $A1 = \{\text{action}(S, \text{order}, \text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'ACM')\}$  and  $A2 = \{\text{action}(S, \text{order}, \text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'UCLA')\}$ .

Examination of clause III results in two alternative sets:  $A3 = \{\text{location}(S, L); \text{action}(S, \text{order}, \text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'ACM')\}$  and  $A4 = \{\text{location}(S, L); \text{action}(S, \text{order}, \text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'UCLA')\}$ .

- Running the minimization procedure, we find that alternative A1 dominates A3, and A2 dominates A4, since they share `action` and `possess` predicates with identical arguments, but both A3 and A4 have extra `location` predicates. Since a request set always dominates its superset, the counter-request generation procedure returns the sets A1 and A2.
- Though not part of this procedure, we will note that the requests in A1 and A2 are formatted to  $\{\text{action}(\text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'ACM')\}$  and  $\{\text{action}(S, \text{closePort}, 25); \text{possess}(S, V), \text{voucher}(V, 'UCLA')\}$  respectively,



and are sent to the negotiator in turn, based on some (application dependent) heuristic selection.

**Comment on Policy Language Features:** In Chapter 4 we mentioned that a policy language for negotiation should be monotonic and support constraint extraction. Counter-request generation is a procedure of extracting constraints in various ways. One, it finds the set of relevant policies that affect a request (that drives negotiation), and two, it finds the unsatisfied conjuncts that could be formed into counter-requests that the opposite party must satisfy. The monotonicity property can be observed from the fact that once these constraints are satisfied (as offers in reply to requests), the negotiation moves *forward* (i.e., in a negotiation step, a pair  $\langle request, true-offer \rangle$  always results in a policy being more satisfiable than it was before that step.)

### 5.3.2 Generating Matching Alternative Offers

The procedure for generating alternative offers is somewhat simpler. The aim is to find the matching satisfiable predicates that are closest to the one that was requested (and could not be satisfied). As in the counter-requests generation algorithm, the clauses that are relevant to the given request are looked up and queried at a higher layer of abstraction (all arguments in the request predicate are replaced with variables). All the predicates that can be associated with the primary request predicate (as `voucher(V)` was associated with `possess(S,V)` above) are generated. All possible sets of such associated predicates are generated along with the variable bindings that make them satisfiable. These are then

ranked based on their closeness to the original request predicate. The closest match is sent as an alternative offer, and the remaining sets are saved in case this offer is rejected.

**Example:** Consider the case when a request is made to turn off sound from 1200 to 1600 hours through the request predicate `sound(prohibit,1700,1800)`. The recipient has policies that prevent it from complying with this request, but it does have other facts in its database that enable it to agree with prohibitions on sound during different times, such as `sound(prohibit,0,1500)` and `sound(prohibit,1750,2400)`. The alternative offer generation procedure finds all asserted facts and rules matching the `sound` predicate with three arguments. It finds the above two predicates, and attempts to rank them. In this case, both predicates differ from the original request in their second and third arguments, so there is nothing to choose between them. One is selected based on some heuristic (perhaps the one with lower time duration) and sent as an alternative offer.

### 5.3.3 Generation and Verification of Affirmative Offer Messages

Suitable proofs or objects associated with request predicates are generated or fetched using pre-programmed helper functions (or methods). For example, a predicate `voucher('UCLA')` would indicate that a voucher granted by UCLA must be looked up and attached. A predicate `voucher` would indicate that a voucher must be generated and granted to the requestor. A predicate `voucher('UCLA','5/16/2008')` would indicate that a voucher from UCLA that is valid upto at least 16 May 2008 must be looked up and

attached. If a suitable object cannot be sent as attachment (for example, if a voucher has expired), a negative offer is sent.

Correspondingly, an attached object is verified in the offer processing stage at the receiving end. As in the generation case, the predicate names and arguments would indicate the appropriate test to be conducted. This does not necessarily have to involve the inspection of objects. A request to close port 25 could be verified (with some level of confidence) by using a port scanning tool like *nmap*. Failure of a verification test results in an OFFER-REJECT being sent, as indicated in Figure 8b.

## Chapter 6

# Design and Implementation of a Policy Manager for a Ubiquitous Computing Environment

In this chapter, we describe the design and implementation of the negotiation protocol (see Chapter 5) within a full-featured ubiquitous computing platform called Panoply [Eustice2003a; Eustice2008b]. As mentioned in the introduction (see Chapter 1), the negotiation protocol is the most prominent (though not the only) contribution of this research. We show how a policy management framework was designed for Panoply, and list the services it provides to the infrastructure and to applications. We describe the design and implementation of a dynamic access control framework that utilizes the negotiation protocol. Other important features and challenges, including handling multiple negotiations, either concurrently or sequentially (*renegotiation*), are also described in detail. We show how the protocol was augmented to make it tolerant to local and network faults. We conclude by describing how users can visualize negotiation and interface with the policy engine during runtime.

### 6.1. Device Communities and a Ubicomp Middleware Platform

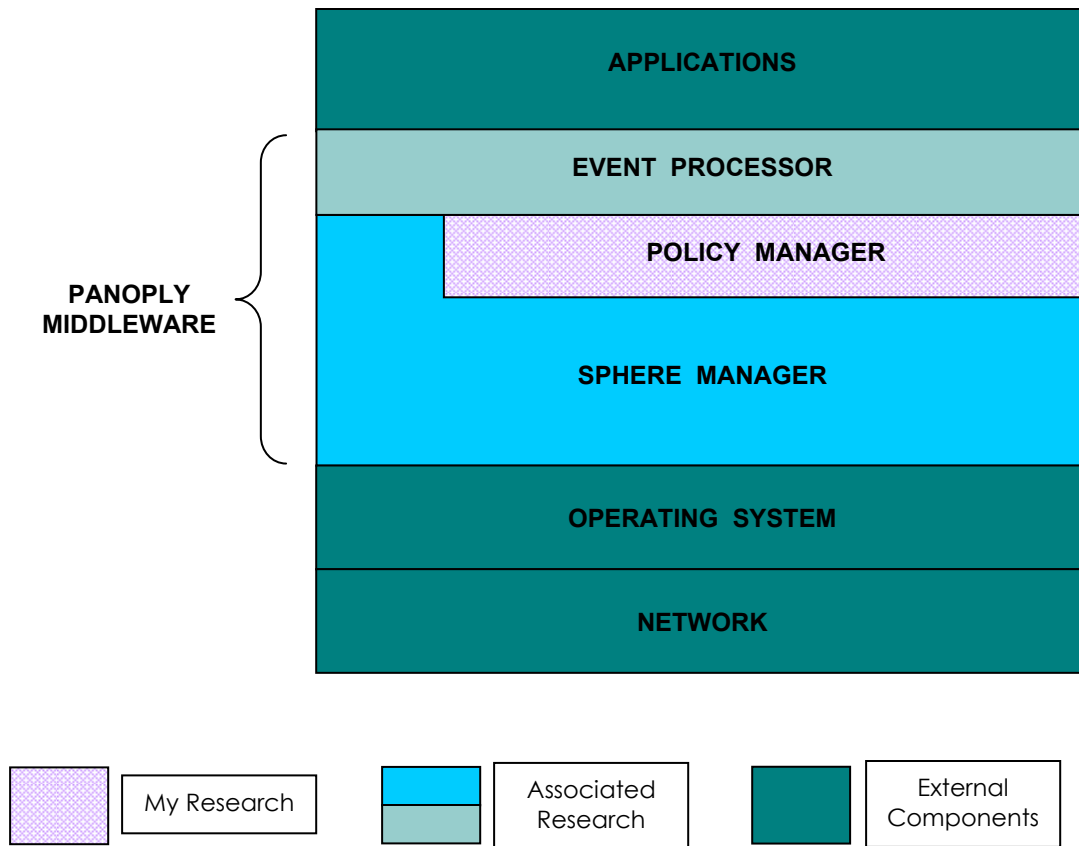
The theory and design of *device communities* that can manage resources (in a centralized manner), impose policy, and ensure secure communication, both within and with the

outside world, is an important research topic in its own right. We described the properties of such communities in the form of domains (see Chapter 2) that interoperate within a global scale ubiquitous computing system. Modeling device groups and handling associated research issues like device and group discovery, event management, secure networking and firewalling, and providing primitives for building context-sensitive applications, which are orthogonal to the policy management and negotiation issues, were contributions of Kevin Eustice in his PhD research [Eustice2008b]; this project is called *spheres of influence*. The spheres concept combined with negotiation resulted in a middleware for ubicomp called *Panoply*. The remainder of this section summarizes the key contributions of his dissertation that impact or are complementary to policy management. Details can be obtained from Eustice's dissertation. Our research does not absolutely depend on spheres, since groups or domains can be virtualized by a single computer, and existing security and MAC protocols could be used for wireless communication. The policy management and negotiation mechanisms are independent of the Panoply design, and could work for other ubicomp platforms with minor adjustments. Still, management of policies and addition of the negotiation concept to Panoply will demonstrate a powerful ubiquitous computing model.

### **6.1.1 Spheres of Influence and Panoply**

Panoply provides functions for applications to manage individual devices and groups. In Panoply parlance, such groups are called *spheres of influence*; a sphere can be built up recursively from smaller spheres, a single device being a unit sphere. Membership of a

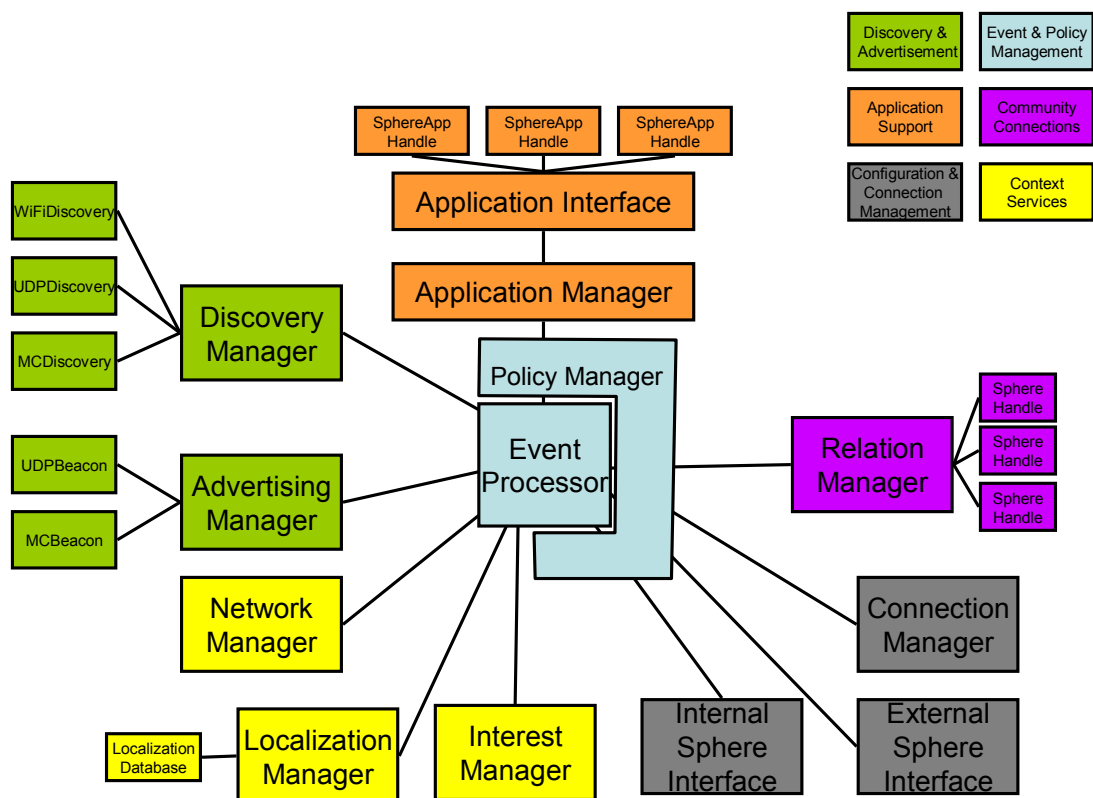
sphere within another is equivalent to a child-parent relationship. Devices in a group or sphere are united by a common interest or attribute such as physical location, application, or social relationship. Spheres unify disparate notions of groups, such as device clusters and social networks, by providing a common interface and a standard set of discovery and management primitives.



**Figure 9.** Overall System Architecture

The building of a sphere that acts as a unified entity when interfacing with the outside world is facilitated by Panoply. Figure 9 indicates what a Panoply-enabled system would look like, clearly differentiating between our and others' research. A Panoply *sphere manager* manages all activity and relationships within a sphere, and as such is the

equivalent of an operating system for a sphere. In every Panoply-enabled device, the sphere manager lies above the operating system and below the application layer (see Figure 9). A more detailed schematic of the Panoply middleware, with the functions of the sphere manager broken down into various components, is given in Figure 10.



**Figure 10.** The Panoply Middleware (Borrowed from Eustice’s PhD Dissertation [Eustice2008b])

A detailed discussion of the components in Figure 10 is beyond the scope of this section, and can be obtained from [Eustice2008b]; however, we list the relevant features in Table 1. Panoply provides group management primitives that allow the creation and maintenance of spheres of influence, including discovery, joining, and cluster management. All inter-sphere and intra-sphere communications are carried out via a

publish/subscribe event model that propagates events between devices and applications, subject to scoping constraints embedded in events and interest. An external sphere interface serves as the security boundary for a sphere and for interactions with external spheres, while an internal sphere interface is used to communicate with child spheres and to interact with applications. The sphere manager contains both a relationship manager and a policy manager. The latter manages local policy, answers queries, and negotiates with external spheres; it interfaces with other spheres and applications through special *policy events*.

**Table 1:** *Core Panoply Services (Borrowed from Eustice’s PhD Dissertation [Eustice2008b])*

High-Level Panoply Functionality	Core Services Provided
Group ( <i>Sphere</i> ) Management	Sphere creation Mediation of membership changes ( <i>join</i> and <i>leave</i> )
Event Communication	Generation of events Interest-based subscription for events Event delivery Scoping of events
Network Configuration	Network discovery and association Reachability verification through beacons Secure connection management
Context Sensing and Inference	Location discovery Mapping of physical to <i>semantic</i> locations Discovery of social group peers
Policy Specification and Enforcement	Inter-sphere negotiation Secure attestation using <i>vouchers</i> Event-condition-action triggers Access control through event filtering



Eustice's and my research have proceeded in tandem within the broad framework of the Panoply project. His sphere management and event dissemination implementations have provided a suitable platform for an implementation of negotiation, maintenance of a policy database with components that observe how the state of a sphere changes over time, and experimentation with event filtering on the basis of dynamic policy. Our experiences have taught us that there is a symbiotic relationship between the Sphere of Influence framework and the policy-guided negotiation framework. The latter ensures the integrity of sphere joining and communication, and provides a security blanket. Panoply ensures that appropriate events are sent to the policy manager, enabling it to keep its database updated and make suitable decisions about when and how to negotiate.

### **6.1.2 Role of the Policy Manager Within a Panoply Sphere**

Within a sphere, the policy manager module serves as a container for policies, which are enforced through both passive and proactive means. As indicated in Figure 10, this module forms a semi-envelope around the event processor, forcing all interactions of sphere modules (like the relation manager) and the sphere interfaces with applications to go through the policy manager. One function of this manager is to observe what is happening in a sphere that affects its state and register these changes as facts in its database. These changes can be observed by subscribing to relevant events. Some of these changes will also require the fulfillment of certain policy obligations; it is the role of the policy manager to trigger the fulfillment of these obligations. The formation of a relationship between two spheres always occurs through negotiation (through the

protocol described in Chapter 5). Also, applications obtain information and control sphere resources through events. By forcing the policy manager to mediate the flow of all such events, it can filter a number of these events based on policy (choosing to send, or drop, or modify the events) so that the interactions do not violate policy. Since policy could change at any time, the policy manager could negotiate with an application to make it prove that it has the necessary rights to receive/send events, thereby demonstrating its right to access a resource.

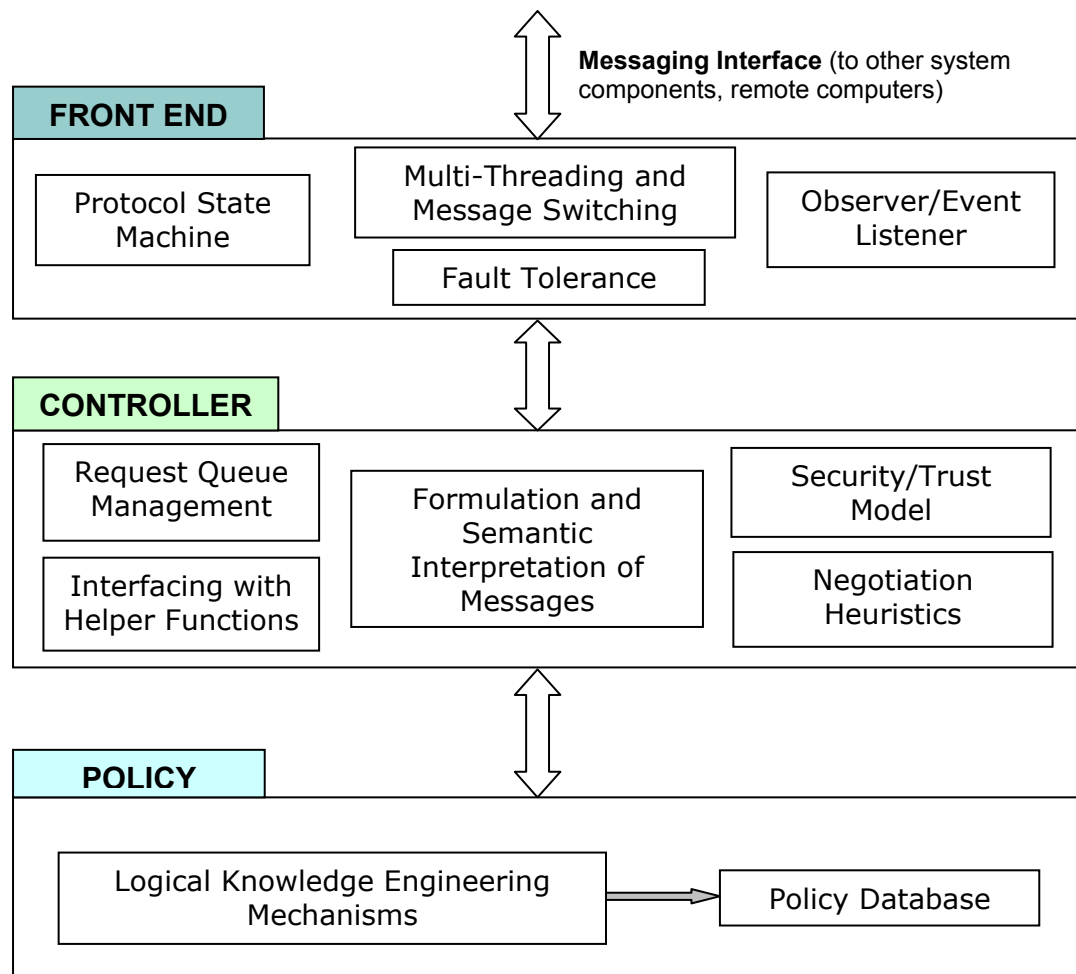
## **6.2. Panoply Policy Management Architecture**

The architecture of the policy manager can be functionally decomposed into three layers (see Figure 11). This choice of design was driven by the need to support negotiation as the core function, and made the most sense given our conceptualization of the protocol described in Section 5.2. We could clearly delineate three functions:

- 1) The negotiation protocol as a sequence of messages, based on the state machine illustrated in Figure 7.
- 2) The interpretation and parsing of the speech acts encoded within the messages, management of the request lists, interfacing with helper functions, and strategizing based on external heuristics and trust models.
- 3) An engine containing and maintaining sphere policies, and offering logical reasoning mechanisms such as the counter-request generation function to extract information.

These three parts are functionally independent of each other, proving the necessary flexibility. Even though we tied the high-level state machine in Figure 7 and

the message processing in Figures 8a and 8b (see Section 5.2), we could replace our flowchart with a different one that maintains request lists and strategizes in a different way. Also, helper functions, as plug-ins, could be replaced if required. Heuristic functions (based on resource needs, trust and risk models) that dictate whether alternative offers are acceptable, or the order in which alternatives are to be proposed, could be completely disassociated from the policies themselves and from the protocol state machine. Similarly, one could replace the knowledge engineering mechanisms offered by the policy engine with others, keeping the state machine and parsing mechanisms intact.



**Figure 11.** Policy Manager Functional Diagram

Our policy manager design is illustrated in Figure 11 above, with the three functions mapping to the three layers in the diagram from top to bottom. The *front end* is the policy manager shell that interfaces with other local sphere components, as well as interacting with remote spheres through Panoply events. It receives state change events from the sphere and applications, and communicates them to the policy engine, which updates the policy database and triggers suitable actions. It mediates information flow to Panoply applications by monitoring events. The negotiation protocol state machine runs here. Multiple simultaneous negotiation threads and the flow of negotiation messages are managed at this layer. Concurrent independent negotiations with multiple peers are supported, though all threads share a common policy database. More discussions of multiple negotiations, both concurrent and sequential, take place in Section 6.4. It is the front end layer that ensures that the negotiation protocol is reliable and tolerant to policy engine and network failures. Users can also observe and modify policies through a graphical interface that runs at this layer.

The *policy engine* manages the database containing state information and policy rules. It interfaces directly with the SWI-Prolog subsystem using a bi-directional Java-Prolog API; the actual facts and rules are maintained by this subsystem, which uses efficient compilation, indexing and retrieval techniques provided by the Warren abstract machine [Aït-Kaci1991]. It recognizes policies written in the language described in Chapter 4, and exports methods for manipulating the database and extracting information. These methods include the simple querying operations, addition and removal operations, which were described in Chapter 4, as well as the counter-request and alternative-offer

generation mechanisms described in Chapter 5. Exactly one instance of a policy engine runs within every Panoply sphere.

The *controller* guides and controls the rate and the strategy of negotiation. Every negotiation thread has its own controller. It is responsible for *unpacking* the contents of the negotiation messages received from the front end, examining them and formulating an appropriate response. The controller effectively runs the algorithm illustrated in the flowchart described in Figures 8a and 8b. It maintains the lists of requests received, requests posed, and alternative offers. It interfaces with the policy database through the policy engine layer, invoking the methods offered by the latter. The decisions regarding which alternative requests to send, or whether to accept an alternative offer, and what criteria to apply when making such a selection, are performed at this layer. One could fit any heuristic into this layer that would guide such selection, which would directly impact the negotiation strategy, since a prudent selection could result in a shorter and less risky negotiation. Negotiation heuristics could include trust and risk models, so that the risks of making an offer and the trust one has in the opposite negotiator could be evaluated against each other. We currently do not have such a trust/risk model in place, but a lot of related work exists in building heuristics based on game-theoretic, utility-theoretic and economic considerations [English2004]. We will discuss these models in Chapter 10; such models can be plugged into the controller with minimal additional work.

The controller also interfaces with external (helper) functions, which perform various operating system operations or object processing functions that are separable from the logic of the negotiation protocol and independent of the policy database.

Referring to the negotiation message format described in Chapter 5, a message could contain not just queries and responses, but also supporting objects and requested proofs, ranging from simple string tokens to credentials, data files and mobile code. It is the controller's task to attach suitable objects to each message, which are then extracted and verified through a converse operation by the controller at the opposite end. For example, in the case of a message containing a *request* to run a virus-scanner, the controller would find the scanning code and attach it to the message. In a message containing an *offer* of a certificate, the controller would look for, extract, and validate the certificate object; different decisions are made depending on whether the correct offer was sent. We cannot handle all possible objects of interest in every ubicomp scenario, so the controller allows pluggable helper functions and mechanisms for different objects as the need may arise.

### **6.3. Features Supported Through Policy Management**

The Panoply model of ubiquitous computing consists of self-contained units (spheres) that scope policy within security perimeters and obtain services or resources not available locally through event-based interactions with external units. These spheres grow or shrink based on certain imperatives; these imperatives could be immediate (service or resource access) or long-term (sharing a common purpose, interest-based groups, etc.). Spheres host applications that access resources and interact with other applications (running both on local and remote spheres). The policy manager tries to ensure that all sphere activities are conducted in a proper manner. The only Panoply service it relies upon is the message passing framework, modeled on event publishing/subscription. From the point of view of

the rest of the sphere, the policy manager fulfills its goals purely by sending and receiving (subscribing to) events. Broadly, it serves three purposes, which we detail below. These services are proactive in nature.

### **6.3.1 Mediated Interactions: Negotiation**

Spheres can grow by allowing other spheres in as members, they can shrink when members leave, and they can form sibling relationships with other spheres. Panoply provides the mechanisms for the initiation of such interactions and the breaking of relationships, but the *terms* and *limits* of these relationships are set by the policy manager through the negotiation protocol. The most common case we have seen in practice is when one sphere wishes to be a member of another. Panoply enables this through a simple JOIN protocol as follows:

*Step I:* The client (C) sends a JOIN-REQUEST to the prospective host (H).

*Step II:* If H is willing to accept C as a member, it replies with a JOIN-SUCCESS, otherwise it replies with a JOIN-FAILURE.

*Step III:* In case of a JOIN-SUCCESS, H adds C as a member, registers its event interests, and brings it within its security perimeter.

The policy manager mediates Step II, and is responsible for deciding whether H would be willing to accept C as a member. Deciding whether or not a sphere should be a member of another could result in a negotiation with arbitrary possibilities, and allows us to experiment with a variety of scenarios and policies. At the basic level, C would initiate negotiation by sending a NEGOTIATION-REQUEST message containing a request for

membership. H would examine its policies governing negotiation, and make a positive offer, negative offer or send counter-requests. Counter-requests could include credentials and promises of resource-sharing, and obligations that C must abide by as a sphere member. When this negotiation instance terminates, H examines the result of the original membership request and continues the JOIN protocol as indicated above.

In such a negotiation, the client C may initiate negotiation by making multiple requests, one of which would be a membership request. For example, a mobile device sphere (represented by a laptop) may wish to gain entry into our office sphere, and in addition, obtain a certain amount of network bandwidth and Internet access. H would evaluate the requests against its policies that allow network bandwidth and Internet access only to valid members, and continue the negotiation in a way that would result in granting those requests pending satisfaction of the membership request. Also, H could grant offers gratuitously as a consequence of granting a membership request. In this example, the office sphere may offer a certificate indicating presence within the office location for a particular duration; this could be used in future negotiations between the laptop sphere and other spheres. A Panoply-supported Interactive Narrative application (to be described in Chapter 7) [Eustice2007] that was deployed in the UCLA campus made use of this feature. Also, the conference room scenario in Chapter 1 indicates an offer of journal membership being made to a conference room attendee that has been granted network access.

Two spheres could interact without one trying to be a member of another. These could be spheres that have no prior relationship with each other and have discovered each



other through a mechanism that Panoply provides, or could be siblings that share a common parent. In either case, there exists a communication channel that permits bi-directional event flow. The spheres can negotiate through this channel for access to resources, permission to run certain services, change the terms of earlier agreements, and impose additional policy constraints.

### **6.3.2 Dynamic Access Control Through Event Filtering**

Sphere components and applications communicate through events. Some components and applications guard resources and information that are not meant to be easily accessible to everyone and under any situation. For example, a sphere component called the *relation manager* maintains a repository of the sphere's relationships, which is information that can be obtained through a pair of query-response events of a designated type. In another scenario that we call the Smart Party [Eustice2008a], dynamic song playlists are generated and played based on the collective preferences of the guests. The playlist offers the option to skip to the next song on the list or repeat the previous song. Such options, by their very nature, change the core behavior of the music-playing application. Therefore, the playlist is a guarded resource and options to change it must be restricted to authorized users. A simple access control policy would allow such access only to party hosts and not to temporary guests. A more complex policy would have associated contextual constraints, such as allowing anyone to change playlists as long as there are not more than three guests present. In another example, an office could have policies that mandate how its smart doors open and close and how its smart lights turn on and off

based on who is present in the office and the identity of the person who is manually attempting to change the state of the resource.

These resources are managed at runtime by either a core sphere component or a Panoply-enabled application. Access to these resources occurs through the sending of events to the resource controllers, which then perform the requested state-changing actions or send information through responses. By inserting the policy manager in the path from the sender to the resource controller, it can control access based on the policies in its database. First, before the event reaches the destination, its attributes—including the source, event type, subtype and application or user type (this can be expanded if required)—are extracted and a suitable query is formulated for the policy engine to evaluate. If the policy engine determines that the query is valid, the event is allowed to pass. If not, unsatisfied constraints are extracted (see Section 5.3), a new negotiation thread created, and these constraints are sent in the form of requests to the source of the event. A negotiation ensues (which could be as simple as requesting proof in the form of a credential), and upon a satisfactory conclusion, the event is allowed to pass; or, an unsatisfactory conclusion results in the event getting dropped.

The utility of this form of event filtering for access control is evident when policies, sphere state, and context dynamically change. For example, on “demo day” in a lab, the professor in charge of the lab sphere could temporarily grant permission to the students to open/shut the door without additional constraints. An event from a student’s sphere would be permitted to change the state of the door only when such a policy is active. When the policy reverts back to its original state, a door-state-changing event

would trigger a (presumably failed) negotiation with the student's sphere. Since access must necessarily occur through events, and events are matched with policy at the end of the event stream, revocation is not a problem. The distributed nature of the policies among the resource controllers (spheres and applications) avoids the scalability issues that afflict traditional modes of access control like security capabilities. Also, on-the-spot negotiation makes our model more dynamic and flexible when compared to existing access control models for open systems.

**Performance Issues:** Querying the policy engine for every query results in a huge performance hit. To avoid that, we run a query (and possibly, a negotiation) only for the first event in a stream, and if successful, save the result in a *policy cache* (which maps a tuple of event characteristics to a *true/false* value) that is unique to that particular recipient. Subsequent events of identical type are allowed to pass through without going through the policy engine if, and only if, the policy cache returns a *true* value, which indicates that the event source has proved its right of access. A FIFO policy is used to clear and populate the cache. This scheme has the obvious drawback that if a policy changes in the meantime, a number of unauthorized events may pass through.

### **6.3.3 Context-Sensitive Responses to Events**

Our policy manager provides the additional service of observing and recording sphere state and context changes, and triggering suitable actions as dictated by policy. As in the case of the above two services, building an observer module is made easier using the

subscription feature of the event processing framework. There are a number of ways in which sphere state and context can change, and these must be reflected in the policy database. Sphere relationships can change: a sphere could gain or lose members, contextual parameters (such as location) of a sphere could change, applications may start and shut down, and sphere components and applications could send state-changing events to each other (as described in Section 6.3.2). All these phenomena are accompanied by the publishing of events, and the policy manager registers its interests with the sphere manager by subscribing for these events. The only extra work the policy manager has to do here is to translate the contents of the events into logical statements that can be added to the policy database.

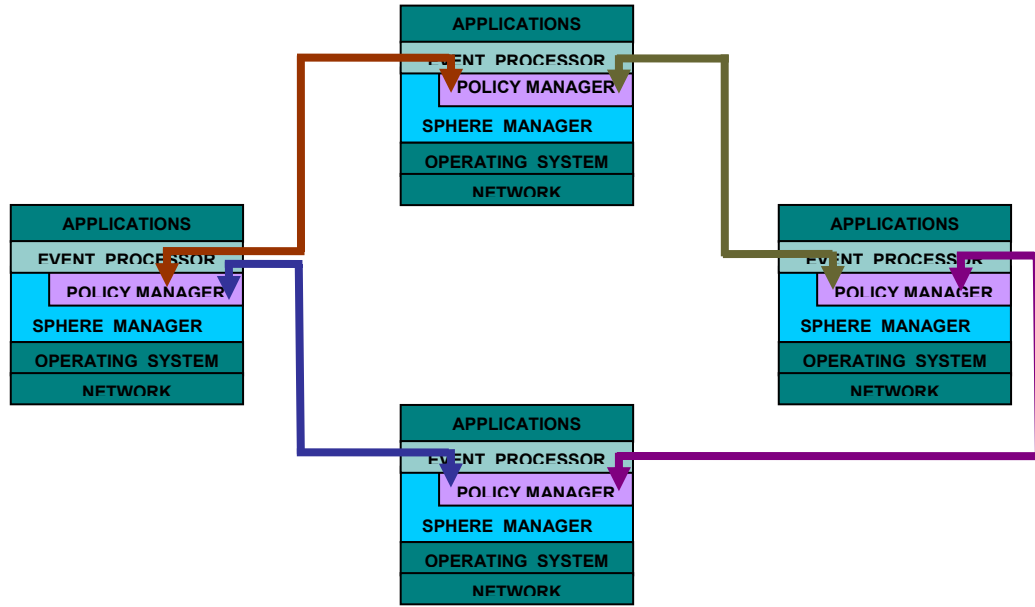
The policy database contains *update* policies (as described in Section 4.2) that dictate what actions must follow observations and database updates. For example, the satisfactory admission of a child to a sphere (through negotiation) could necessitate the creation of a credential certifying this newly created relationship. A smart door (mentioned in Section 6.3.2) could have a policy dictating that it must never be open for longer than five seconds during the daytime; the contextual parameter that changes and triggers a door closing action is time. The list of actions are inferred through the *chainEventAction* operation (see “Modification Operations” in Section 4.2) and then acted upon using available sphere mechanisms (either using Java method calls or by publishing Panoply events).

## 6.4. Multi-Threaded Operation

It is neither practical nor efficient for a policy management framework to support only one negotiation session at a time, making other potential negotiators wait for their turn. The Panoply policy manager supports multiple concurrent negotiations; each negotiation runs in its own thread but is dependent on, and impacts, the others. All negotiations are bilateral (one-to-one). Neither our protocol nor a Panoply policy manager support true  $n$ -party negotiation, where the  $n$  parties are actively aware of one another. Therefore, in our design and implementation,  $N_1 \leftrightarrow N_2$  and  $N_1 \leftrightarrow N_3$  may be concurrent negotiations, but from the point of view of  $N_2$  and  $N_3$ ,  $N_1$  is the only negotiator they are aware of. The policy manager also supports sequential renegotiations, which we will describe later.

### 6.4.1 Multiple Concurrent Negotiations

As indicated in Figure 11, the policy manager front end handles multiple negotiation threads and message switching. A sphere can negotiate with multiple other spheres concurrently, using the common event processing framework. These negotiations are direct communications between policy managers, and are transparent to the rest of the spheres, as indicated in Figure 12 below. Each negotiation thread is uniquely identified by the identity of the sphere it is negotiating with and maintains its own controller, and associated request/offer state.



**Figure 12.** Four Concurrent Negotiations

**Issues:** In our implementation, concurrent negotiations are unaware of each other. But since a sphere maintains only one policy database, which is shared by all negotiation threads, any changes made to the contents of the database through an intermediate step in one negotiation could impact and change the course of another negotiation. Two kinds of changes could occur in a database, *assertion* and *retraction* of facts and rules; *modification* is simply a combination of a retraction followed by an assertion. Granting and receiving offers typically result in database changes, and also any updates that are triggered by event-condition-action policy rules. Such changes could impact other negotiations in the following ways:

- *Pending requests posed, and alternatives, could be invalidated:* This could occur if a database change results in an earlier evaluated policy (to generate counter-requests) becoming more stringent. So even if the opposite negotiator were to respond with an

- affirmative offer, it would not satisfy the current policy. The result of negotiation would be consistent with current policy, since any requests satisfied after this change would necessarily have been satisfied in the absence of the change. The correctness of negotiation (dealt with in more depth in Chapter 8) requires that no requests be granted that would violate policy, and concurrent negotiations maintain that standard.
- *Potentially, more alternatives could be generated for pending requests received:* This could occur when database changes result in policies becoming less stringent. Therefore, more alternatives could potentially be generated if the counter-request generation procedure were to be re-run. In our implementation, we do not do this, because it would increase performance overhead and is prone to bottlenecks. As a result, a received request that would be satisfiable under the changed database conditions would not be satisfied, leading to a more conservative result. This result is less than ideal, yet maintains the correctness property defined above.

**Fixes and Tradeoffs:** The problems we encounter here have strong parallels with the problems in database transactions. Database scientists deal with these problems by enforcing the ACID properties (atomicity, consistency, isolation and durability) through *serializable* schedules. It is somewhat easier to enforce this on databases, with the only operations being reads and writes on individual data items. In our policy engine, the individual facts and rules can change dynamically, and rules and facts depend on each other. Keeping track of all the impacts produced by a change is exponential in complexity, thereby being prohibitively expensive in a real-world policy negotiation

protocol. Even if we chose to do this, arbitrary delays could be introduced in individual negotiation threads; a sphere could send an offer, and then keep waiting indefinitely because the recipient must finish negotiating with a third sphere before registering the offer in its database. This is practical for a typical database scheduler, but not in the environments where we would like to use negotiation.

We therefore just allow concurrent negotiations to proceed, with negotiations being allowed to change the contents of the database at any instant. The addition and removal operations are atomic (enforced using locks and synchronization), but different negotiation threads, in effect, race with each other. As mentioned above, the correctness property is maintained, but completeness (ideal result) is sacrificed for efficiency.

In practice, we leave it to individual spheres to attempt renegotiations even after the failure of the first attempt. In a typical Panoply deployment in our lab, a sphere attempts three joins before declaring failure. As we mentioned earlier, some negotiations may deliver more conservative results because they do not repeat the counter-request generation procedure upon change of database state; attempting multiple sequential renegotiations could rectify this problem.

#### **6.4.2 Dynamic Renegotiation**

As we saw in the above section, negotiation steps result in changes to the database and are also affected by dynamic changes. Negotiation decisions, such as the granting of requests, are made on the basis of invariants specified in policy rules. Consider an example where sphere A negotiates with sphere B for the privilege of becoming its



member. Sphere B has a policy that allows member spheres to enable sound alerts on their devices only if they are the lone member in the sphere. Sphere A is granted membership because there are no other constraints and it was the first supplicant. Sphere C now comes along asking to be a member in B. B imposes the “*turn off sound alerts*” policy, which C agrees to abide by. Now that C has become a member, B reviews its prior agreement made with A, and determines that A was not obligated to abide by the policy of “*no sound alerts permitted when there are two or more members.*” Therefore, it initiates a renegotiation with A for the membership privilege. If A is willing to abide by the policy, it remains a member; otherwise it is forced to leave (after all alternatives have been exhausted).

This example is implemented for general negotiation agreements (not just for sphere joins). Whenever a negotiation session terminates, earlier negotiated agreements are reviewed, and renegotiated if necessary. The procedure to infer the need for, and to perform, renegotiations is given below.

- *Preliminary*: Sphere S maintains a set of tuples  $T = \langle SID, OfferSet \rangle$ , each tuple mapping unique sphere IDs to the set of affirmative offers that have been made to it as part of the negotiation.
- At the conclusion of a negotiation, the list of offers granted as part of the agreement is saved as the set *CurrentOffers*.
- For each *SID* in *T*,  $RequestsForRenegotiation = OfferSet \cap CurrentOffers$  is computed. If *RequestsForRenegotiation* is *null*, there is no need to renegotiate with *SID*. Otherwise, these entries are removed from *OfferSet* and a negotiation thread is

started by simulating an initiation message from *SID* consisting of requests from the set *RequestsForRenegotiation*. Based on the results of this new negotiation, earlier granted offers may be revoked by retracting statements from the policy database.

We conclude with a caveat. By their very nature, some agreements cannot be renegotiated; the act of sending a file that was requested cannot be undone. But in the case of a sphere, the act of granting membership can and is revoked when change in context demands it.

## **6.5. External Helper Functions**

The controller module within the policy manager is an expandable framework which external functions can be plugged into. These functions perform non-logical tasks, such as returning objects, information, or simple yes/no answers. There are two places where such functions can be called. Referring to the flowcharts in Figures 8a and 8b, the formulation of an OFFER message, and the verification of an offer, involve external function calls. What is the purpose of such calls? They typically involve operations on software resources; in our framework, these are Java objects. Often, the functions at both places are complementary, as we will see through examples.

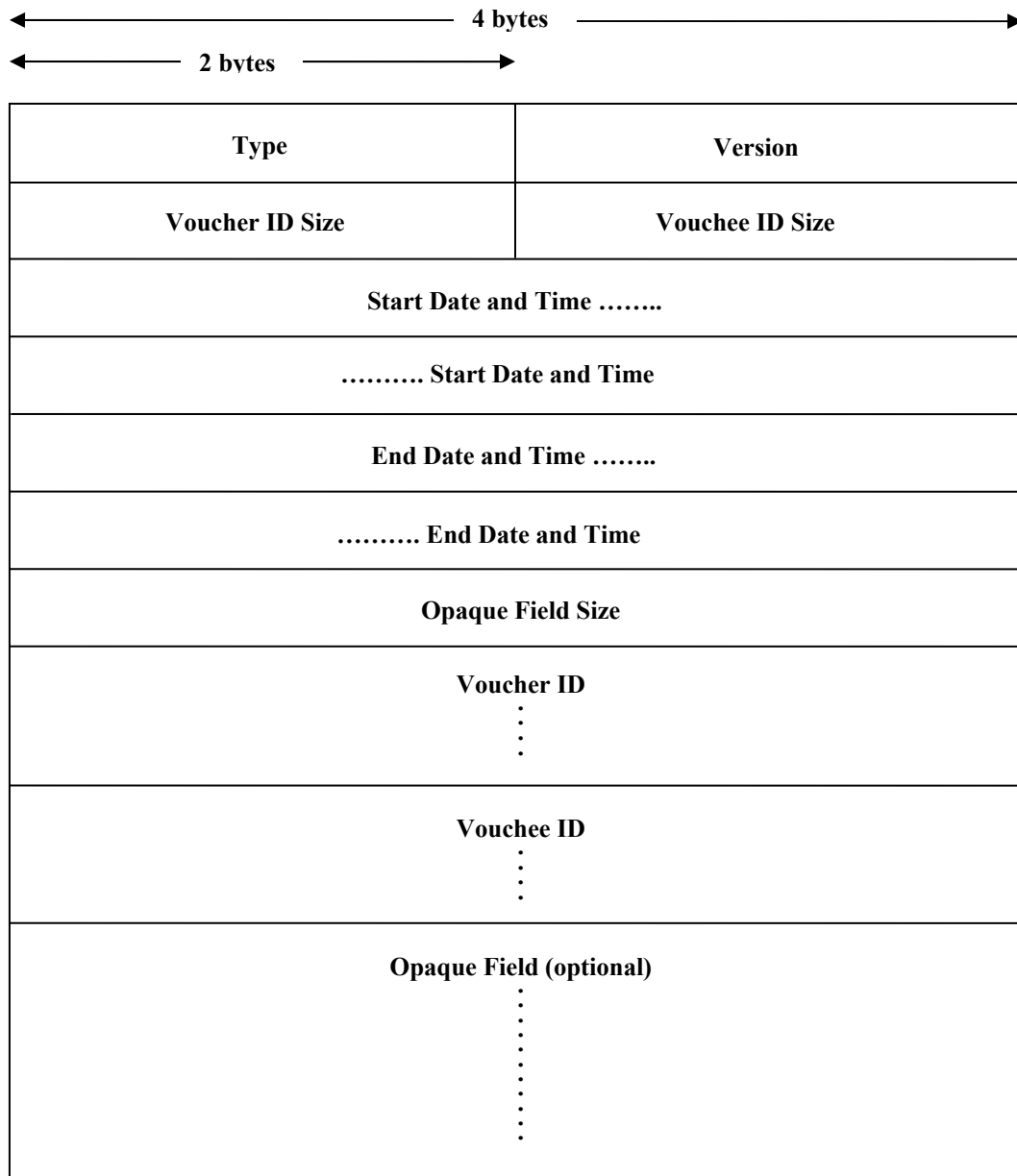
An OFFER message may contain affirmative and negative entries. For the latter, nothing else needs to be done before sending the message. But for an affirmative offer, which indicates the acceptance of a request, just sending a message indicating affirmation (or agreement) may not be enough. Some kind of object, or piece of data, may also need to be attached to the message. Such an object could represent something that was

explicitly requested, such as a file (e.g., media file, sensitive document), a credential (e.g., certificate, *voucher*), or a generic object (e.g., public key, agent code, Java class file). Alternatively, some objects could be implicitly requested, such as a credential indicating that the offering entity possesses valid authorization, or a proof of some kind (such as the result of running anti-virus software, which may indicate a clean bill of health). Exactly what the context is and what the appropriate object is that needs to be attached is determined by the external function, which takes the request and its support predicates (including the variable bindings) as input. The returned object is then attached to the outgoing OFFER message.

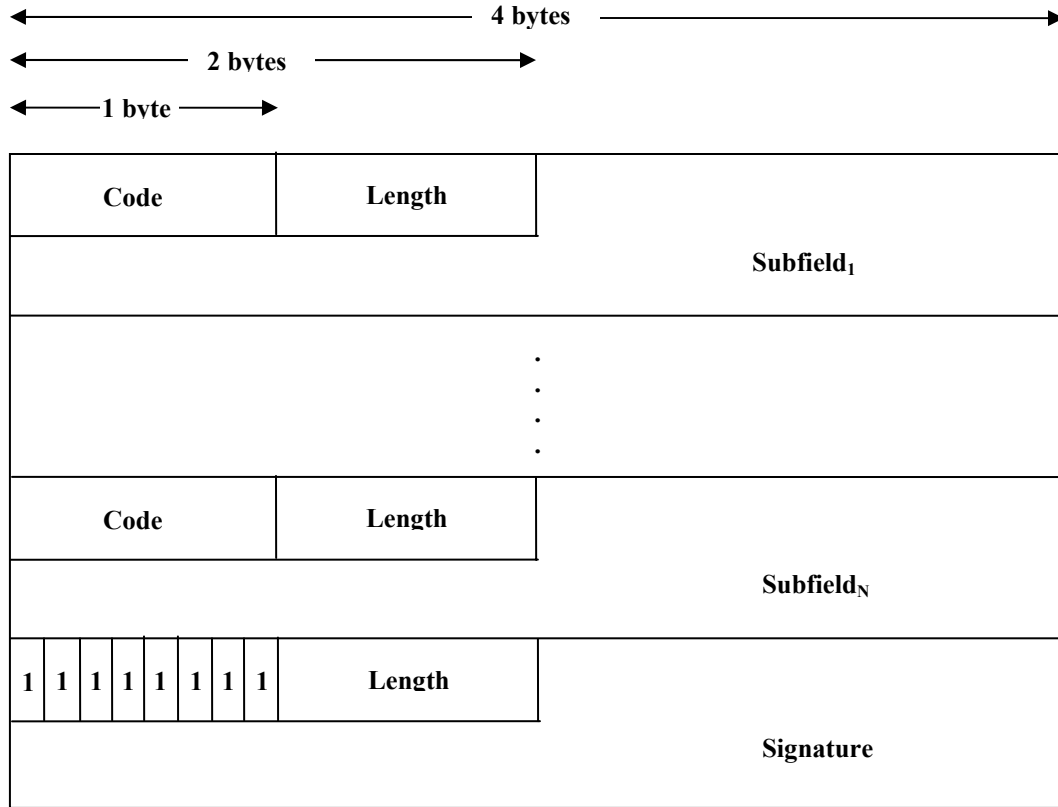
Correspondingly, when an OFFER message containing such attachments is received, the objects are extracted and verified using integrity checks. Such checks are carried out through external functions, which also take the request and support predicates as arguments. If the sent attachment has been encrypted or signed, the received attachment is decrypted or verified through a complementary operation.

*Termination:* Helper functions are not considered in our discussion of negotiation protocol and termination semantics (Chapter 8). Therefore, a helper function that triggers an exception or goes into an infinite loop will result in an erroneous or non-terminating negotiation. Care must be taken before such functions are used in a negotiation.

**Vouchers:** For application-layer security in Panoply, we designed and used a generic multipurpose cryptographic credential, similar to SPKI [RFC2693] called a *voucher* [Eustice2008a] (see Figures 13a and 13b below).



**Figure 13a.** Voucher Data Structure



**Figure 13b.** Opaque Field Data Structure

We wanted this credential to have as flexible a data structure as possible so that it could be used to specify arbitrary properties and privileges of both the voucher-issuer and the voucher-owner. Privileges could include delegation rights, a valuable and useful concept in ubiquitous computing systems. A voucher could be signed and verified using RSA public key cryptography, though that is not mandatory. A voucher has an *opaque* field that contains multiple entries, each representing a particular property or privilege. For example, we encode the current location of the *vouchee* in a *location-certifying voucher* in the form of a policy statement, and its group affiliation in a *social-group-*

*certifying voucher*. The voucher data structure can be subclassed (we do this in two ways: a Location Voucher and a Social Voucher) to make it certify more specific properties.

Many of our experiments with negotiation involve generation of signed vouchers, which are then verified using public keys at the receiving end. As vouchers also specify start and expire times, we use predicates of different lengths to differentiate between general vouchers and vouchers that are requested and granted for a particular duration. Vouchers are stored in a separate database for each sphere, and can either be dynamically generated using a helper function or retrieved from the database, as the scenario may require. When a voucher is received, the possession of the voucher and properties it certifies are asserted in the policy database.

We have also used helper functions that retrieve and attach appropriate X.509 certificates, audio files and public key data structures in our applications.

## **6.6. Protocol Reliability and Fault Tolerance**

The state machine illustrated in Figure 7 does not handle reliability issues, or system and network faults. It does indicate a transition to the STOP state upon a timeout, which is a trivial (though efficient) form of fault tolerance. Given that the policy manager supports multiple concurrent negotiation threads, and inter-sphere interactions depend on negotiation, we need to have a better and cleaner notion of protocol reliability, and gear it towards avoiding race conditions and conflicting negotiation sessions.

There are a number of scenarios that could result in faulty or unreliable behavior if the policy manager were to implement just the state machine in Figure 7. This is not

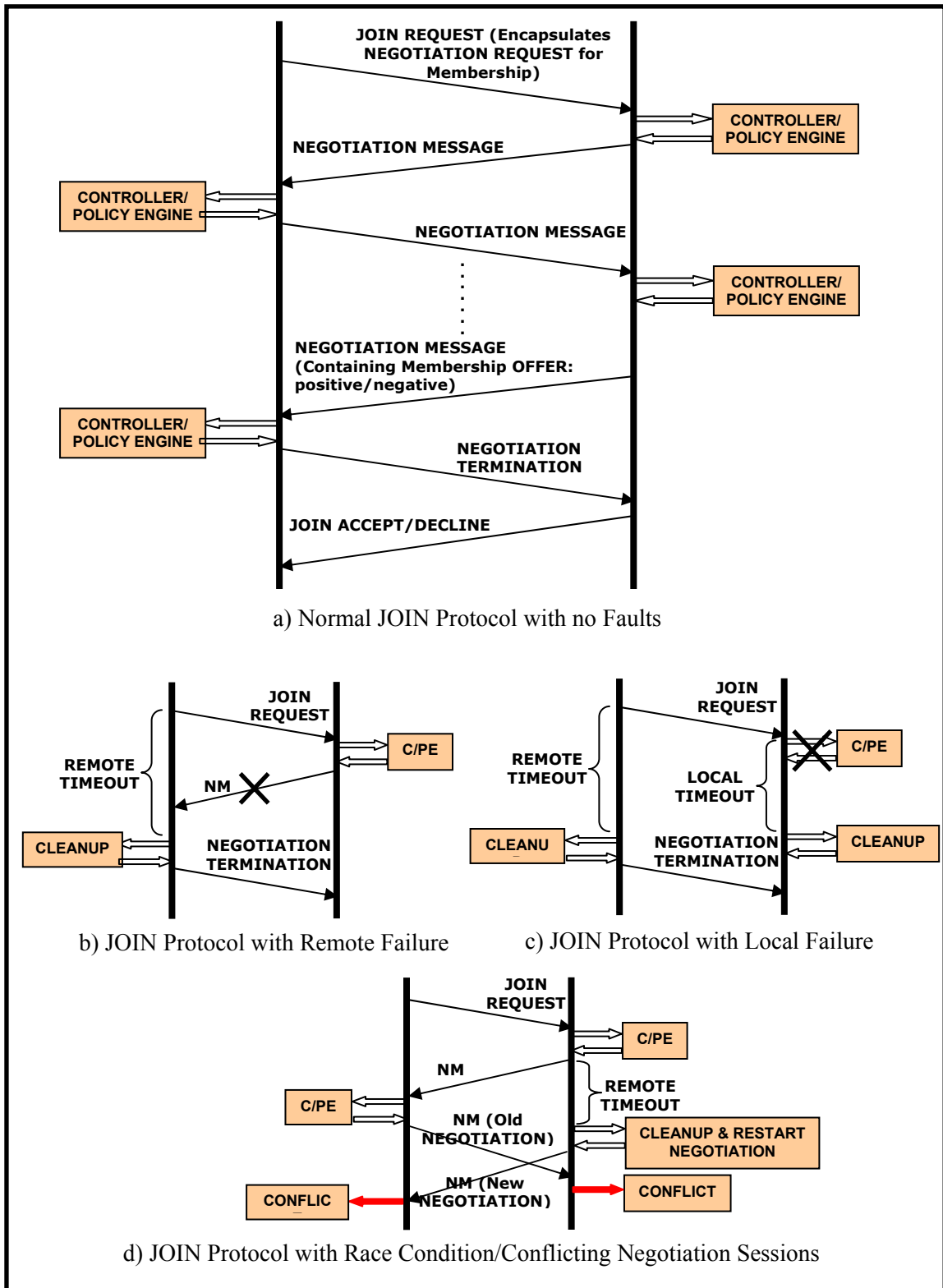
only because of real failures in local computation or on network links, but also because the policy manager's actions are transparent to the rest of the sphere manager. Since the policy manager was supposed to mediate interactions, it was reengineered to take into account these systems faults as well as the responses triggered by other sphere modules. We sought to make the negotiation protocol resilient to three basic faults:

1. *Remote engine/network failures*: When a negotiation thread is in the *expect* state, it is waiting for a message from the policy manager at the other end of the communication channel. It could end up waiting indefinitely, either because of some problem in the remote policy engine (event was not sent) or in the event processing framework (event was dropped in transit). The latter scenario includes basic network link failures, which are uncommon in modern networks but may occur when a mobile device lies at the periphery of a wireless LAN's range.
2. *Local engine failures/delays*: There is no bound on the time a policy engine might take to process and send a response to a negotiation message. It could vary arbitrarily with the number of entries in the message, helper function run times, and the size of the policy database. Alternatively, there might be some latent bug in the Prolog subsystem (as we have found in practice) which simply causes the policy engine to get stuck or go off into exception mode. In practice, though, one cannot spend an unlimited amount of time on negotiation, so a non-response (or a timeout) from the local engine is treated as a fault.
3. *Race conditions and conflicting negotiation sessions*: Unlike the above two classes of failures, which can occur in any network protocol, race conditions occur because of

the nature of the role the policy manager plays in building sphere relationships. We allow multiple concurrent negotiations, but only one per negotiation pair at any given instant. This is enforced to prevent multiple redundant requests, and to a lesser extent, to avoid potential policy conflicts and loops, which could result from the policy database being modified independently by each negotiation thread. Race conditions may occur because of the decoupling of the sphere JOIN mechanism from the negotiation protocol. A sphere may time out when trying to join another, and be programmed to retry the procedure. But spheres are not obliged to have identical timeout values, and have no time synchronization. So the sphere that has timed out might trigger a new join, and consequently a new negotiation, while the other sphere still thinks that the old negotiation is valid. This could cause multiple such negotiations and joins going back and forth, with the request lists on both sides building up; clearly this is an undesirable and unpredictable situation.

The three types of faults are illustrated in Figure 14. All faults are illustrated in the context of a policy-manager-mediated JOIN protocol.





**Figure 14.** Faults and Recovery in a Negotiation Protocol Instance

### **Fixing the Problems:**

We use two techniques to recover from these faults, which have been commonly used in network protocols and in distributed systems:

1. *Timeouts*: Each negotiation/JOIN thread tracks the time taken both by the local policy engine to return a reply and to receive a message from the remote sphere. If either of them exceeds a pre-decided timeout value, the negotiation instance is terminated. This value is a core property of the sphere. Upon termination, the request lists maintained by the controller are purged. Since negotiations can have intermediate requests and offers, it is likely that the policy database would have been modified by the partial negotiation. We do not make any effort to attempt a rollback of negotiation at this stage. First, such rollback would impact performance, and second, it produces no observable benefit. This is because negotiations are not equivalent to atomic database transactions, where updates are made final (through commit protocols) only at the end. Intermediate offers to requests must necessarily result in some tangible change in state; otherwise a negotiator has no incentive to proceed. For example, a file or a credential sent to the other side cannot be un-sent. On the other hand, acceptance of certain policy obligations, and promised access rights, could be reneged upon. We do not perform a rollback for these types of offers upon negotiation failure (in case a negotiation is attempted again shortly), but do so when the spheres explicitly break their relationship.

The timeout values must be chosen with care so as to maximize overall efficiency. The local timeout should be less than the remote timeout, simply because

of the extra latency of event transmission and communication. The local timeout should also be more than the default timeout value of the communication channel (in our case, a Java socket). Also, these timeout values should not be too low, which might result in premature terminations and unnecessary negotiation sessions.

2. *Timestamps*: Timeouts avoid unnecessary delays and a waste of resources upon failure, but could cause race conditions (see Figure 14) when the sphere manager at one end decides that it needs to restart a JOIN while an older negotiation is still going on. We cannot avoid the creation of multiple conflicting negotiation threads, but we can enforce a policy whereby only one thread among a pair of negotiators will be considered legitimate at any given instant. Timestamps, which indicate the instants at which negotiation threads are started, act as unique identifiers for these threads. They are set by the initiator, thereby synchronizing between the two entities. Each negotiation message within a thread is stamped with this timestamp value, indicating which thread it belongs to. A table indicates the current legitimate timestamp value for a given negotiator. A negotiation message that arrives carrying an older timestamp is dropped without consideration. Conversely, if a message carrying a newer timestamp value arrives, the currently running negotiation thread is terminated and a new thread created to process the received message.

Could security problems arise because we do not attempt rollbacks of partial negotiations? It is unlikely, except in cases where an entity knows about the other side's policies. This entity could then engineer the negotiation in such a way that intermediate offers are made, which benefit it in some way, and then simply terminate the negotiation.

Also, some actions are reversible and some permissions are revocable, but all are not; we will discuss the consequences of this further in Chapter 8. But as our entire negotiation model is based on the presumption that policies are local and private, these kinds of security breaches are unlikely to occur as long as that property holds.

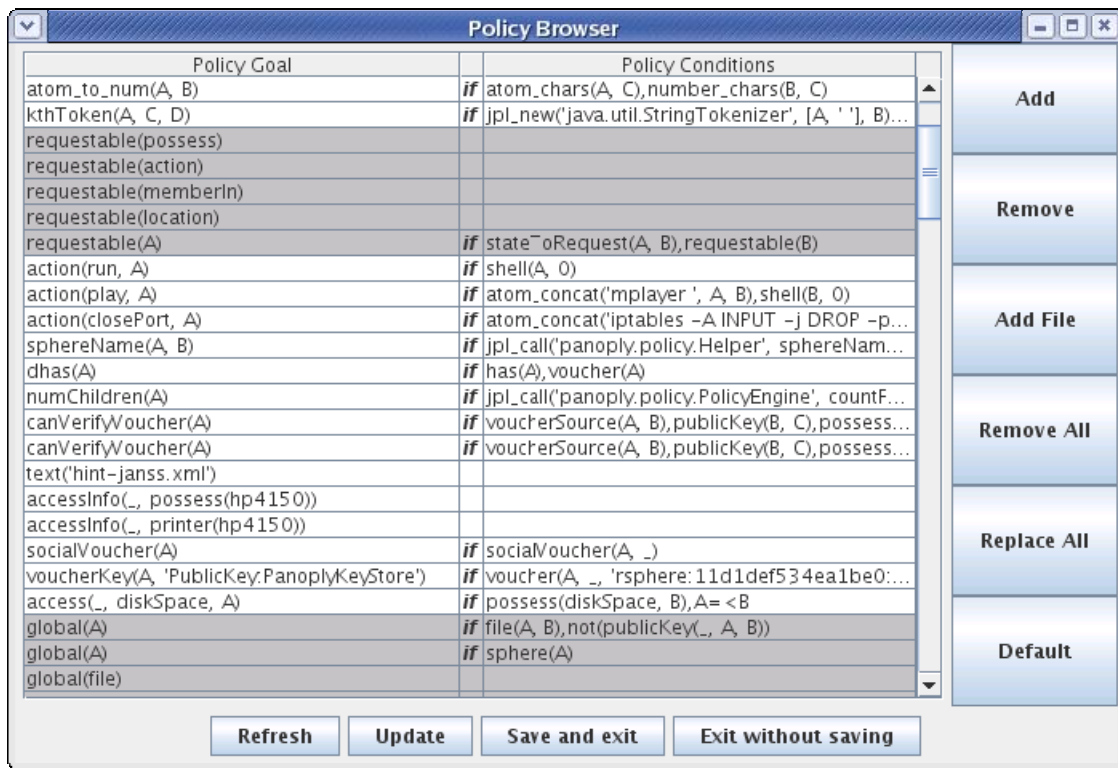
## **6.7. Visualization and Administration Tools**

The Panoply middleware provides primitives for the design and implementation of applications that can make use of the physical and social context provided by the hosting sphere without necessarily being aware of it. Because these applications perform their activities using events, they can be easily coupled to, or decoupled from, an individual sphere. Any subsequent changes in behavior can be made easily by adjusting the set of events the application is interested in. Such a loosely coupled model also enabled us to build tools that users and system administrators would find useful when interacting with the policy manager of a sphere. Not only would this allow us to change the functionality of these applications without having to modify the core behavior of the policy manager, but this would also provide better performance, as the activities of these applications lie outside the policy manager's processing loop.

### **6.7.1 Policy Browser**

A real-world policy manager is not very useful if it does not provide an option to observe and manipulate its entries (policies in the database) during runtime. This is especially useful to a system administrator who would like to see a snapshot of the policy database

at a given instant, and the changes that have occurred in sphere state. Both the administrator and an ordinary user may want to change individual policy rules during runtime without having to shut down and restart the entire Panoply subsystem. The *policy browser* application, a snapshot of which can be seen in Figure 15, provides a window to the policy engine that is not unlike a *database view*. As we can see, the individual policy rules are listed in the rows of a table. The antecedent of every clause is displayed on the left column and the consequents on the right column. The browser offers options to add and remove policies, modify entries in individual cells and update the database, and import policies from an external file, among others.



**Figure 15.** Policy Browser Snapshot

By default, the browser displays both system (low level) and user policies. We can also prevent core system policies from being modified by ordinary users (indicated

by the grey rows in the table in Figure 15). Though not implemented, manipulation of the database (or particular sections of it) could be controlled using a mechanism similar to the one we use to control resource access (see Section 6.3.2); the browser itself could allow the user to perform an action but the policy manager would then negotiate with the user for the privilege of changing database contents.

### **6.7.2 Negotiation Timeline**

A user can observe the progress of a negotiation on a dynamically updated protocol timeline diagram, which indicates the kinds of messages and their contents, as well as the time taken for each step. This tool does not provide any inherent value of the kind a policy browser offers, but is very useful as a way of tracking and visualizing how a negotiation is proceeding, in a way that is similar to how one would observe the activity of a network being simulated using NS2 through a Tcl/Tk GUI. It is also useful for demonstration purposes. An example of such a timeline is illustrated in Figure 16. A user could move his mouse over the individual negotiation message boxes and observe the contents of the messages, as indicated in the figure.

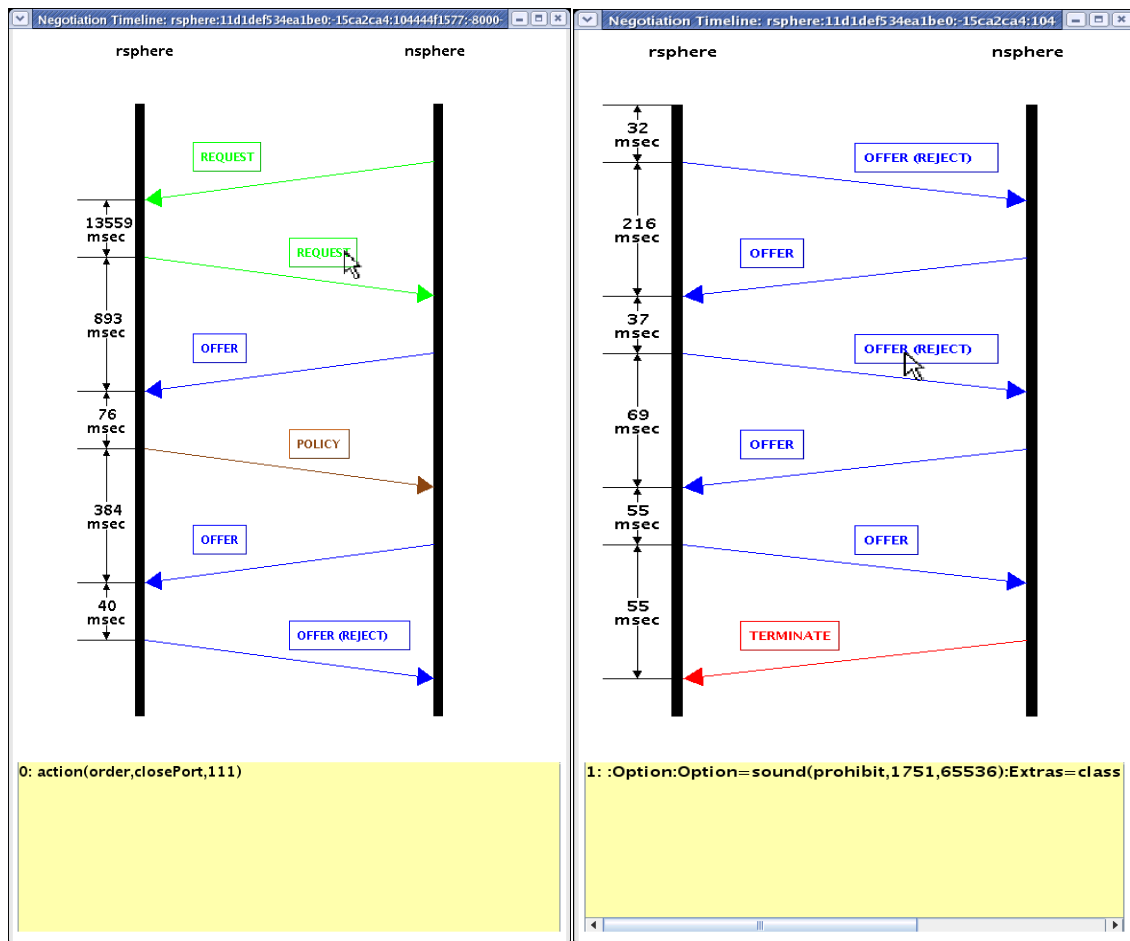


Figure 16. Negotiation Timeline Example

## Chapter 7

### Demonstrative Applications

The Panoply policy manager was exercised in various application scenarios that either occur in the real world or emulate those that occur in the real world. These applications were designed and implemented under Panoply, and they depend on Panoply's event publish/subscribe framework for communication. Sphere join (one sphere attempting to be a member of another), which is the most common type of inter-sphere interaction, is carried out as a negotiation, and we show how this works in the context of our flagship Panoply applications. We describe a range of applications scenarios and variations that result in different protocols and outcomes. These scenarios exercise all aspects of the policy manager, namely negotiation, policy filtering, and event-action triggers.

**Policy Complexity:** Before we proceed, we should mention that a number of policies that will be described in this chapter are quite hard to read and understand for someone who is not familiar with logic, Prolog, and more specifically, SWI-Prolog. Our goal was to design a policy language that primarily served the purposes of negotiation, and usability was not a top priority. Still, usability needs can be met without changing or replacing our current language in its entirety. We will discuss how this can be achieved in Section 11.4.3. More specifics on understanding the syntax and semantics can be found in both Chapter 4 and Appendix A.



## 7.1. Negotiation in Prominent Panoply Applications

In this section we describe the most prominent uses of the policy manager in Panoply and its applications. First, we describe some simple negotiations that occur as part of sphere join operations, where the goal is simply the following: a supplicant sphere wishes to be a member of a host sphere (representing a home or a lab network), and the latter has constraints and standards it would like to impose upon prospective members. Following that, we describe the role played by the policy manager in two flagship Panoply applications: the *Interactive Narrative* and the *Smart Party*.

### 7.1.1 Membership in a Home/Lab Environment (Sphere Joins)

Typical negotiations for membership are initiated by a supplicant sphere. In response, the host sphere examines its policies governing member eligibility; it either makes a decision to admit or reject the supplicant, or continue the negotiation by posing counter-requests. Our scenarios typically consist of the following:

- A computer (H) in the laboratory/home running a sphere that represents and manages the lab/home. We used a variety of machines for this purpose, ranging from desktop computers to IBM Thinkpad laptops and Sony MicroPCs. The sphere serves as a container for various pieces of information, including what kinds of displays and printers it controls, how much bandwidth and disk space is available, what credentials it possesses or can create for certification purposes, and policies that govern these.
- A prospective client sphere (C) hosted on a mobile device like a laptop or a MicroPC that requires membership within the lab/home. It serves primarily as a container for

credentials and for information like its current location and team affiliation that may have to be released for the success of a negotiation.

- A regular user, or a guest, carries his/her computer and walks into the lab/home. The mobile device C detects the host environment H (as represented by the sphere), connects to the appropriate wireless network and initiates a Sphere Join, and a negotiation ensues. Such detection, connection, and initiation could be done manually as well, but the Panoply framework provides an automated mechanism that uses network information embedded in the vouchers, combined with localization maps for this purpose. The full procedure is described in Eustice's PhD thesis [Eustice2008b].

**Open environment:** H could be a completely open environment, having a tautological membership policy:

```
member(Sphere) :- true.
```

C would become a member, or a child sphere of H, in a three-step negotiation:

REQUEST<member>

→ OFFER<affirmative>

→ TERMINATION

An event-action-trigger policy is used to register the requests that must be posed in C's policy database. For example, C would want to be a member of a sphere if it currently is not a member of one, and is newly associated with a new location. A location update event results in the registering of a new location in the policy database, which in turn results in the assertion of a membership request through the following policy rule:

```

update :- (candidateInSphere(Sphere) ->
          (not(request(member,r)) -> assert(request(member,r)))) .

```

**Negotiation with credentials:** A voucher certifying presence in the vicinity of the host sphere could also be part of the transaction. Such a voucher could either be explicitly requested by the client, or the host may grant one in a gratuitous manner because it has a policy of granting one to any sphere that gains membership privileges.

In the former case, the negotiation would be initiated by the REQUEST message containing the entries `<member; (possess(LV),locationVoucher(LV))>`. H has an access policy of the form: `access(Sphere,LV) :- locationVoucher(LV)`. The predicate `locationVoucher(LV)` has a helper function associated with it that creates and digitally signs a voucher and attaches it to the OFFER message.

In the latter case, H has a policy rule of the form:

```

update :- ((negiator(Thr,Sphere),member(Sphere)) ->
          ((locationVoucher(LV),(LV='voucher_location=LASR'),
            (not(possess(S,LV))),
            not(offer((possess(V),locationVoucher(V)),nil))) ->
            (assert(offer((possess(V),locationVoucher(V)),nil)))) .

```

This policy has the effect of obligating H to offer a location voucher to a member sphere that has not already been granted one. The negotiation again involves three steps:

```

REQUEST<member; [possess(LV),locationVoucher(LV)]>
→ OFFER<affirmative; {LV='voucher_Object'}>
→ TERMINATION

```

Other requests for information and resources could also be requested using similar policies. The following policy statement expresses the need to find out whether the host owns a color printer:

```
update :- (candidateInSphere(Sphere) ->
    (not(request((possess(X),printer(X),type(X,color)),q)) ->
        assert(request((possess(X),printer(X),type(X,color)),q))))).
```

### **7.1.2 Membership and Interaction in an Interactive Narrative**

The Interactive Narrative [Eustice2007] is a location- and team-aware application that was designed and deployed throughout the UCLA campus. It was built on the Panoply platform, and the interacting parties were modeled and implemented on spheres. In a nutshell, the application allows people playing the role of characters in a story to visit designated locations in the campus through an unscripted trajectory, uncover more of the story, find new clues, and perform actions given available choices at each location. Actions and location visits drive the narrative forward. Participants are represented in this virtual narrative by their mobile computers, either laptops or Sony MicroPCs or OQOs. The content is already available and associated with each location; dynamism arises from the random nature of inter-sphere associations. These associations occur through sphere joins: specifically, characters' device spheres joining and negotiating with location and team spheres. The negotiations are constrained by the policies of the overall game, and of each location.

To join a team (social) sphere, a character's device negotiates with it typically in the following way:

- The device asks to join the sphere by sending a REQUEST message containing a member entry:

REQUEST<member>

- The team sphere has a membership policy like the following:

```
member(S) :-
    candidateSphere(S),
    teamMember(S), numChildren(N), maxChildren(M), N <= M,
    possess(S,V), socialVoucher(V,G), localSphereID(G),
    location(S,L), permissibleLocation(L).
```

This policy allows granting of membership to a sphere S that is already a candidate (an intermediate stage in a sphere join procedure whereby a prospective client sphere is partially associated) and if S's ID identifies it as being a team member, if the current team size does not exceed the maximum supportable, if it can prove its affiliation by providing a team voucher formerly acquired, and if its current location is known.

- If the team size constraints are met, the social sphere sends a counter request with two entries: request for a social voucher as proof, and an inquiry of the device's location.

→ REQUEST<possess(LV),socialVoucher(LV); location(L)>

- The device has open access policies for these requests:

```
access(S,V) :- socialVoucher(V,R), negotiator(Thr,R).
accessInfo(S,location(L)).
```

These state that social vouchers can be revealed to whoever issued them, and location information can be released to anyone.

- The device returns affirmative offers containing its location information and its social voucher.

→ OFFER<LV='voucher\_Object'; L='Location\_A'>

- The team sphere examines the voucher cryptographically, and matches the device's location with the predicate `permissibleLocation(L)`. If they check, the membership request is granted.
- The team sphere also has an event-action policy of the following form:

```
update :- ((negotiator(Th,S),member(S)) ->
    ((locationVoucher(LV), (LV='voucher_location=LASR'),
    (not(possess(S,LV))),
    not(offer((possess(V),locationVoucher(V)),nil))) ->
    (assert(offer((possess(V),locationVoucher(V)),nil))))).
```

When the decision to make an affirmative offer to the membership request registers in the policy database, the database is updated with a statement which indicates that a location voucher must be granted to the device visiting the location. Though not explicitly requested, this voucher is gratuitously offered to the device. The offers are made, and the negotiation terminates.

→ OFFER<affirmative; {LV='voucher\_Object'}>

→ TERMINATION

A device that has already joined a social sphere wanders around (carried by its owner) and associates with different locations, getting content appropriate to its immediate context. The application design allows each location to have its own location sphere, or have the social sphere host multiple virtual location spheres, and act as a narrative content provider. For ease of implementation, we followed the latter approach. Various application-independent tasks are performed through policy management when

devices dynamically associate with locations. First, obtaining content is a process of negotiation. The social sphere's state manager gets requests in the form of <STATE, REQUEST> events (the *type* field of the event is set to STATE and the *subtype* field is set to REQUEST) that are filtered by the policy manager, which could initiate a negotiation with the device for the privilege of providing content. The low-level policy rule that is evaluated whenever an event is diverted to the policy manager is given below.

```
action(pass,event,EID) :-
    jpl_call('panoply.policy.EventPolicyMediator','currentEvent',[
        EID],E),
    eventType(E,T), eventSubType(E,ST), eventUserType(E,UT),
    eventSource(E,So), condition(So,T,ST,UT).
```

The *condition* predicate represents the customizable portion of the above policy rule; example policies that govern event filtering (with which we experimented) are given below.

```
condition(So,T,ST,UT) :-
    T='STATE',
    ST='REQUEST',
    obey(So,pass).

obey(So,pass) :- not(player(So,'William')),
    location(So,L), not(L='InvertedFountain').

obey(So,pass) :-
    player(So,'William'), location(So,'InvertedFountain'),
    numLocationsVisited(So,N), N >= 4.

obey(So,pass) :-
    player(So,'William'), location(So,'InvertedFountain'),
```

```

numLocationsVisited(So,N), N < 4,
action(So,order,changeCharacter,'Amanda').

```

The '*obey*' policies indicate the restrictions on characters getting content at particular locations (equivalently, joining those location spheres). The first indicates that any character not assuming the persona “William” can join locations other than the “InvertedFountain.” Only player “William,” if he has already visited four or more locations, may join the location “InvertedFountain.” If he has visited fewer than four locations, he must change his persona to “Amanda” in order to access content at the “InvertedFountain” location. If the predicate `numLocationsVisited(So,N), N < 4` evaluates to *true*, a negotiation is initiated and a counter-request is sent to the device hosting the character “William,” as follows:

```
REQUEST< action(So,order,changeCharacter,'Amanda)>
```

If the device has a policy “`obey(S,changeCharacter) :- true`”, it will send an acceptance offer indicating that it will change its persona to “Amanda.”

```
→ OFFER<affirmative>
```

```
→ TERMINATE
```

The changing of the persona occurs through the following low-level policy:

```

action(changeCharacter,CharacterName) :-
    string_concat('LMPersona:',CharacterName,S),
    send_event('APPLICATION','SET',S).

```

This triggers the sending of a Panoply application event to the application running the Interactive Narrative at the client, resulting in a change of persona.

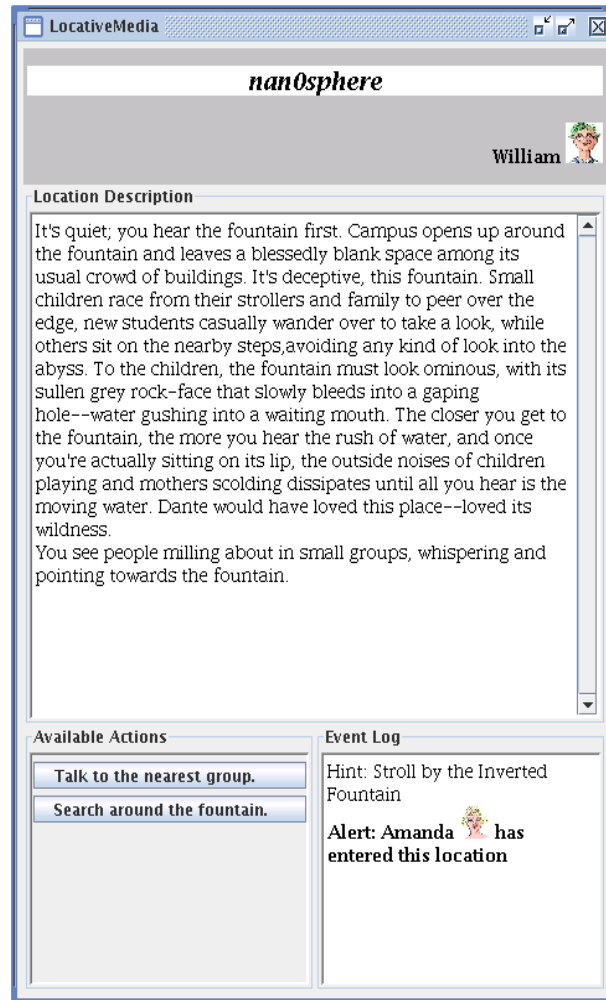


Thus, we can see how negotiations can be used to make character joins and content access procedures dynamic and flexible, as the constraints are independent of the story outline and can be modified during runtime. The predicates like *player* and *numLocationsVisited* are updated whenever a state change is detected through events received by the social and location spheres' policy managers.

An example of an event-action trigger can be illustrated through the following policies:

```
nextLoc('LASR','BoelterGarden').
nextLoc('BoelterGarden','BombShelter').

update :-
    ((player(S,P),location(S,'LASR'),numLocationsVisited(S,N),N>4,
    nextLoc('LASR',Loc),not(visited(S,Loc)),
    atom_concat('Head off immediately to ',Loc,Hint)) ->
    jpl_call('panoply.policy.Helper','execAction',['Hint',S,Hint],
    @true)).
```



**Figure 17.** A Snapshot of the Interactive Narrative Client GUI (Note the Policy-Driven Hint Displayed in the Bottom Right-Hand Panel)

The `update` has the following effect: if a player visits the location “LASR,” and has visited more than four locations, but not the next location, which is “BoelterGarden,” it is sent a hint encouraging it to visit “BoelterGarden” through a Helper function that sends a suitable event. The hint is displayed on the client GUI (see Figure 17).

### 7.1.3 Membership and Interaction in a Smart Party

Another application that was successfully designed and implemented using the Panoply middleware was the Smart Party [Eustice2008a]. The scenario is identical to the one outlined in Chapter 1. The party can be conducted within any bounded geographical location of arbitrary size. For demonstration purposes, we hosted one within our laboratory (3564 Boelter Hall at UCLA) with three rooms that are distinguishable using an 802.11 signal strength triangulation-based localization map. The application itself contains a social sphere consisting of all guests to the party, three location spheres that represent the three rooms in which guests congregate and which contain speakers. Guests enter the party environment with personal mobile devices running Linux (either IBM Thinkpad T42 laptops or Sony Vaio UX2800 MicroPCs). Each device contains audio files with metadata, and also contains that user's musical preferences (these are weighted orderings of songs, genres and artists).

Policy management plays an important role in various aspects of the party. Joining the party after discovering it requires negotiation for membership. The sphere managing the overall party (referred to as the *Smart Party Master*) has already granted cryptographic vouchers to the spheres running on devices owned by the people on the guest list prior to the party. These vouchers have embedded information associating the party with certain wireless networks, and the Panoply framework enables discovery and association with those networks once a guest's device enters the vicinity of the party. As described earlier in this section, being associated with a sphere automatically registers a request for membership within the guest device's policy database, and a sphere join

triggers negotiation. Negotiation then takes place as part of the join procedure. A typical membership policy is similar to the membership policies in scenarios where a device joins a home sphere, an office sphere, or a team sphere in the Interactive Narrative. A typical entry policy would be the following:

```
member(S) :- candidateSphere(S), possess(S,V),
             socialVoucher(V,G), localSphereID(G).
```

This policy indicates that the guest device must possess a valid social voucher indicating an invitation to the party; this voucher would have been granted by the party sphere that grants entry permission through a variety of mechanisms, including email and sphere associations. When a guest first enters the vicinity and associates with the party sphere, the latter requests production of the social voucher, and allows the join process to complete upon successful verification of the voucher. If the guest wanders away temporarily and tries to rejoin the party, his device will not be prompted to show the voucher because the party sphere's policy database has registered the knowledge that the device possesses a valid voucher.

The door of our laboratory can be controlled remotely using OPEN/CLOSE events to a Panoply application that directly interfaces with the door through the serial port of the computer it is connected to [Eustice2008b]. A guest's device can negotiate upon entry for the right to be able to open the door automatically from the outside. In this case, the request message will contain the following entries: `member,`  
`(action(permission,open,D),door(D))`. Example party sphere's policies governing who has the right to open the door are as follows:

```

obey(S,open,D) :- door(D), possess(S,V), socialVoucher(V,G),
    localSphereID(G), numChildren(N), N<=5.
obey(S,open,D) :- door(D), possess(S,V), socialVoucher(V,G),
    localSphereID(G), numChildren(N), N>5,
    runApp(S,prohibit,'panoply.apps.smartparty.SmartPartyUserDeviceApplication')

```

These policies enforce the constraint that only a guest device with a valid social voucher will be allowed to control the door as long as there are five or fewer guests currently associated with the party. If there are more guests, additional guests will be allowed to control the door as long as they are willing to refrain from running the *SmartPartyUserDeviceApplication* application, thereby allowing them to join the party but not participate in suggesting and offering songs. The guest device enforces this prohibition through the following low-level policy in its database:

```

runApp(S, prohibit,
'panoply.apps.smartparty.SmartPartyUserDeviceApplication'):-
    obey(S,runApp), assert(runApp(prohibit,App)),
    assert(prohibitedSoUT(L,'MediaSearch')),
    assert(prohibitedSoUT(L,'MediaSuggestion')).

```

This ensures that *MediaSearch* and *MediaSuggestion* events will be blocked, thereby rendering the *SmartPartyUserDeviceApplication* ineffective.

The party sphere on the other hand asserts the following in its policy database:

```

condition(guestDeviceSphere,T,ST,UT) :-
    UT='DoorControl', T='APPLICATION'.

```

Here, *guestDeviceSphere* is the sphere name of the particular guest device that has been granted permission to control the door. Whenever a door control event originating from that device is received by the DoorController application run by the party sphere, it is allowed to pass because the query `condition(S,T,ST,UT)` evaluates to *true* when `S=guestDeviceSphere`, `T='APPLICATION'`, and `UT='DoorControl'`. Thus the guest device is authorized to control the door.

Using the rules stated below, the party sphere can enforce the following policy on the door: “*the door must not be open for longer than ‘t’ seconds*”.

```
% Policy governing duration of door opening
maxOpenDuration(door3564,'5').

openDuration(D,T) :-
    maxOpenDuration(D,T).

% Policy that triggers event for closing of the door
update :-
    ((openDuration(X,T), door(X), openD(X), not(doorOpen(X))) ->
    (assert(doorOpen(X)), retract(openD(X)),
    send_event('APPLICATION','STOP','DoorControl',T))).
```

These policies have the effect of setting the maximum permissible open time for the door to be five seconds. When a door is opened, the predicate `openD(door3564)` is asserted in the database, and the resulting event-action policy indicated above evaluates the *send\_event* predicate with the argument five. Consequently, the low-level policy below is evaluated; this rule calls a helper function that schedules a door closing event to be sent to the DoorController application after five seconds.

```

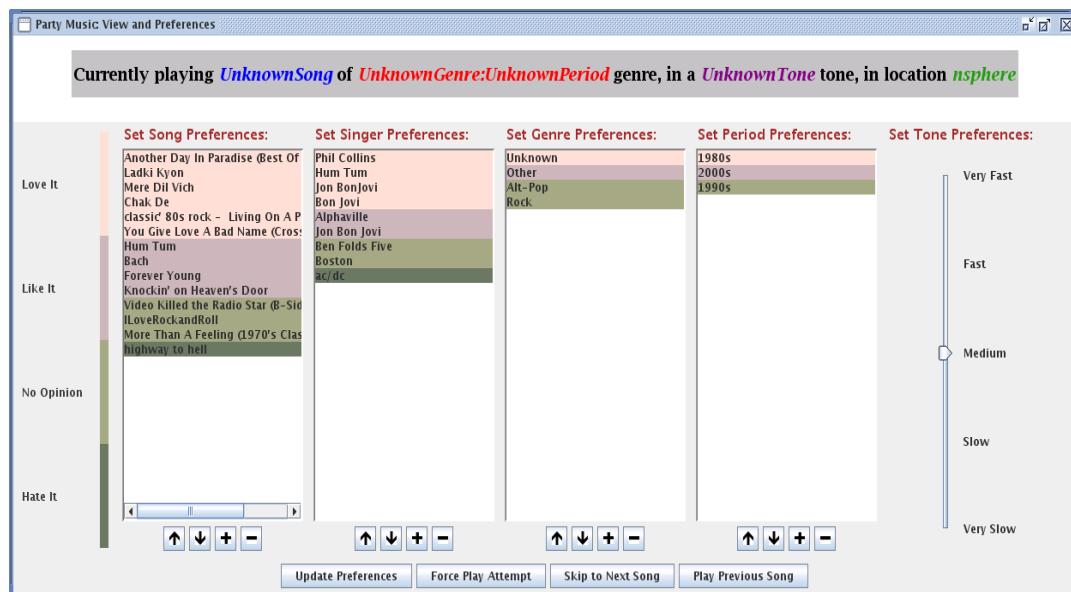
send_event(EventType,EventSubType,UserType,Time) :-
    string_concat(EventType,':',S1),
    string_concat(S1,EventSubType,S2),
    string_concat(S2,':',S3),
    string_concat(S3,UserType,S), string_to_atom(S,E),
    jpl_call('panoply.policy.Helper','execAction',['Event',E,Time]
    ,true).

```

Once the guest device has associated with the party, it obtains a localization map from the party sphere. Using this map, it can localize to one of the designated rooms within the party (in our implementation, we have three cubicles within our office that our localization map can distinguish). Each room has its own sphere associated with it, and the guest device negotiates with a location sphere in order to be able to join it, and suggest and share music files. The negotiation is initiated by the guest device requesting membership, and the location sphere asks for a valid social voucher in return. The guest device produces the voucher and completes the join successfully. The membership policies could also be framed in such a way that no guests are allowed to associate if the room size exceeds a certain capacity.

In our default implementation, the voucher carried by the guest device lets it know that it must start the *SmartPartyUserDeviceApplication (SPUDA)* application in order to be able to suggest and share music. This can alternatively be done as part of the negotiation process. In return for membership privileges, the party environment can oblige the guest device to run the *SPUDA* application using an *action* predicate (in the same way a location sphere was able to oblige a change of persona on the character's

device in the Interactive Narrative scenario). The counter-request therefore contains both a request for production of a voucher (a possession predicate) and a request to run the *SPUDA* (an action predicate). The guest device's policy allows release of the voucher to anyone who asks for it, and has an *obey* policy that authorizes it to obey any action request to run a Smart Party application. The negotiation therefore could complete successfully with the guest device starting the *SPUDA*.



**Figure 18.** Smart Party GUI: Enables Manual Playlist Control

During the party, each guest device can run a GuestPreferencesGUI application, an instance of which is indicated in Figure 18 above. It allows one to control the music playlist within a room (moving the playlist marker forward or back). When a guest tries to manipulate the playlist, the events are filtered through the policy engine, which checks to see if the manipulator possesses a valid voucher indicating 'host' privileges (indicated as a predicate within the voucher). If the query fails, a negotiation could result, and if the



guest device does possess a valid voucher indicating host privileges, the playlist manipulation events are permitted to go through.

## 7.2. A Conference Room Environment

We describe here how a negotiation along the lines of the conference room scenario can be performed using our framework. The device owned by the conference attendee and the conference environment are both modeled as Panoply spheres, and both run on separate computers. Our setup is not a real conference room, but it accurately simulates the set of devices and resources that comprise the scenario.

In the experimental scenario, a user carries a laptop (an IBM Thinkpad T42 running Linux) into our lab (which emulates the conference room). The lab is represented by a sphere hosted by a desktop computer (*specs*) running Linux. The appropriate wireless network-sphere/IP address association is made in a voucher stored in the laptop. This enables the laptop to discover the sphere and initiate a join; a negotiation ensues.

The initial goals of the attendee device (the laptop) are to obtain membership, access to a printer, and access to a display device. The following message is created and sent to the conference room computer, thereby initiating negotiation.

REQUEST<*member*, (*possess(P),printer(P)*), (*possess(D),disp(D),ipAddress(D)*)>

The latter has the following policies governing membership.

member (X) :-

possess (X,V) , credential (V) , sound (X,prohibit,1700,1800) .

member (X) :-

possess (X,V) , credential (V) , sound (X,prohibit,T1,T2) ,

```

    acceptable(sound(X,prohibit,1700,1800),
    sound(X,prohibit,T1,T2)).
credential(V) :- socialVoucher(V,'UCLA','UCLA').
credential(V) :- socialVoucher(V,'ACM','ACM').

```

These policies mandate the following: membership can be granted to anyone who possesses a valid ACM or UCLA credential (which is a social voucher in our implementation) and which is willing to shut off sound between 1700 and 1800 hours.

The policies governing access to the printer and display device are as follows.

```

possess (hp4150) .
printer (hp4150) .
possess ('PROJECTOR') .
disp ('PROJECTOR') .
ipAddress ('PROJECTOR','131.179.192.200') .
access (S,P) :- candidateSphere(S), printer(P) .
access (S,D) :-
    candidateSphere(S), disp(D), conferenceOfficial(S) .
global (conferenceCertification) .
conferenceOfficial(S) :-    possess(S,V),
    socialVoucher(V,'ACM_Officials','ACM_Officials'),
    conferenceCertification(V,'ACM_Conference') .

```

These policies mandate that access to a printer can be granted to any device that is currently a candidate for membership, whereas access to a display device can be granted to any conference official (someone in possession of a valid social voucher).

The conference room sphere therefore responds with a counter request message that contains the following:

- ACM-certified social voucher ( $\langle \text{possess}(V), \text{socialVoucher}(V, 'ACM', 'ACM') \rangle$ ).

(There are two alternatives: an ACM certified voucher and a UCLA-certified voucher; the selection process depends on internal SWI-Prolog algorithms; in this description, we assume that the ACM voucher is requested first.)

- A voucher that certifies one to be an ACM official.
- A prohibition on sound from 1700 to 1800 hours ( $\langle \text{sound}(\text{prohibit}, 1700, 1800) \rangle$ .)

The message contents are given below.

```
→ REQUEST<possess(V),socialVoucher(V,'ACM','ACM');
    possess(V),socialVoucher(V,'ACM_Officials','ACM_Officials');
    sound(prohibit,1700,1800)>
```

The attendee device sphere does not possess either of the two kinds of vouchers that have been requested. It also has the following policies governing when it can play or turn off sounds.

```
sound(play,1201,1720) .
sound(prohibit,X,Y) :-
    (sound(play,A,B), ((nonvar(X),nonvar(Y)) ->
        ((X<A,Y<A);(X>B,Y>B)))) .
sound(prohibit,0,1200) .
sound(prohibit,1721,2400) .
sound(S,prohibit,X,Y) :- sound(prohibit,X,Y), obey(S,sound) .
obey(S,sound) :- candidateInSphere(S) .
```

Some of these policies involve low-level SWI-Prolog predicates, as we can see above. These policies indicate that the attendee device sphere needs sound to be turned on between 1200 and 1720 hours, and can be turned off at all other times. It is also

willing to obey prohibitions on playing sounds if asked by anyone it is currently in negotiation with for membership (as indicated by the `candidateInSphere` predicate). These sets of policies are incompatible with the sound prohibition request received (from 1700 to 1800 hours). Therefore, the following alternative offers are generated.

- Agree to turn off sounds between 0 and 1200 hours
- Agree to turn off sounds between 1721 and 2400 hours

Either may be offered as a first alternative, depending on SWI-Prolog internals. In this description, we will allow the first one to be offered. The resulting offer message therefore contains two entries: 1) decline the two ACM voucher requests, and 2) offer to turn off sound between 0 and 1200 hours.

→ OFFER<*negative; negative; sound(prohibit,0,1200)*>

The conference room sphere checks to see if the offer regarding sound prohibition is acceptable. It does that by evaluating the following policies.

```
overlap(T1,T2,T3,T4,O) :-
    not(T4<T1), not(T2<T3),
    ((T4<T2,T3>T1) -> O=T4-T3);
    ((T4>T2,T3>T1) -> O=T2-T3);
    ((T4<T2,T3<T1) -> O=T4-T1);
    ((T4>T2,T3<T1) -> O=T2-T1)).

acceptable(P1,P2) :-
    P1=sound(S,prohibit,T1,T2),
    P2=sound(S,prohibit,T3,T4), overlap(T1,T2,T3,T4,O), O>30.

member(X) :-
    possess(X,V), credential(V), sound(X,prohibit,T1,T2),
```

```

acceptable(sound(X,prohibit,1700,1800),
sound(X,prohibit,T1,T2)).

```

These policies indicate that an overlap of 30 minutes or greater with mandates on sound prohibition between 1700 to 1800 hours would be acceptable, and would be sufficient to grant membership. The given offer (prohibition between 0 and 1200 hours) does not satisfy this requirement, and is therefore rejected. In response, the attendee sphere tries the other alternative offer (prohibition on sound between 1720 and 2400 hours). This offer satisfies the conference room's policies and is accepted.

→ OFFER<REJECT>

→ OFFER<*sound(prohibit,1720,2400)*>

Now the conference room sphere attempts the other counter-request—a UCLA-certified social voucher.

→ REQUEST<*possess(V), socialVoucher(V,'UCLA','UCLA')*>

The attendee sphere does possess one, but its access policy for the voucher is the following.

```

access(S,U) :-
    socialVoucher(U,G,G), candidateInSphere(S), possess(S,V),
    socialVoucher(V,'NSF','NSF').

```

This generates a counter-request to the conference room sphere for an NSF-certified voucher.

→ REQUEST<*possess(U),socialVoucher(U,'NSF','NSF')*>

The latter produces such a voucher in an affirmative offer, and the former then replies with an affirmative offer containing its UCLA-certified voucher.

→ OFFER<U='voucher\_Object'>

→ OFFER<V='voucher\_Object'>

Request rollbacks occur, resulting in the attendee sphere getting membership and access to a printer. It is denied access to the display device as it cannot produce a voucher certifying it to be an ACM official. It is also offered a journal subscription in the form of a voucher, which the conference room is mandated to offer to a new member because it has the following *update* policy:

update :-

```
((negiator(Th,S), member(S)) ->
((socialVoucher(SV,G), (SV='ACM_Journal_Subscription'),
(G='ACM_Journal'),
(not(possess(S,SV))),
not(offer((possess(V), socialVoucher(V,VG)),
(V='ACM_Journal_Subscription', VG='ACM_Journal')))) ->
(assert(offer((possess(V), socialVoucher(V,VG)),
(V='ACM_Journal_Subscription', VG='ACM_Journal')))))
```

The negotiation steps are given below.

→ OFFER<affirmative; P='hp4150'; negative;

((possess(V),socialVoucher(V,VG)),(V='ACM\_Journal\_Subscription',  
VG='ACM\_Journal'))>

→ TERMINATE

Helper functions are involved in various ways during this negotiation. Whenever a voucher is received in an offer, it is verified for cryptographic integrity using a helper function. Upon conclusion of the negotiation, a helper function schedules an event to

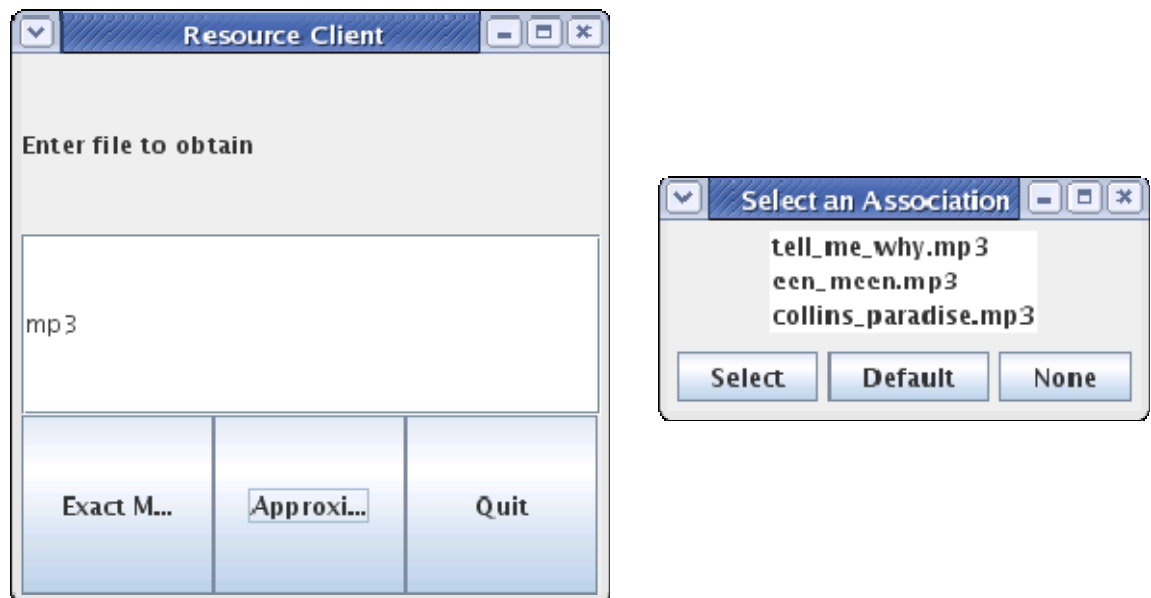
mute sound at 1721 hours and another event to un-mute it at 2400 hours (we used the *amixer* command offered by Linux for this purpose). Helper functions were also used to turn on firewall rules allowing access to the display device and printers for this particular attendee device.

We did not use a real projector display but simulated one through stub helper functions. Given a real projector display, these stub functions can be replaced with real mechanisms. These mechanisms are orthogonal to the negotiation protocol itself, so they do not detract from the veracity of the latter.

### **7.3. Peer-to-Peer File Sharing**

We designed and implemented a simple peer-to-peer application that can be configured and run in a customizable manner, and that illustrates both the negotiation and event filtering aspects of the policy manager. This application was built using the Panoply application API and is called the *P2PClient*. Many of its key features were incorporated from the Smart Party application described earlier in this chapter, and which was built to demonstrate the benefits of Panoply. A sphere must run the *P2PClient* and the *SmartPartyUserDeviceApplication* (*SPUDA*) applications in order to share files. The *P2PClient* application searches for files specified by a user by sending search events and selecting suitable spheres from which to download files. The pattern can be exact or partial; if the latter, all matching files are displayed and the user can select the one to download. The *SPUDA* application acts as a file server, responding to media search and delivery requests, whereas the *P2PClient* application receives the corresponding

responses. The *P2PClient* application also keeps track of the status of files that have been partially downloaded. If sessions are interrupted, they may be resumed at a later time without requiring a full re-download. Also, if the needed file is obtainable from multiple sources, pieces of the file can be obtained from each source, resulting in saved bandwidth at the sources. A snapshot of the *P2PClient* GUI is illustrated in Figure 19.



**Figure 19.** Snapshot of the *P2PClient* Application GUI

The peer-to-peer client application is useful in itself as a demonstration of the benefits of the Panoply middleware, but it also serves to demonstrate certain aspects of policy management as described in the example scenario in Chapter 1.

The scenario consists of a guest bringing a device into an environment that offers connectivity. We emulate this in a similar way to our other applications by having a lab member bring his personal device sphere (hosted on an IBM Thinkpad T42 laptop or a



Sony Vaio MicroPC) into the lab and attempting to join the lab sphere as a way of obtaining connectivity and being able to run the *P2PClient* application.

We discuss below three negotiation cases involving the peer-to-peer application. In the first case, the application is prevented from running as a condition of joining the lab sphere. In the second case, the guest device is allowed to run the application but with limited privileges agreed upon through negotiation. The third case illustrates the negotiation for a quantitative resource, where the parties settle on an ideal level that is dependent on the degree to which the guest device is willing to comply with the lab sphere's policies.

**First case:** The lab sphere has a policy of not letting anyone run a network application while they are members, as indicated below.

```
member(X) :- runApp(X,prohibit,App), networkApp(App,[]).
global(runApp).
global(networkApp).
```

Therefore, when the guest device initiates negotiation by asking for membership, it is presented with a counter-request asking it to prohibit the running of network applications, which includes the *P2PClient* application. The counter-request contains the following entry: `<runApp(prohibit,App), networkApp(App)>` (the predicate `networkApp` is part of the global vocabulary). In order to be a member, the guest device agrees, sending back an affirmative offer, based on its policies (which are listed below).

```
obey(S,runApp) :- candidateInSphere(S).
networkApp('panoply.apps.p2pfilessharing.P2PClient',[]).
```

The first policy rule (with `obey(S, runApp)` at its head) indicates that the guest sphere is willing to agree not to run applications in exchange for membership. The negotiation steps are as follows.

```
REQUEST<member>
→ REQUEST<runApp(prohibit,App), networkApp(App)>
→ OFFER<affirmative>
→ TERMINATE
```

The guest sphere now enforces this prohibition using the following policy.

```
runApp(S, prohibit, App) :-
    networkApp(App, []), obey(S, runApp),
    assert(runApp(prohibit, App)),
    assert(prohibitedSoUT(L, 'MediaSearch'),
    assert(prohibitedSoUT(L, 'MediaDelivery'))).
```

The above low-level policy when evaluated changes the policy database in such a way that `MediaSearch` and `MediaDelivery` events are prohibited from reaching applications. Whenever such events reach the guest sphere, the policy filter check will result in them being dropped. (*Note:* Both the lab and the guest sphere recognize that the `MediaSearch` and `MediaDelivery` events are necessary for the peer-to-peer application to work.) Also, the guest device is not permitted to start the *P2PClient* application (which is asserted as a network application through the statement: `networkApp('P2PClient', [])`) because of the following low-level update policy.

```
update :-
    ((parentSphere(S), networkApp(App, P),
    not(runApp(prohibit, App)), not(running(App))) ->
```

```

(((is_list(P),length(P,L),L>0) ->
  (jpl_datums_to_array(P,Parr0),
   jpl_datums_to_array([Parr0],Parr),
   jpl_call('panoply.utils.ApplicationLoader','launchApplication'
    , [App,Parr],Res))) ;
((is_list(P),length(P,L),L=0) ->
  jpl_call('panoply.utils.ApplicationLoader','launchApplication'
    , [App],Res))),
assert(running(App))).

```

The above policy is supposed to trigger the start of network applications whenever a sphere has been joined (thereby obtaining connectivity). But it results in the start of only those applications that are not prohibited (which can be checked by evaluating the `runApp(prohibit,App)` predicate). In our scenario, since the guest device agreed not to start this application as part of the negotiation, it will not start the *P2PClient* application after joining the sphere.

This of course does not prevent the guest device from cheating; for example, manually launching a *P2PClient* application irrespective of the negotiated agreement. The lab sphere can perform limited intrusion detection. In the same way as events are filtered through the policy engine before reaching an application, events of interest (MediaSearch and MediaDelivery events) that are routed to the guest sphere through the lab sphere can be filtered at the latter and dropped. Such a filter is not currently in place, but could be added with minimum effort, and would be guaranteed to work in the same way as event filters for applications work (for which ample evidence has been presented in this chapter).

**Second case:** The lab sphere has the following policies governing membership.

```

member(X) :- runApp(X,prohibit,App), networkApp(App,[]).

member(X) :-

    prohibitCommunicationWith(X,App,Sphere),

    networkApp(App,[]), blacklistedSphere(Sphere).

blacklistedSphere(sphere_A).

blacklistedSphere(sphere_B).

global(runApp).

global(networkApp).

global(prohibitCommunication).

```

These policies have the effect of either prohibiting members from running a network application (which includes the *P2PClient* application) or permitting members to do so while preventing them from communicating with certain blacklisted spheres. Blacklists for commonly used peer-to-peer systems consist of websites and IP addresses, which are the basic communication units. Since spheres are the basic communication units in Panoply, and our *P2PClient* application communicates on the basis of sphere events, our blacklists consist of sphere IDs.

When the lab sphere receives a membership request, it computes two alternative counter-requests:  $\langle \text{runApp}(\text{prohibit}, \text{App}), \text{networkApp}(\text{App}, []) \rangle$  and

$\langle \text{prohibitCommunicationWith}(\text{App}, [\text{sphere\_A}, \text{sphere\_B}]), \text{networkApp}(\text{App}, []) \rangle$ .

The selection in this case is non-deterministic. In this scenario, the guest device sphere does not have a policy rule of the form  $\text{obey}(S, \text{runApp})$  (absence implies negation in Prolog) but does have the following rule:

```

obey(S,prohibitCommunication) :- candidateInSphere(S).

```

Therefore, if the request `runApp` is attempted first, the guest device replies with a declination offer. When the request `prohibitCommunicationWith` is attempted, an acceptance offer is sent back. Also, the guest device updates its policy database through the following policy rules that prevent `MediaSearch` and `MediaDelivery` events destined for the *P2PClient* from crossing the policy engine filter.

```
prohibitCommunicationWith(S,App,SIDS) :-
    SIDS=[SID|RSIDS], RSIDS=[], prohibitCommunication(S,App,SID).
prohibitCommunicationWith(S,App,SIDS) :-
    SIDS = [SID|RSIDS], not(RSIDS=[]),
    prohibitCommunication(S,App,SID),
    prohibitCommunicationWith(S,App,RSIDS).
prohibitCommunication(S,App,SID) :-
    networkApp(App,[]),
    obey(S,prohibitCommunication),
    assert(prohibitedSoUT(SID,'MediaSearch')),
    assert(prohibitedSoUT(SID,'MediaDelivery')).
```

Thus the lab sphere is able to enforce its blacklist policy on members that run the peer-to-peer file sharing application. The negotiation steps are listed below.

```
REQUEST<member>
→ REQUEST<runApp(prohibit,App), networkApp(App)>
→ OFFER<negative>
→ REQUEST<prohibitCommunicationWith(App,[sphere_A,sphere_B]),
    networkApp(App,[])>
→ OFFER<affirmative>
→ TERMINATE
```

**Third case:** The guest device in this case initiates the negotiation by asking for membership as well as for a desired quantity of disk space. The following policies trigger such a request.

```

update:-
    (candidateInSphere(S) ->
        (not(localRequest(needForResources,r)) ->
            assert(localRequest(needForResources,r)))) .
maxDiskSpaceNeeded(16000) .
minDiskSpaceNeeded(6000) .
diskSpaceDecrement(2000) .
ints(K,Max,Min) :-
    Max>Min -> (K=Max) .
ints(K,Max,Min) :-
    diskSpaceDecrement(Dec), Max>Min ->
        (Max1 is Max-Dec,ints(K,Max1,Min)) .
ints(K,Max,Min) :- Max=Min -> (K=Max) .
ints(K,Max,Min) :- Max<Min -> (K=Max) .
needForResources(S) :-
    possess(S,diskSpace,D,Path),
    maxDiskSpaceNeeded(MAX), minDiskSpaceNeeded(MIN),
    diskSpaceDecrement(DEC), ints(D,MAX,MIN) .

```

These policy rules have the effect of setting the maximum desired disk space to be 16000 KB and the minimum to be 6000 KB. Also, the *diskSpaceDecrement* predicate dictates a constant concession of 2000 KB if the currently desired level cannot be obtained through negotiation. The counter-request generation algorithm computes the

possible alternative requests for disk space as the following:

`<possess(diskSpace,16000,Path)>`, `<possess(diskSpace,14000,Path)>`,

....., `<possess(diskSpace,6000,Path)>`. These different alternatives are ordered

according to the following policy rule:

```
preferable(P1,P2) :- P1=possess(diskSpace,D1,Path),
                    P2=possess(diskSpace,D2,Path), D1 >= D2.
```

The `preferable` predicate is evaluated at the time of selecting an alternative to send as a request; the first one selected is `<possess(diskSpace,16000,Path)>`. The *Path* variable is meant to return a link to the relevant filesystem where a certain amount of disk space has been allocated.

The lab sphere has the following low-level policies governing where disk space is available and how much of it is available.

```
allocatableFreeDiskSpace(FS) :-
    freeDiskSpace(FDS), allocatedDiskSpace(ADS), FS is FDS-ADS.

sumList(L,0) :- length(L,0).
sumList(L,S) :- length(L,1),L=[S|L1].
sumList(L,S) :- length(L,K), K>1, L=[F|L1], sumList(L1,S2),
    S is F+S2.

allocatedDiskSpace(FS) :-
    findall(D,(sphere(S),
    possess(S,diskSpace,D,'131.179.192.222:/mnt/tmp/'),LS),
    sumList(LS,FS).

freeDiskSpace(FS) :-
    action(run,'df /dev/hda5 | grep hda5',Result),
    string_to_atom(Result,Res), kthToken(Res,T,3),
```

```

    atom_to_num(T,FS) .
possess(diskSpace,S,'131.179.192.222:/mnt/tmp/') :-
    freeDiskSpace(F), allocatedDiskSpace(A),
    ((var(S) -> S is F-A); ((integer(S);float(S)) -> S =< F-A)) .

```

The policies that govern access to the disk space are given below.

```

access(S,diskSpace,X,'131.179.192.222:/mnt/tmp/') :-
    possess(diskSpace,D,'131.179.192.222:/mnt/tmp/'),
    minDiskSpaceGrant(G), G =< D, X =< G.
access(S,diskSpace,X,'131.179.192.222:/mnt/tmp/') :-
    possess(diskSpace,D,'131.179.192.222:/mnt/tmp/'),
    maxDiskSpaceGrant(G), minDiskSpaceGrant(H),
    G =< D, X > H, X =< G,
    runApp(X,prohibit,App), networkApp(App,[]) .
minDiskSpaceGrant(9000) .
maxDiskSpaceGrant(11000) .

```

Theoretically, the maximum amount of disk space that the lab sphere is willing to grant anyone is 11000 KB. It will freely grant requests of 9000 KB or less to anyone without imposing any conditions. In practice, these numbers may be adjusted based on the amount of disk space currently used up or allocated. The above policies result in the lab sphere denying requests for disk space above 11000 KB. A request lying between 9000 KB and 11000 KB will result in a counter-request to prevent the running of a network application, including the P2PClient.

Let us consider the case where the guest device has a policy rule which states that it is willing to prevent network applications from running if requested.

```

obey(S,runApp) :- candidateInSphere(S) .

```



```

networkApp('panoply.apps.p2pfilessharing.P2PClient', []).
runApp(S,prohibit,App) :-
    networkApp(App,[]), obey(S,runApp),
    assert(runApp(prohibit,App)),
    assert(prohibitedSoUT(L,'MediaSearch')),
    assert(prohibitedSoUT(L,'MediaDelivery')).

```

The negotiation starts by the guest device requesting 16000 KB; in response to this, it receives a declination offer. It then asks for 14000 KB and 12000 KB respectively; both requests are denied. Then it asks for 10000 KB, which lies below the maximum limit of the lab sphere but above its minimum limit. The lab sphere then responds with a counter-request asking the guest device to prohibit the running of network applications. The guest device can comply with this request, and sends back an affirmative offer. The request for 10000 KB is therefore granted and the negotiation terminates.

If the guest device does not have a policy dictating that it can stop network applications when requested, it will deny the lab sphere's following request: `<runApp(prohibit,App),networkApp(App,[])>`. The lab sphere, in turn, denies the request for 10000 KB of disk space. Now the guest sphere attempts another request, for 8000 KB. As this is less than 9000 KB, the lab sphere freely grants the request and the negotiation terminates. The negotiation steps are listed below.

```

REQUEST<possess(diskSpace,16000,Path)>
→ OFFER<negative>
→ REQUEST<possess(diskSpace,14000,Path)>
→ OFFER<negative>
→ REQUEST<possess(diskSpace,12000,Path)>

```

→ OFFER<*negative*>

→ REQUEST<*possess(diskSpace,10000,Path)*>

→ REQUEST<*runApp(prohibit,App),networkApp(App,[])*>

If the guest device is willing to agree not to run a network application, the remaining steps are as follows.

→ OFFER<*affirmative*>

→ OFFER<*Path='131.179.192.222:/mnt/tmp/'*>

→ TERMINATE

If the guest device is not willing to agree to prohibit the running of network applications, the remaining steps are as follows.

→ OFFER<*negative*>

→ OFFER<*negative*>

→ REQUEST<*possess(diskSpace,8000,Path)*>

→ OFFER<*Path='131.179.192.222:/mnt/tmp/'*>

→ TERMINATE

After an offer of disk space is granted, the lab sphere uses helper functions to set quotas on the amount of space that the recipient sphere can access. On Linux and Unix systems, this is done using the *quota* command.

#### **7.4. Network Access Control Based on the QED Model**

We outlined a scenario in Chapter 1 where maintaining a secure perimeter around a network of IEEE panel attendees requires either manual reconfiguration of attendees' computers or intrusive and inflexible automated patching mechanisms. Panoply

researchers originally designed and implemented an automated framework for this purpose called QED (Quarantine, Examination and Decontamination) [Eustice2003b], through which incoming devices can be isolated from the network temporarily, examined for vulnerabilities, and patched if necessary. Here, we demonstrate how our negotiation protocol can provide a functionality that is identical to that which QED provides, and which enables more flexible security perimeters from both the network's and the client's point of view.

In our scenario, the IEEE network was emulated by our laboratory's wireless network, and a lab member's IBM Thinkpad T42 laptop was treated as an attendee's device. The laptop communicated with the laboratory sphere, which ran on the wireless network's gateway. The scenario starts with the laptop discovering and joining the lab sphere, and initiating the negotiation process by requesting membership. The lab sphere's membership policies are given below:

(I) member(X) :-

```
action(X,order,run,C,ubuntu),checkDistribution(C),file(C,'.').
```

(II) member(X) :-

```
action(X,order,run,C,redhat),checkDistribution(C),
file(C,'.').
action(X,order,run,'uname -a | cut -f 3 -d \' \' | cut -f
1,2,3 -d \'.\'','2.6.20').
```

(III) member(X) :-

```
action(X,order,run,C,redhat),checkDistribution(C),
file(C,'.').
```

```

action(X,order,run,'uname -a | cut -f 3 -d \' \' | cut -f
1,2,3 -d \'.\'',V), not(V='2.6.20'),
action(X,order,run,PF,Result), patchFile(PF), file(PF, '.'),
Result='SUCCESS'.
file(check_os_distro, '.').
checkDistribution(check_os_distro).
file(patch_redhat_oldver, '.').
patchFile(patch_redhat_oldver).

```

These policies indicate that the network is willing to provide free membership to a computer that is running Ubuntu Linux (rule I). The constraints for a computer that runs RedHat Linux are greater. However, a computer running RedHat may get free access if its kernel version is 2.6.20 (rule II), presumably the most up-to-date version; otherwise it must run patch software (rule III). These are extracted as different alternative sets by the counter-request generation method. The attendee's laptop is sent a counter-request containing any of these alternative sets. If it is running Ubuntu, it will eventually receive the appropriate alternative request, and send back an affirmative reply stating that it is running Ubuntu. In return, it will be granted membership rights and the negotiation will terminate. The negotiation steps for this scenario are given below.

```

REQUEST<member>
→ REQUEST< action(run,C,ubuntu),file(C,')>
→ OFFER<affirmative>
→ TERMINATION

```

Likewise if it is running RedHat and the 2.6.20 kernel, it will eventually send back an affirmative reply. The steps for such a negotiation are given below.

REQUEST<*member*>

→ REQUEST< *action(run,C,ubuntu),file(C,') {C='check\_os\_distro'}*>

→ OFFER<*negative*>

→ REQUEST< *action(run,C,redhat),file(C,') {C='check\_os\_distro'}*;

*action(run,'uname -a | cut -f3 -d \'\' | cut -f1,2,3 -d \'\'','2.6.20')>*

→ OFFER<*affirmative; affirmative*>

→ TERMINATION

Helper functions are used to attach the file '*check\_os\_distro*' to the REQUEST messages above. Counterpart helper functions are used to extract and save the files to the local filesystem at the destination.

The low-level *action* policies that are used to run code are given below.

```
action(run,Prog) :- shell(Prog,0).
```

```
action(run,Prog) :-
```

```
    file(Prog,Dir),atom_concat(Dir,'/',COM1),
```

```
    atom_concat(COM1,Prog,COM),shell(COM,0).
```

```
action(run,Prog,Result) :-
```

```
    jpl_call('panoply.policy.Helper','generateTemporaryFileName',[
    ],TF),
```

```
    atom_concat(Prog,' 1> ',C1), atom_concat(C1,TF,C), shell(C),
```

```
    open(TF,read,Stream),
```

```
    (readFile(Stream,Result1) ->
```

```
    (close(Stream), delete_file(TF))), string_length(Result1,L),
```

```
    ((L>0 -> (L1 is L-1, sub_string(Result1,0,L1,1,Result))) ;
```

```
    (L==0 -> (Result=''))).
```

```

action(run, Prog, Result) :-
    file(Prog, Dir), atom_concat(Dir, '/', Prog1),
    atom_concat(Prog1, Prog, Prog2),
    jpl_call('panoply.policy.Helper', 'generateTemporaryFileName', [
    ], TF),
    atom_concat(Prog2, ' 1> ', C1), atom_concat(C1, TF, C), shell(C),
    open(TF, read, Stream),
    (readFile(Stream, Result1) ->
    (close(Stream), delete_file(TF))), string_length(Result1, L),
    ((L>0 -> (L1 is L-1, sub_string(Result1, 0, L1, 1, Result))) ;
    (L==0 -> (Result=''))).

```

If the laptop is not running either Ubuntu or RedHat, its membership request is denied and the negotiation results in failure.

In a variation, if the laptop is running RedHat and the 2.6.17 kernel, it is asked to run a patch file. This patch file is sent by the lab sphere along with the request (a helper function attaches the file object to the negotiation message). The file is extracted and saved by the laptop sphere (through another helper function). The steps for this part of the negotiation are given below.

```

REQUEST<member>
→ REQUEST< action(run,C,ubuntu),file(C,') {C='check_os_distro'}>
→ OFFER<negative>
→ REQUEST< action(run,C,redhat),file(C,') {C='check_os_distro'};
    action(run,'uname -a | cut -f3 -d \'|\' | cut -f1,2,3 -d \'|\'','2.6.20')>
→ OFFER<affirmative; negative>

```

```

→ REQUEST< action(run,C,redhat),file(C,') {C='check_os_distro'};

        action(run,'uname -a | cut -f 3 -d ' ' | cut -f 1,2,3 -d '\.',V);

        action(run,PF,Result),file(PF,') {PF='patch_redhat_oldver'}>

```

But the following policies govern how the laptop runs arbitrary binary code.

```

obey(S,run,A,B) :-
    obey(S,run),      file(A,D),      executable(A),      possess(S,V),
    socialVoucher(V,'UCLA','UCLA'),
    action(S,permission,runApp,App), networkApp(App,Params).

obey(S,run,A,B) :-
    obey(S,run), file(A,D), not(executable(A)).

obey(S,run,A,B) :-
    obey(S,run), not(file(A,D)).

executable(A) :-
    file(A,D),      atom_concat(D,'/',C),      atom_concat(C,A,C1),
    atom_concat('file ',C1,C2),
    atom_concat(C2,' | grep executable 1>/dev/null',COM),
    action(run,COM).

```

These policies are currently unsatisfied, resulting in a counter-request for a valid UCLA voucher and permission to run a *network application* (which is understood by both interacting domains). The lab sphere has policies allowing it to release its UCLA voucher to whoever asks for it, but it is not willing to permit the running of network applications unless the requester shuts down any service that is listening on port 25. The policies (both high- and low-level ones) are given below.

```

access(S,V) :- socialVoucher(V,'UCLA').

```

```

action(S,permission,runApp,App) :-
    networkApp(App,Params), closedPort(S,25).

closedPort(S,Po) :-
    (negotiator(Thr,G) ; childSphere(G)), ipAddress(G,IP),
    atom_concat('nmap -sS ',IP,C0),
    atom_concat(C0,' -p ',CP), atom_concat(CP,Po,C1),
    atom_concat(C1,' | grep ',C2),
    atom_concat(C2,Po,C3), atom_concat(C3,'/tcp',C),
    action(run,C,Result),
    string_to_atom(Result,Res), kthToken(Res,'open',1),
    action(S,order,closePort,Po).

closedPort(S,Po) :-
    (negotiator(Thr,G); childSphere(G)), ipAddress(G,IP),
    atom_concat('nmap -sS ',IP,C0),
    atom_concat(C0,' -p ',CP), atom_concat(CP,Po,C1),
    atom_concat(C1,' | grep ',C2),
    atom_concat(C2,Po,C3), atom_concat(C3,'/tcp',C),
    action(run,C,Result),
    string_to_atom(Result,Res), (kthToken(Res,'filtered',1) ;
    kthToken(Res,'closed',1)).

```

The laptop is willing to shut down services on port 25, and does so through the following low-level policy.

```

action(closePort,Po) :- atom_concat('iptables -A INPUT -j DROP -p
    tcp --dport ',Po,C1), atom_concat(C1,' -i lo',C), shell(C,0).

```



It sends back an affirmative reply. Rollback of requests occurs through multiple affirmative offers, resulting in the laptop gaining membership within the lab sphere. The steps of this portion of the negotiation are listed below.

```

→ REQUEST<possess(V),socialVoucher(V, 'UCLA', 'UCLA');
           action(permission,runApp),networkApp(App,Params)>
→ REQUEST<action(closePort,25)>
→ OFFER<affirmative>
→ OFFER<V='voucher_Object'; affirmative>
→ OFFER<affirmative; V='2.6.17'; Result='SUCCESS'>
→ OFFER<affirmative>
→ TERMINATE

```

## 7.5. Opportunistic Configuration

Successful interactions between domains (with or without negotiation) often combine transaction with configuration. In our framework, negotiations that result in affirmative offers are logically equivalent to transactions. But these transactions may be functionally incomplete unless the recipient has the ability to recognize, parse, or utilize the offer in a meaningful way. Often, supplementary objects or mechanisms that are not currently available to the recipient must be obtained from the provider for a successful transaction. Consider these examples. A negotiation results in one domain offering an image or audio file to the other. The file is encrypted using DRM technology, and the recipient may (if it has a prior relationship with the sender) or may not (if this is a new relationship) have the encryption key. In the latter situation, the received file is of no use if it cannot be read. In

keeping with the principles of our overall research objectives, the necessary keys should be transacted along with the primary requested file through negotiation. Similarly, a credential (a certificate or a voucher) can be verified using the owner's public key, and the recipient ought to be able to get that public key from the sender so that it can verify the integrity of the received credential. In a different scenario, a negotiation can lead to one domain offering access to a service, but without the ability to use that service through a suitable interface or proxy (typically a piece of code), the access rights are meaningless. The recipient may already possess the knowledge and a suitable interface; if it does not, it ought to be able to get that information (and code if necessary) through negotiation.

The last example above is related to early research in open and ubiquitous computing, a significant portion of which was motivated by the need to discover local services and use them. JINI [Waldo1999] was one such framework that solved the problem by combining lookup directories with portable Java class files as service proxies. Our framework can provide an identical functionality through negotiation, given that the requester has suitable policies framed in its database. Though these frameworks provided protocols for configuration, we can specify arbitrary constraints in addition to the basic object of the transaction, creating more flexible and realistic scenarios.

Below, we describe two examples of opportunistic configuration through negotiation. The examples we show are simple negotiations with no variations, and serve to demonstrate only the kinds of configurations discussed above.

### 7.5.1 Negotiation for Credential and Associated Key

We have two spheres  $S_1$  and  $S_2$  negotiate with each other.  $S_1$  initiates the negotiation by requesting membership from  $S_2$ . The latter has the following membership policies:

```
(I) member(S) :- possess(S,U), socialVoucher(U,G),  
    negotiator(Thr,G), verifiable(X,U).  
  
(II) member(S) :- possess(S,U), socialVoucher(U,G),  
    negotiator(Thr,G), verifiable(U).  
  
verifiable(S,V) :-  
    possess(S,K), voucherKey(V,K), publicKey(K,F,D).  
  
verifiable(V) :-  
    voucherKey(V,K), possess(K), publicKey(K,F,D).
```

$S_2$  sends back a counter-request for a social voucher, and  $S_1$  replies with an offer of a social voucher because its *access* policy is the following:

```
access(S,U) :- socialVoucher(U,G), localSphereID(G).
```

This policy allows a social voucher to be released to anyone who asks for one.  $S_1$ , in turn, grants membership and the negotiation terminates.

Variations in this scenario are dependent on  $S_1$ 's possession of  $S_2$ 's public key which is necessary to verify the integrity of the voucher received. If  $S_1$  already possesses the public key prior to receiving the membership request, membership policy II is evaluated, and a counter-request for a social voucher is sent to  $S_2$ .  $S_2$  sends back the social voucher in response. The negotiation steps are listed below.

```
REQUEST<member>  
→ REQUEST<possess(S,U), socialVoucher(U,'S2')>
```

→ OFFER<U= 'voucher\_Object'>

→ OFFER<affirmative>

→ TERMINATION

If  $S_1$  does not possess the public key prior to the negotiation, its counter-request consists of both the voucher and its public key. The extra support predicates (`voucherKey` and `publicKey`) let  $S_2$  know that it must append its public key to the voucher in its reply. The steps are listed below.

REQUEST<member>

→ REQUEST<possess(U),socialVoucher(U,'S<sub>2</sub>'),voucherKey(U,K),publicKey(K,F,D)>

→ OFFER<U= 'voucher\_Object'>

→ OFFER<affirmative>

→ TERMINATION

The *controller* module within  $S_2$ 's policy manager invokes the appropriate helper function that retrieves the public key and attaches it to the *extra* field of the negotiation message. Correspondingly,  $S_1$ 's *controller* module invokes a dual helper function that extracts the key object, stores it, updates the policy database, and verifies the integrity of the voucher object. The helper functions consist of 10 to 15 lines of code that implement data extraction from files, serialization, and storage of the bytes into files.

### 7.5.2 Negotiation for Service and Associated Access Mechanism

We have two spheres  $S_1$  and  $S_2$  negotiate with each other.  $S_1$  has the following policies governing its resource needs, which currently include only printer access.

```

needForResources(S) :-
    possess(S,U), printer(U), printerCommand(P,C) .
needForResources(S) :- possess(S,U), printer(U),
    not(printerCommand(P,C)), printerCommand(P,C) .

```

$S_1$  initiates negotiation by requesting access to a printer, and  $S_2$  replies with an affirmative offer promising access to a printer because its *access* policy is the following:

```
access(S,P) :- printer(P) .
```

This policy allows granting printer access to anyone who asks for it. The following facts indicate the type of printer possessed by  $S_2$  and the command used to invoke the print service.

```

possess('HP7100') .
printer('HP7100') .
printerCommand('HP7100', 'printCommand') .
file('printCommand') .

```

The negotiation terminates when  $S_1$  receives the offer. The `printerCommand` predicate indicates that the printer can be accessed by running the *printCommand* file.

Variations in this scenario are dependent on  $S_1$ 's possession of a suitable piece of code that would enable it to actually use the printer. If  $S_1$  already possesses the appropriate piece of code, the predicate `printerCommand(P,C)` is satisfied, membership policy II is evaluated, and the corresponding counter-request consists of the following:  $\langle \text{possess}(U), \text{printer}(U) \rangle$ .  $S_2$  sends back an affirmative offer in response, with the binding  $\{U='HP7100'\}$ .  $S_1$  can access the printer through the code it already possesses. The negotiation steps are listed below.

```
REQUEST< possess(U), printer(U) >
```

→ OFFER<U='HP7100'>

→ TERMINATION

If  $S_2$  does not possess the appropriate piece of code, its counter-request message consists of the following:  $\langle \text{possess}(U), \text{printer}(U), \text{printerCommand}(U,C) \rangle$ . The extra support predicate `printerCommand` lets  $S_2$  know that it must append a piece of code or a command to its reply. The steps are listed below.

REQUEST<*possess*(U), *printer*(U), *printerCommand*(U,C)>

→ OFFER< U='HP7100', C='printCommand'>

→ TERMINATION

The *controller* module within  $S_2$ 's policy manager invokes the appropriate helper function that retrieves the *printCommand* file and attaches it to the *extra* field of the negotiation message. Correspondingly,  $S_1$ 's *controller* module invokes a dual helper function that extracts the *printCommand* file, stores it, and updates the policy database.

In our test implementation, we simply had the following command within the *printCommand* file: `'/usr/bin/lpr -P ficus'`. The following update policy enabled  $S_1$  to print a test page as soon as printer access was obtained.

```
update :- ((printer(P), possess(P), not(printedTestPage(P))) ->
           (action(print, TestPage), assert(printedTestPage(P)))) .
```

The following low-level policy results in the printing of the test page.

```
action(print, F) :- file(F), printer(P), printerCommand(P, C),
                    atom_concat(C, ' < ', C1), atom_concat(C1, F, COM), shell(COM, 0) .
```

## **Chapter 8**

### **Theoretical Analysis and Commentary**

Our negotiation framework has two aspects, or roles:

- 1) The negotiation protocol is a system/middleware mechanism for information and object exchange and agreement generation.
- 2) The negotiation model is a decentralized way of policy resolution, which manifests itself through the protocol.

We cannot completely separate these two roles, but it is useful and illuminating to deal with these separately. We have already mentioned some of the systems properties of the negotiation protocol in Chapters 5 and 6. We will first expand on those properties here, and then discuss more theoretical formal properties of negotiation when visualized as a distributed search tree.

#### **8.1. System Analysis of the Negotiation Protocol**

From a system/network designer or administrator's point of view, certain properties of a protocol are of paramount importance. These properties impact practical considerations like robustness and efficiency. It would be desirable to have the protocol either anticipate or be tolerant to failures, and be self-correcting. Also, such faults should not disrupt the normal workings of other components of the framework which the protocol is part of. In Chapter 6, we described how our negotiation protocol is tolerant to failures and race

conditions. There, we dealt with the protocol itself as being one unit, or a black box. Here, we go into the internals of its workings, and prove its termination properties. We draw upon the operations described in the flowcharts in Figure 8a and b, as well as the steps outlined in the policy engine algorithms (see Section 5.3) to infer properties, and show why any instance of the protocol is guaranteed to terminate, given that certain conditions can be imposed on the policies within the database. In particular, any distributed system procedure, which is what the negotiation protocol is, is susceptible to deadlocks and livelocks, and theoretical guarantees for the absence of either are necessary for it to be usable in a real-world system. We must note that eventual termination is not the only consideration; if a negotiation instance takes many hours and a huge number of steps, manual intervention would be preferable. Also, eventual termination does not say anything about the quality and nature of the agreement reached. We analyze these properties, from both a theoretical point of view and through measurements, later in this chapter, and in the next chapter.

### **Negotiation Protocol Termination**

Both the controller layer (which maintains request lists and formulates responses) and the policy engine layer (which runs counter-request and alternative-offer generation algorithms) run complex procedures, which were described in Chapter 5. We state certain properties of each of these procedures in the form of theorems, and provide an analytical proof for termination of the protocol irrespective of the policy contents, number of



negotiation steps, and the number of generated alternatives. These properties were mentioned in Chapter 5, but we state them here in a formal way.

***Policy Engine Features:***

*Property 1:* The policy database is of finite length; i.e., the number of statements (facts and rules/clauses) is bounded at any instant.

*Property 2:* Each policy rule is of finite length; i.e., the number of predicates in the body of a clause is bounded.

*Property 3:* There are no cycles in the policy database; i.e., predicates are not self-referential either directly or through transitive links.

*Property 4:* A policy rule is examined for the purpose of generating counter-requests no more than once during a negotiation session.

*Theorem 1:* The counter-request generation procedure always terminates.

*Proof:*

The algorithm takes as argument a request predicate in the form of a string.

- I) At each level of recursion, only the *relevant* clauses are examined, which are those clauses whose heads match the argument string (predicate name and arity for both are identical).
- II) The number of relevant clauses is bounded by the number of clauses in the policy database, which is finite.
- III) The number of recursive calls that are made for each clause examined is bounded by the number of predicates in the body of the clause. Since each clause has finite

length, the number of recursive calls is bounded.

- IV) The total number of recursive calls made is bounded, since there are no policy cycles.
- V) Each clause, or predicate, or recursive call is made exactly once during a run of the algorithm. Since the total number of these operations is bounded, the procedure is guaranteed to terminate within a bounded amount of time.

*Analysis:*

Let the number of facts in the policy database be  $F$  and the number of rules be  $R$ , the maximum number of predicates in the body of any rule be  $S$ , and the maximum arity of any predicate be  $A$ .

At each level of recursion, a set of matching clauses is extracted and examined. The number of matching clauses is bounded by  $R$ . For each such clause, the predicates that can be sent as counter-requests are extracted. This is bounded by  $S$ . A Prolog query is run over the body of each matching clause. This query processing time, denoted by  $QPT$ , can be roughly assumed to be constant for practical purposes; this is evident from the performance measurements in Section 9.1, where the processing time does not change with change in clause body size. The number of solutions is bounded by  $2^F$ , which is the total number of subsets of the set of facts in the database. For each such solution, the variables in the relevant clause body are substituted with the solution; time complexity for this substitution is  $O(AS)$ . For each possible request predicate from this body, we test for satisfiability, which takes  $QPT$  (roughly constant) time. If unsatisfiable, the counter-request generation algorithm is run on that predicate in a recursive manner. Let us

temporarily assume the running time for the recursive call to be  $REC$ . Extracting support predicates then takes  $O(AS)$  time. Therefore, the total running time complexity is  $O(R)*[O(S) + QPT + O(2^F)*[O(AS) + O(S) [REC + O(AS)]]]$ . Since we have mandated that our policy database cannot have cycles, the maximum number of recursive calls (or the number of nodes in the recursion tree) is bounded by  $R+F$ . Therefore, we can eliminate the  $REC$  term from the above expression and instead multiply the entire term by  $R+F$ . We can eliminate the constant term  $QPT$  as well. Therefore, the running time of the entire procedure is  $O(R(R+F)(S + 2^F S^2 A)) = \mathbf{O(R(R+F)2^F S^2 A)}$ .

*Theorem 2:* The alternative-offer generation procedure always terminates.

*Proof:*

The algorithm takes as argument a request predicate in the form of a string.

- I) First, the given argument predicate is made more general by replacing argument constants with variables. Since the arity of each predicate is bounded, this procedure is linear in the size of the predicate.
- II) Each clause whose head matches the request string is examined for a potential alternative offer. The number of such clauses is bounded by the number of policy rules in the database (Policy Engine: property 1).
- III) The body of each satisfiable clause is then examined in order to extract descriptive or support predicates. Each examination is bound by the size of that particular policy rule, which in turn is bounded (Policy Engine: property 2).
- IV) The set of obtained alternative offers are ranked in order of relevance (closest

match) to the original argument string. This procedure terminates in polynomial time in terms of the size (arity) of the request string and the size of the set (of alternative offers) generated.

*Analysis:*

Let the number of facts in the policy database be  $F$  and the number of rules be  $R$ , the maximum number of predicates in the body of any rule be  $S$ , and the maximum arity of any predicate be  $A$ .

The first step in the above procedure runs in  $O(A)$  time. Selection of matching clauses takes  $O(R)$  time. For each clause, a Prolog query is run on the body. This query processing time, denoted by  $QPT$ , can be assumed to be constant (as we assumed in the counter-request generation procedure case). Support predicates are extracted in  $O(AS)$  time. Since the set of alternative offers obtained is bounded by  $R$ , the ranking algorithm runs in the time it takes to sort the set, or  $O(R * \log R)$  time. Hence the entire procedure runs in  $O(R(AS + \log R + QPT)) = O(R(AS + \log R))$  time. It would be extremely uncommon for predicate arities ( $A$ ) to be larger than 3 or 4, which is effectively a constant. In most cases,  $S$  will also be relatively small compared to the number of rules. If the database is small,  $QPT$  is approximately constant. On the other hand, since query processing is exponential in complexity, it would dominate in medium- and large-size databases. Therefore, we can conclude that the run time complexity is  $O(R * \log R)$  for small databases, and is  $O(R * QPT)$  for large databases.

***Controller Features:***

*Property 1:* A request is added to the MADE\_REQUESTS list only if:

- It is an initial request (goal).
- It is a counter-request posed in response to a (currently unsatisfiable) request just received.
- It is an alternative counter-request posed in response to a request received earlier.

*Property 2:* A request is added to the RECEIVED\_REQUESTS list only if it is an entry in a REQUEST message just received.

*Property 3:* A request is removed from the RECEIVED\_REQUESTS list only if an OFFER message going to be sent contains a corresponding entry.

*Property 4:* A request is removed from the MADE\_REQUESTS list only if:

- An affirmative reply is received through an OFFER message.
- A negative reply is received through an OFFER message and there are no alternative counter-requests available to send.

*Property 5:* An offer to a received request is sent only if:

- That request is satisfiable.
- No counter-requests can be posed in response to that request.

*Theorem 1:* The number of requests made, starting from the original goals and including all counter-requests, is bounded by the total number of request predicates occurring in the bodies of all policy clauses within the local database.

*Proof:* Follows from Policy-Engine properties 1, 2, 3, and 4.

*Theorem 2:* The MADE\_REQUESTS list will become empty within a finite number of steps.

*Proof:*

- I) The number of requests made during a negotiation session is finite (Controller: theorem 1). Let us denote this set by  $R_{MADE}$ .
- II) The set of alternative counter-requests is a subset of  $R_{MADE}$ , and hence is finite.
- III) The number of counter-requests available for a particular entry in MADE\_REQUESTS is finite.
- IV) If that entry keeps receiving negative offers in reply, it will eventually be taken off the list.
- V) An affirmative offer will result in the entry being taken off immediately.
- VI) Every request must receive a reply.
- VII) Each entry in the MADE\_REQUESTS list will be removed in a finite number of steps (IV, V, and Controller: property 4).
- VIII) The number of additions to the MADE\_REQUESTS list is bounded (I), and each entry added will be removed; hence the list will eventually become empty.

*Theorem 3:* The RECEIVED\_REQUESTS list will become empty within a finite number of steps.

*Proof:*

- I) There are a finite number of requests received during a negotiation session.

(Controller: theorem 2, tells us that the number of requests made will be finite, hence the number of requests received by the other side will be finite.)

- II) A request received will either be satisfiable, unsatisfiable, or engender counter-requests.
- III) The size of the counter requests set is bounded (Policy Engine: properties 1, 2, 3, and 4).
- IV) An offer is always sent in response to a request received (II, III).
- V) The RECEIVED\_REQUESTS list eventually becomes empty (I, IV).

*Theorem 4:* The negotiation protocol is guaranteed to terminate in a finite number of steps.

*Proof:* This follows from the above theorem. Referring to Figure 8b, a negotiator sends a TERMINATE message when its RECEIVED\_REQUESTS list becomes empty. Since this list will become empty within a finite number of steps, the protocol will terminate within a finite number of steps.

***Deadlocks:*** *The negotiation protocol does not suffer from deadlocks.*

A procedure is said to be deadlocked when it ends up in a state where all participants are waiting for an event to occur, but none of them is able to initiate an action that would advance the overall state. Referring to the high-level negotiation state machine (see Chapter 5, Figure 7), there are potentially four states in which a deadlock could occur:

- *Initiate:* This state only checks whether or not there are needs that must be met

through negotiation. These needs are inferred by running a Prolog query, which terminates because the policy database is finite and does not contain cycles.

- *Service*: The following operations are performed in this state: database queries (in Prolog) to verify satisfiability of received requests, counter-request generation, and alternative-offer generation. We have proved that these procedures terminate. Since a message is always sent in response to a request, this state does not deadlock.
- *Process*: In this state, a received offer is checked for veracity, and one of the following is guaranteed to be returned: an OFFER REJECT, an alternative offer if available, alternative counter-requests if available, offers in reply to requests received (rollback). The default response, when no alternative offers or requests exist, is to consider requests in the RECEIVED\_REQUESTS list to be unsatisfiable, and send back negative offers. If the RECEIVED\_REQUESTS list is empty, a termination message is sent. A response is always sent; therefore, this state does not deadlock.
- *Expect*: When a negotiator is in this state, it is waiting for a response from the other party. The dynamics of negotiation indicate that the other party must then be in the *process* or *service* state. As we have seen, those states do not deadlock. Hence, a response is always received, and the *expect* state never deadlocks.

***Livelocks***: *The negotiation protocol does not suffer from livelocks as long as the collective policies of the two parties contain no cycles.*

A procedure is said to be in a livelock if the state transitions are cyclical. This results in participants performing repetitive actions that do not lead to overall progress, or achieve a



useful result. Whereas a deadlock implies being stuck in one particular state, a livelock implies being stuck in a set of states. In practice, this is harder to avoid and detect than a deadlock, since there is apparent progress being made.

Referring to the state machine, we can see that the only possibility of a livelock involves a cycle of *expect* and *process/service* states. Such a cycle could conceivably result only if the contents of the requests themselves form a cycle. In the simplest case:

- Negotiator N1 sends a request R1 to negotiator N2
- N2 replies with a counter request R2
- N1 replies with a counter-request that is identical to R1

Other livelock scenarios could be imagined that would have such a cycle interspersed with more requests and offers. Incidentally, this could cause the state to blow up at both ends, since duplicate requests keep getting added to the request lists. The core reason for the occurrence of such a livelock is a cycle in the combined database containing policy statements of both negotiators.

There are two ways of handling a livelock situation.

- 1) One way is to mandate that there be no policy cycles. On the relatively rare occasion where such a cycle exists, the protocol would fail to reach a result in the following way: the parties would have set limits on the amount of state they are willing to maintain, and once that limit is exceeded, they both declare failure by one negotiator sending a termination message to the other.
- 2) The other way is by enforcing Policy Engine: property 4. Every individual policy rule has a unique identifier associated with it (in practice, SWI-Prolog provides this

feature). Each negotiator can then keep track of which policy rules have already been examined for counter-requests, and none is examined more than once. This has the advantage of avoiding the blowing up of state at both ends, though it would incur additional overhead in all scenarios where no livelock exists. The result would still be a negotiation failure, since the negotiators would eventually run out of policy statements to examine.

We will discuss the correctness and completeness properties of a negotiation protocol with potential livelocks in the following theoretical analysis section.

## **8.2. Theoretical Analysis of Negotiation**

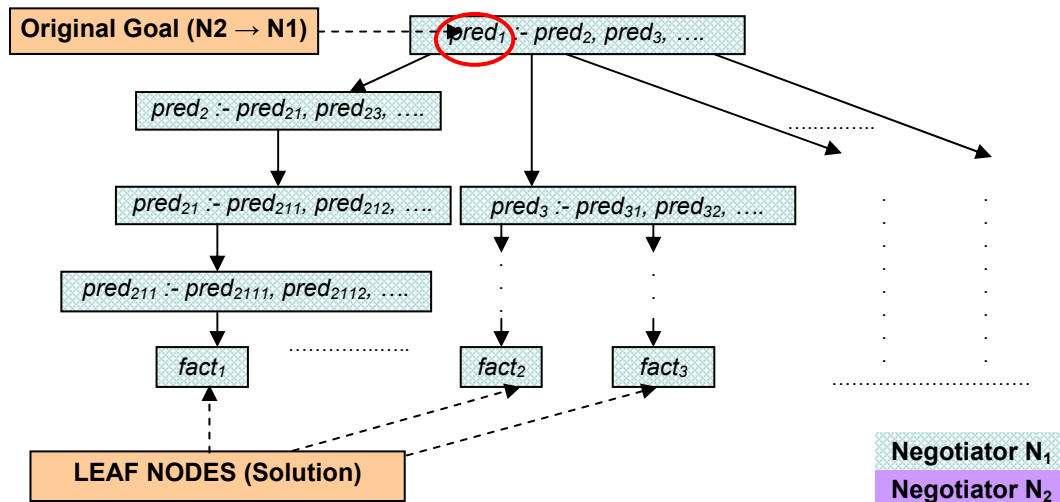
Our system analysis of the negotiation protocol did not consider the logical aspects that are fundamental to the operations of the back-end policy engine. The facts and rules in the policy database have logical properties, and consequently, the algorithms for generation of counter-requests and alternative offers can be formally analyzed on a logical basis. The messages being communicated contain parts of policy rules; hence, the collective operations performed by two negotiating domains, starting with goal requests, have properties that can be evaluated against standard logical metrics. In the remainder of this section, we state what the important metrics are, and compare our negotiation procedure to them.

### 8.2.1 Policy Resolution

As described through our negotiation model in Chapter 3, the purpose of interaction between two domains is the satisfaction (partial or full) of their goals within the bounds of their collective policy constraints. Through this process, the goals and policies of one domain are resolved against the other's goals and policies. Resolution here refers to evaluation of logical consistency among the policies and requirements of the interacting domains. It is not to be confused with logical resolution, a standard theorem-proving technique in AI. In the ubiquitous interoperation context, policy resolution is a way of inferring whether or not a set of goals can be satisfied, and if so, how they can be satisfied.

A goal can be resolved against a set of policies, and a policy rule can be resolved against a set of policies, by evaluating whether they are logically consistent, and if so, what is the set of ways in which they are logically consistent. Our policy language is Prolog-based, and the requests/goals of a negotiation are formatted as Prolog queries. As described in Chapter 5, the policy engine algorithms use the Prolog querying framework to infer whether requests are satisfiable, or whether counter-requests or alternative offers are available. Our counter-request generation algorithm is a process of extracting unsatisfied constraints and presenting them as requests to the negotiator. The negotiation protocol results in the execution of this algorithm multiple times in a sequence, and is equivalent (as we will see further below) to the backward-chaining algorithm used by Prolog (and by first-order logic theorem provers).

The backward chaining algorithm is used to find all possible solutions to a goal predicate consistent with a given database consisting of facts and rules. It generates a search tree, with each intermediate node representing a rule (a policy with constraints and dependencies) and each leaf representing a fact (a policy without constraints and dependencies). Paths are inspected in a depth-first manner, and the lack of a matching fact triggers backtracking (or a rollback of the recursion). A search tree with valid paths and leaves (excluding the backtracked paths) contains goals (at the root), policy rules and facts (nodes at lower levels) that resolve (are logically consistent) with each other. The set of leaves returns a set of solutions, all of which could satisfy the original goals. Figure 20 illustrates an example of such a search tree. Resolving goals and policies with each other using a search tree is an inherently centralized operation, but we envisioned negotiators simulating such a resolution in a distributed manner in order to reconcile their needs and constraints. The differences between the centralized and distributed cases are described in the following sections.



**Figure 20.** Centralized Search Tree Representing Policy Resolution by an Oracle

We must note that all scenarios that involve domains policy resolution are not, and do not need to be, based on logic. All policy frameworks have some notion of matching though; i.e., there is a list of policies, and goals are matched with those policies (or rules) to find out whether or not the goal fits within the constraints. This ranges from applications like games (using game-theoretic strategies), resource allocation in an operating system or a distributed system, to role-based access control frameworks. We use logic because it provides formal correctness metrics, and also makes the protocol general-purpose in a way that others are not. Matching predicates through unification and having rules stated in the form of Horn clauses provides a model powerful enough to express constraints on resource allocation, game moves and access control policies. For example, a distributed resource allocation framework would have goals expressing resource needs in a quantitative manner, and the rules would be expressed in terms of the amount of resources available. The operations to be performed are straightforward quantitative comparisons. The resource allocator would have to be re-engineered to handle other scenarios where resolution does not involve quantitative comparisons.

### **8.2.2 Centralized Policy Resolution Using an Oracle**

Performing policy resolution using the backward-chaining procedure is straightforward when global knowledge is available. Such resolution is performed by an AI knowledge-based system or expert system (such as theorem provers, Prolog, and Datalog) by generating search trees. These systems make wide use of the *unification* procedure, through which goal queries can be matched against facts and predicates within rule

bodies. But all of them are based on a single database, the contents of which are completely visible to the query processor. If a centralized entity, or an *oracle*, had complete access to the goal and policy sets of two interacting ubiquitous computing domains, it could run a backward chaining algorithm, starting with the goals of both domains at the root of the search tree. The two policy sets would be combined into a single policy database, after suitable reformatting of the individual rules. Since rules and facts typically refer to beliefs and rules held by the owning domains, conflict would be unlikely. There may still be global beliefs that could contradict. Self-contradicting rules would be eliminated in the process of merging. Given this merged database, the oracle assumes as its goal an expression that is the logical conjunction of all the goals in the two goal sets and formats it into a Prolog query. All valid solutions to the combination of goals are obtained from the leaves of the generated search tree, an example of which is illustrated in Figure 20.

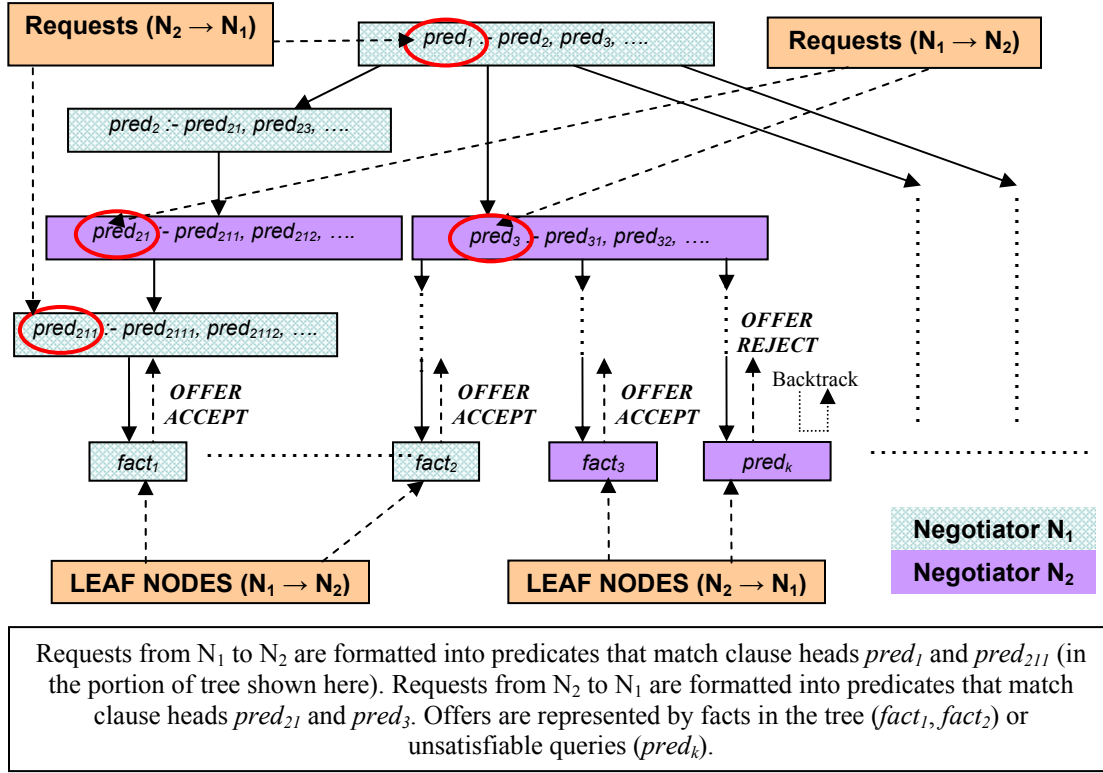
Because an oracle can perform an exhaustive search through the database, it can find all possible solutions, and thereby infer what the best solution is. The running time is exponential in the number of policies. If cycles are present, depth-first search may not terminate; otherwise, the maximum size of a search tree will always be bounded (the number of nodes in the tree =  $O(s^r)$ , where  $s$  is the maximum number of predicates in the body of a clause, and  $r$  is the number of facts and rules).

Considering only the quality of the result, an oracle is our gold standard for policy resolution, and any other framework must be evaluated against it. Its efficiency in terms of the time taken is still open to question, and we present sample performance numbers in

Chapter 8. In real life, using such an oracle is not practical or desirable because domains need to keep their policies private.

### **8.2.3 Visualization of Negotiation as a Distributed (or Decentralized) Search Tree**

Figure 20 illustrates how an oracle would resolve goals and policies by running a backward-chaining procedure with the given goal set as argument on the given policy database. A negotiation simulates this procedure and generates a similar search tree, which is different from the oracular tree in one respect: the nodes are distributed among the negotiators. As indicated in Figure 21, some nodes in the tree are processed at domain  $N_1$  and the rest at domain  $N_2$ . The transition from a node of one color to a node of another color indicates an unsatisfied constraint at the former, which is communicated in the form of a request to the latter. The contents of the nodes reflect the private policies of the respective domains. Leaves represent validated constraints at a domain, and could also represent an acceptance offer. Backtracking in this tree represents failed requests and declined offers. The more available alternative requests there are, the more the tree diverges (i.e., the average branching factor, or breadth, of the tree increases).



**Figure 21.** Distributed Search Tree Representing Policy Resolution through Negotiation

#### 8.2.4 Quality of a Negotiation

The quality of a negotiation depends both upon the nature of the result, or agreement, reached, as well as the nature of the process that led to that result. We stated in Chapter 3 that the result consisted of a set of bi-directional resource and service offerings that would be a subset of the goals the negotiation started off with. The assignment could range conceptually from the null set to complete and exact satisfaction of the goals. The *result quality* varies according to the extent to which the result matches the goal set. The quality, or efficiency, of the process varies according to the length of negotiation, which can be measured both by the total time taken and the number of steps.



**Quality Metrics:** We define three metrics for evaluation of the negotiation protocol:

1. *Correctness*: A negotiation protocol is defined as correct if the result, which is a mapping from the goal set to the level of satisfaction (either modal: *true/false*, or an alternative of lower utility compared to the original goal), is an improper subset of the oracular result, and is also consistent with the policies of the negotiators. This was stated in Chapter 3 as part of the negotiation model, as follows:

- $grant-access(D_2, Q_2) \wedge P_{11} \wedge P_{12} \wedge \dots \wedge P_{1m}$ , and
- $grant-access(D_1, Q_1) \wedge P_{21} \wedge P_{22} \wedge \dots \wedge P_{2n}$

Here  $(Q_1, Q_2)$  represents the result, are not logical contradictions. The result set is a subset of the oracular result in the following ways:

- If a goal  $G$  maps to *true* in the negotiated result set, it must map to *true* in the oracular result set.
- A goal  $G$  that maps to *true* in the oracular result set may map to *false* in the negotiated result set.
- A goal  $G$  that maps to a result  $Q$  in the oracular result set may map to a result  $Q'$  of strictly lesser value (or utility) than  $Q$  to the requestor.

As we can see, *correctness* is a rather weak metric, and is the most basic property that our negotiation framework must possess. We exclude the *trivial result* from our correctness definition, however. A trivial result would be the null set, where neither negotiator satisfies any goal of the other even though their policies are consistent with some or all of those goals. A correct negotiation would have to generate a non-null result, given that an oracle generates a non-null result.

2. *Completeness*: A negotiation protocol is defined as complete if it always generates a result comparable to that an oracle would generate, given the same goal and policy sets as arguments. The negotiated result need not be identical to the oracular result, but must be equivalent to it in the following ways:

- A goal  $G$  that maps to *true* in the oracular result set must map to *true* in the negotiated result set.
- A goal  $G$  that maps to *false* in the oracular result set must map to *false* in the negotiated result set.
- A goal  $G$  that maps to a result  $Q$  in the oracular result set must map to a result  $Q'$  of equal value (or utility) than  $Q$  to the requestor.

Based on the above definition, a complete negotiation protocol is *correct*.

If multiple goals/requests are provided as input to a policy resolution procedure, and the satisfaction of a subset of these goals conflicts with the satisfaction of a different subset, a variety of results may be achieved. This scenario deserves a deeper analysis. For example, consider a scenario where negotiator  $N_1$  poses requests  $R_1$ ,  $R_2$ ,  $R_3$ , and  $R_4$  to negotiator  $N_2$ . Due to the way  $N_2$ 's policies are framed, it can satisfy one of the following request sets:  $\{R_1\}$ ,  $\{R_2, R_3\}$ , or  $\{R_2, R_4\}$ . Each set contains entries that don't conflict with each other but do conflict with elements in a different set; i.e.,  $R_1$  conflicts with all the others,  $R_2$  conflicts only with  $R_1$ ,  $R_3$  conflicts with  $R_1$  and  $R_4$ , and  $R_4$  conflicts with  $R_1$  and  $R_3$ . One could jump to the intuitive conclusion that a best result is the request set of maximal cardinality; i.e., the policy resolution procedure is complete only if the final result is  $\{R_2, R_3\}$  or  $\{R_2, R_4\}$ .

but not  $\{R_1\}$ . But since the protocol does not know beforehand what the relative values of  $R_1$ ,  $R_2$ ,  $R_3$ , and  $R_4$  are to  $N_1$ , this is not an objectively best measure of completeness. Though it may be a useful yardstick to measure negotiation protocols by in a particular class of scenarios, such protocols would have to be modeled and designed in a different way (with extra preference inputs) than the one we have. On the other hand, if a protocol were to satisfy only the set  $\{R_2\}$ , we could definitively declare it to be incomplete, irrespective of the class that scenario belongs to. Therefore, we consider a negotiation protocol to be complete if it generates one of the non-conflicting sets. In our example, a complete negotiation would generate one of the sets  $\{R_1\}$ ,  $\{R_2, R_3\}$ ,  $\{R_2, R_4\}$ . We will see how our protocol measures up to these standards in the Section 8.2.5.

3. *Optimality*: A negotiation protocol is considered to be optimal if it always generates a result that is identical or comparable to the oracular result in the minimum number of steps, or using the least number of messages. An optimal negotiation is *complete*, based on the above definition with this added constraint of being maximally efficient.

These metrics are related to each other through the following inequality, where the relation  $A < B$  indicates that metric  $A$  is less stringent than metric  $B$ , and also that if the protocol measures up to metric  $B$ , it will measure up to metric  $A$  by implication.

$$Correctness < Completeness < Optimality$$

By definition, an optimal negotiation protocol will always be complete and correct, and a complete protocol will always be correct.

### 8.2.5 Metric Evaluation of Negotiation

**Correctness:** The trivial correctness property of the negotiation protocol is evident from the two following properties that have already been proved:

- A negotiation result is consistent with, and does not violate, the collective policies of the negotiators. This is proved by our use of Prolog to obtain answers through queries. Prolog semantics are logically correct. Since we avoid predicates that would require occur-checks (see Chapter 4) in Prolog unification, any queries run by our policy manager are guaranteed to return correct results. Specifically, any affirmative offer is made only when the policies governing that offer can be provably satisfied by running a Prolog query. Therefore the individual negotiation steps are guaranteed to be correct.
- The protocol is guaranteed to terminate, as the policy databases have finite size, the length of each policy statement is bounded, and there are no policy cycles.

We can also show that the protocol is non-trivially correct. The procedure, by inferring alternative sets of counter requests, exhaustively examines the search space. If a solution exists, it will be eventually found. Multiple invalid alternatives may be examined before a satisfactory one is followed, but if such an alternative exists, it is guaranteed to be examined and a correct result generated.

We still need to reexamine the issue of intermediate requests and offers and the effect they will have on the overall result. Affirmative offers are registered in the policy databases of the negotiators in the form of Prolog facts; the resulting state changes may have effects on the remainder of the negotiation. Can an intermediate offer made during the examination of a failed alternative prevent a subsequent alternative from succeeding? This could happen if two requests made in two different alternative sets (under different subtrees in the negotiation policy tree) happen to be related through one or more policy constraints at the receiver's end. To state this problem in more concrete terms and in the simplest form, a negotiator receives a request  $R$  and generates counter-request sets  $C_1$  and  $C_2$ . A successful negotiation would result if  $C_2$  were to be explored first, whereas a failure would result if  $C_1$  were to be explored first. Now  $C_1$  is partially satisfied by the other party, i.e., some of its requests are satisfied while others are rejected. But the satisfied requests are related to some other requests in  $C_2$  (explored after the failure of  $C_2$ ) in such a way that now  $C_2$  cannot be satisfied in its entirety, resulting in a negotiation failure. As we will see below, we can handle some of these situations, but not all of them.

All situations in which offers are revocable can be handled. Examples of these are the following: if the requests in  $C_1$  and  $C_2$  involve granting access to disk space, and the receiver of the request has a finite amount of disk space, it may not be able to satisfy the request in  $C_2$  after it has already conceded access to some portion of its disk space in  $C_1$ . Another example: requests in the two alternative sets are related through a policy involving opposite modalities; i.e., granting one request is incumbent upon not granting

the other, and vice-versa. The underlying concept is the same: attempting to satisfy both requests results in a logical contradiction.

We can make the negotiation protocol correct in these situations. All we need to do is to track the intermediate requests that have been satisfied, and simply revoke them upon failure and backtracking. If request  $R_2$  is sent in response to request  $R_1$  (received earlier),  $R_2$  contains a reference to  $R_1$ ; hence this tracking is straightforward. We have not currently implemented this feature, but it can be added with minimum difficulty. But in practice, this scheme is not foolproof. Not all offers are revocable (though the odds would improve if negotiation used a heuristic that gives revocable alternatives higher priority). If offers are simply promises to abide by obligations, or promises of future access to resources, or the performance of an action (e.g., starting or stopping a service), revocation is straightforward. Revocation is accompanied by the removal of policy statements from the database, returning it to the pre-offer state. But there may be offers that cannot be revoked. A private piece of information, once given, cannot be un-given. Sending a file (as a string of bytes) is another irrevocable action; this may occur in the *smart party* scenario where media files are transmitted from one domain to another. Thus negotiations may fail in these scenarios because of a bad alternative selection, even though it is potentially successful. Theoretically, the rollback added to negotiation results in a correct procedure if we consider rollback to be equivalent to a reversion of the policy database to its original state. But revocation also involves non-logical operations; hence it may not yield a non-trivial result in some scenarios.

**Completeness:** Our negotiation protocol attempts to satisfy the original (maximal) demands first before falling back to alternatives (in the form of alternative offers). It generates all possible counter-request combinations at every step, and exhaustively examines the search tree. A number of equivalent *best* solutions may exist, and a negotiation will result in the first encountered one being picked. Hence, the correctness of negotiation also implies completeness. As we have shown, negotiation is correct in scenarios where offers are revocable, and trivially correct in all the remaining scenarios. Likewise, negotiation is complete in the former class of scenarios and incomplete in the latter class.

When negotiation starts with multiple requests (refer to the *completeness* metric discussion in Section 8.2.4), our negotiation protocol is complete in the more general sense, where one of the several non-self-conflicting request sets is granted as the final result. Referring back to the example, the negotiation protocol will generate one of the following results:  $\{R_1\}$ ,  $\{R_2, R_3\}$ , or  $\{R_2, R_4\}$ . Which of these it will generate depends entirely on the order in which the requests are evaluated, and the way the negotiation plays out. But the protocol is definitely guaranteed to generate a maximal cardinality result if every maximal cardinality result set dominates (is a super-set of) another valid result set. If we exclude  $R_1$ , our protocol is guaranteed to return  $\{R_2, R_3\}$ , or  $\{R_2, R_4\}$  and will not return just  $\{R_2\}$ . For example, let  $R_1 = \{Access\ to\ local\ file\ system/repository\}$ ,  $R_2 = \{Permission\ to\ run\ networked\ applications\}$ , and  $R_3 = \{Access\ to\ 'x'\ amount\ of\ disk\ space\}$ .  $N_1$  may end up getting either access to the repository, or disk space plus the permission to run networked applications. The protocol does not know what value either

negotiator has assigned to  $R_1$ ,  $R_2$ , and  $R_3$ , and so it could generate either result. But any negotiation that results in  $N_2$  granting  $R_2$  will also result in  $N_2$  granting  $R_3$ . Therefore, our protocol is complete even if it results in  $N_2$  granting only  $R_1$  to  $N_1$ .

This looks non-intuitive at first glance; if the alternative to granting  $R_1$  were a set of ten other requests (instead of two), it may seem obvious that a protocol that resulted in the set of ten requests would be superior. But just taking numbers (or set cardinality) into account is a flawed approach of measuring success, in our opinion. In our example, gaining access to the local file system may be very valuable to  $N_1$ , and correspondingly more risky to  $N_2$ . Perhaps that is why the policies were set up in a way that would cause conflicts to occur among the sets  $\{R_1\}$  and  $\{R_2, R_3\}$ . *Note:* the protocol does not know at the beginning what sets are conflict-free; these sets are discovered through the negotiation process. The way a negotiation plays out completely depends on the policies of the negotiators. Therefore, the resulting sets roughly reflect an equal distribution of the weights assigned to them by the request granter (ten rather minor requests may be cumulatively as important as one separate unrelated request.)

Consider a second example: In our conference room scenario (see Section 7.2), both the attendee and the conference room may consider the granting of network access to be far more important than printer or display services. Therefore, if  $R_1 = \{Network\ access\}$ ,  $R_2 = \{Printing\ service\}$ , and  $R_3 = \{Display\ service\}$ , and we had the same conflict sets  $\{R_1\}$  and  $\{R_2, R_3\}$ , it would turn out to be more advantageous to  $N_1$  to get only network access. Therefore we do not consider the cardinality aspect in our completeness metric, and by our defined standards, negotiation is complete.



Precedence ordering among goals can be set to ensure that certain requests are given higher priority than others. Referring to the peer-to-peer file sharing example in Section 7.4, the request that held the highest precedence among all desired requests was selected first and negotiated for; only upon failure were other requests attempted. In that example, the precedence relation was explicitly described through policy rules in our language, and that guided negotiation strategy. However, if multiple requests are made simultaneously, our protocol assumes that there is no inherent ordering among them, and runs a best-effort procedure, generating the kinds of results shown in the above examples.

Still, certain classes of scenarios may consider the cardinality metric to be of paramount importance; in such scenarios, a protocol that does not guarantee a maximal cardinality request set would be considered incomplete. Let us see how we can modify our protocol to make it complete for this class of scenarios. A *delayed satisfaction* approach can be used. If a goal is found to be satisfiable, it is not immediately granted as an offer. That goal is retained in the received requests list (maintained by the controller), and an offer is postponed until the satisfiability or unsatisfiability of all the other goals can be inferred. At this point, we have a set of goals for which affirmative offers can be sent (assuming independence with other similarly satisfiable requests) and others for which negative offers will be sent. The controller then runs queries examining which subsets of the satisfiable request set are non-self-conflicting (i.e., can be satisfied simultaneously). Such a procedure can be optimized so that we don't actually need to run a query for each possible subset. From this collection of satisfiable request sets, we can pick a maximal cardinality set to return (through affirmative offers) to the other party.

**Optimality:** Theoretically, we cannot provide any optimality guarantees. Decentralized policy resolution, as performed by negotiation, is a best-effort procedure. A number of alternative options are available at every step, and in the absence of complete information, the probability of selecting the best alternative cannot be guaranteed to be unity. Currently the protocol is engineered to pick the first available alternative, and the remaining ones are saved. The ordering of the generated alternatives depends on the implementation of the SWI-Prolog subsystem, and the order in which individual Prolog facts and rules are read into the database. To a policy manager, the nature of the ordering is transparent; therefore, the selection process is effectively random over a large number of cases.

We can use heuristics to make more intelligent selections. The one that we use simply orders the alternative request sets (returned by the counter-request generation algorithm) by the cardinality of the sets. The logic behind this is that the fewer requests posed, the fewer computations the opposite party needs to make, and this may result in a shorter negotiation. Though this may work well in a large number of cases, there are obvious scenarios under which it fails. We can understand this by viewing the policy resolution tree with varying branching factor and depth. A subtree rooted at the node corresponding to the original request could have two children, one with a small branching factor (request set will be small) and a large depth, but another subtree could have a large branching factor (request set will be large) but be quite shallow. In this case, it would have been better to examine the latter subtree first. But it is impossible to estimate the depth of the tree without having more knowledge of the opposite party's policy rules. The

branching factor at the immediate level of the tree is known, and a negotiator can only try to make an intelligent guess based on that knowledge.

Other heuristics can be used if one knows what the other's constraints are. For example, in the interest of maintaining security and privacy, many domains would be far more willing to turn off certain vulnerable applications, or agree to obligations dictating how it could use its sound and display devices, as compared to allowing intrusive access to its system, providing private information, or giving access to resources when the access is hard or impossible to undo. Therefore, an alternative drawn from the former would be more likely to lead to an agreement through a shorter negotiation. Similarly, if alternatives differ only in quantitative terms (such as required bandwidth), and a range of values is acceptable, a lower value request would likely result in a shorter negotiation.

An oracle, having the advantage of complete knowledge, can determine an ideal result, and also (by examining the search space) the least-depth tree that would reach that result. It generates this tree by always selecting the correct alternative. The complexity of this search and the tree construction is exponential. This least-depth tree is equivalent to the most optimal negotiation that could be conducted.

**Comments on Complexity Analysis:** This research was conducted primarily from a practical system design viewpoint. We wanted to both establish design principles and implement a framework that could be applied in a wide range of real-world scenarios. Theoretical rigor was not a primary goal or motivation, though we did want to investigate and establish the limits of the negotiation protocol, as we have attempted to do in this

chapter. As we made progress in our research, we also made decisions and reached conclusions about the nature of the policy language and the nature of the negotiation mechanism, all of which have been described with justifications in the previous chapters. Describing policies in logic provides various advantages, most prominently domain independence and application generality. But this makes our framework similar to an expert system that necessarily uses search trees as the basis for its algorithms. For all the advances made in Prolog compiler designs, search trees are ultimately generated, the theoretical complexity of which are inherently exponential. Various researchers have attempted to analyze the complexity of Prolog programs, or any logic programs based on Horn clauses [Aarts1995a; Aarts1995b], and they typically use the depth of the search as a parameter. We will revisit this metric in Chapter 9 when we describe our testing procedure, but we must emphasize that search depth is only one factor, and the real complexity is exponential in the value of this depth. We also must take into account the branching factor of the trees. The key point to note is that the complexity of our negotiation system will necessarily be exponential, which is an unflattering result in theory but does not detract from the usability of the system in practice (much as most Prolog and Datalog programs provide adequate performance). Also, the number of parameters and variables are fairly large; these include not only the branching factor and depth of a search tree but also the number of variables in a predicate, the number of terms in a clause and the number of related variables among the predicates in the body of a clause. We do not attempt to do a heavy theoretical analysis here, since we know a priori

that the complexity will be exponential, and from a practical system viewpoint, it does not shed any real light.

Still, we can try to make some rough estimates of the bounds on the number of negotiation steps, given certain assumptions. One question we could ask is: What is the maximum number of steps a negotiation can take in a given scenario? We can infer a simplistic bound for this. If we assume that the examination of a particular policy rule yields at most one set of counter-request alternatives, and we have already established that each policy rule is examined at most once during the course of a negotiation, the maximum number of steps is linearly related to the number of policy rules. Specifically, if the number of rules in the two databases is  $R_1$  and  $R_2$ , each possible alternative is examined, and some yield affirmative offers and some yield negative ones. Therefore, each rule corresponds to two steps (a request and an offer), so the length of the negotiation  $\leq 2(R_1 + R_2) + 1$  (adding one step for the termination message). We have made a simplifying assumption here: the number of request sets generated by examining a policy rule may be more than 1; the exact value depends on the number of facts in the policy database. If we can bound the number of sets generated to  $k$ , the resulting upper bound would be  $2k(R_1 + R_2) + 1$ , which is still linear in the collective size of the databases.

Can we also get an idea of the average number of steps taken by a negotiation? We revisit the search tree for this purpose. Let our search tree have a uniform branching factor  $b$  and uniform depth  $d$ . The branching factor indicates the number of available alternatives at each node (where a policy is being examined). We assume that all the nodes at a particular level consist of nodes examined at one end or the other, and all

backtracking takes place at the lowest level (maximum depth). The number of nodes is linearly related to the number of negotiation steps (roughly equivalent to half the number of negotiation steps). Therefore, the number of nodes ( $N$ ), which is  $1 + b^2 + b^3 + \dots + b^d = (b^{(d+1)} - 1)/(b-1)$  can be used as a measure of the number of steps in the worst case. If we assume that a unique best alternative exists in each set, and each alternative is equally likely to get selected, then roughly half the number of alternatives ( $b/2$ ) is examined on average. Our search tree on average (assuming the same uniform depth  $d$ ) now examines  $((b/2)^{(d+1)} - 1)/((b/2) - 1)$  nodes (say  $N'$ ). If we define  $R = N'/N$ : as  $d \rightarrow \infty$ ,  $R \rightarrow 0$ ; i.e., the average number of negotiation steps grows much slower compared to the maximum.

### 8.3. Other Protocol Properties and Issues

#### 8.3.1 Privacy Maintenance

One of the motivations and an assumption of this research was that the local policies of domains must be private by default and not known to their opposite negotiators at the beginning of a negotiation session. We therefore analyze whether, and to what extent, negotiations are able to maintain the privacy of their participating domains' policies.

As we have seen, a negotiation (or decentralized policy resolution, as we define it in this chapter) is a sequence of information exchanges between the two entities. The pieces of information could be requests or offers (often accompanied by non-logical objects as evidence). Requests are predicate strings accompanied by descriptive predicates; they are also components of policy rules in the database of the requestor, extracted using the counter-request generation procedure. The act of sending a request

therefore reveals partial information about policy, since the recipient of the request can associate the request it sent with the counter-request it received. If a bunch of requests are sent in one message, a counter-request received can only be associated with a request sent with a certain probability. For example, if a domain sends requests  $R_1$  and  $R_2$ , and receives counter-requests  $R_7$  and  $R_9$  in reply,  $R_7$  may have been a counter to either  $R_1$  or  $R_2$ . In our implementation, a negotiation message containing a counter-request contains a reference to the original request (see Chapter 5), but this is only for the purpose of printing out an explanatory message if a negative offer is made. This reference may be omitted if a domain is more interested in privacy than in tracing reasons for failure.

Affirmative offers also result in some privacy loss, as the recipient of the offer now knows that the sender's policies allow satisfaction of the request. Also, any accompanying object or proof appended to the message is also private information. Negative offers do not yield any useful information, since the recipient cannot determine why the sender denied its request.

In real scenarios, not all information releases carry equal weight. But the importance attached to each piece of information and to every transacted object is subjective, non-logical, and dependent on the priorities of the negotiators. For the purpose of analyzing negotiations without restricting ourselves to specific scenarios, we have to assume that all such releases carry equal weight for the purpose of measurements. In Chapter 9 we will see to what extent negotiations maintain privacy on the average.

The extent of privacy maintenance of a negotiating domain can range from 0% to 100%. Negotiators start off with 100% privacy, but this percentage gradually reduces

during the session. A negotiation that maintains 100% privacy for both its negotiators is useless and trivial, since it generates a null result. If a negotiation must result in something useful, some privacy loss is inevitable. An oracle falls at the other extreme, since the parties (or one of them) completely surrender their privacy, though an optimal result is achieved. In practice, though, optimal negotiations are better than oracular policy resolutions, since they maintain privacy at a level between 0% and 100% while producing the oracular result. This is simply because an optimal negotiation contains the least number of intermediate (or counter) requests, and consequently the least number of affirmative offers. Every failed alternative attempted before the right alternative was explored results in more information released than would be absolutely necessary to achieve the same result as the optimal negotiation.

We can define a quantitative metric for privacy loss. A policy database consists of a set of facts and a set of policy dependencies. Each policy rule is a set of policy dependencies (whose size is equal to the length of the rule body). Release of knowledge of any dependency constitutes privacy loss, as does the release of the knowledge of any fact. Policy dependencies are released through posing of counter-requests during the intermediate stages of a negotiation, and facts (either stated or provable) are released when making affirmative offers. Therefore, the privacy loss for a negotiator can be quantitatively estimated to be  $\langle \#intermediate\ requests + \#affirmative\ offers \rangle$ .



### **8.3.2 Limitations of this Negotiation Procedure**

The effectiveness of any negotiation is only as good as the mechanisms that are available to the negotiators. Affirmative offers to requests could take the form of information, objects, or simple promises in many cases. It is incumbent on the requesting domain to have effective verification mechanisms, invoked through helper functions. In the absence of foolproof mechanisms, policies that guard sensitive resources should take into account the level of security assurance provided by available mechanisms. For example, an offer of a piece of mobile code is inherently dangerous, since the code may exploit some flaw in the host's operating system to cause harm. A mechanism employed by the receiver could be as follows: if the code carries proof that can be verified [Necula1997], it is accepted, and otherwise the offer is rejected. Even proof-carrying code is not completely foolproof, so policies may contain extra constraints that refer to the identity or affiliation of the domain that is supplying the code.

Similarly, affirmative offers framed as promises to abide by the terms of the request may sometimes have to be taken at face value if the negotiation must yield a result. In practice, ubiquitous computing researchers have recognized the difficult nature of this problem, and have suggested a combination of access control and a trust network that is dynamically reevaluated through constant monitoring of events and actions [English2003; English2004]. Interactions between domains are monitored to make sure that the domains do not break their promises. If a domain violates a promise, others lose trust in it, and may not accept its promises in future interactions. As we have seen earlier, the utility of the goals that are designed to be met through requests also plays a part.

Alternative request sets could be ordered based on heuristics that take into account both the utility of the requested information/object and the level of trust in the negotiator and locally available verification mechanisms. These heuristics would be logically separate from the policies in the database. Such trust and utility heuristics could also be used by a domain considering whether or not to send a piece of information or a guarded object in the form of an offer, since the posing of a counter-request contains an implicit promise of fulfilling an earlier request.

### **8.3.3 Security Properties of the Negotiation Protocol**

We try to analyze the security of communication and of the communicators that participate in a negotiation. The following standard security properties can be analyzed: authentication, non-repudiation, integrity, confidentiality, and availability.

The process of negotiation itself takes care of authentication, since proof of authenticity in the form of keys (or similar non-forgable information) can be requested and obtained as part of a negotiation, assuming that suitable policies are set up. Likewise, non-repudiation could be achieved if desired by the negotiators signing and verifying transacted objects and keeping records of negotiation. Neither of these properties are essential features of negotiations. In ubiquitous computing scenarios, domains may agree on the sharing of resources and information without a high level of security assurance.

We use existing secure communication technology to ensure confidentiality and integrity. The Panoply platform allows cross-network communication only through secure TLS sockets. Events, including those containing negotiation messages, are

communicated over these TLS connections. We could use TLS or any other secure communication mechanism if we had to implement negotiation independent of the Panoply middleware.

When we consider availability, we are primarily concerned about denial-of-service (DoS) attacks that take advantage of the protocol dynamics. Is it possible for a domain to prevent another from carrying out its normal operations through malicious uses of negotiation? We showed earlier in this chapter that the protocol is guaranteed to terminate, and is free of deadlocks and livelocks, so a malicious entity that is nonetheless playing by the rules of negotiation cannot keep another entity indefinitely busy. But it is definitely possible for a malicious entity to keep inventing dummy requests and counter-requests that have no purpose but to keep its counterpart negotiating indefinitely. For a single thread, this kind of a DoS attack is not as serious as say, ping flooding, because negotiation messages are exchanged synchronously. With a malicious entity initiating multiple negotiations with the target, a more effective DoS attack could be mounted. We cannot avoid this in a theoretically complete way, but there are many practical measures one could use to thwart such attacks. For example, a domain could choose to have a limit on the number of steps it is willing to undertake in a negotiation. As each party has the right to send a termination message at any point, a negotiation that has gone on for too long and has used up too many resources could be terminated.

## Chapter 9

### Performance

In this chapter we evaluate our policy management and negotiation framework from the viewpoint of its real-world performance. We state and analyze the results of our measurements, attempting to leave readers with a good of idea of what to expect if they use the negotiation protocol or other policy management features for their applications.

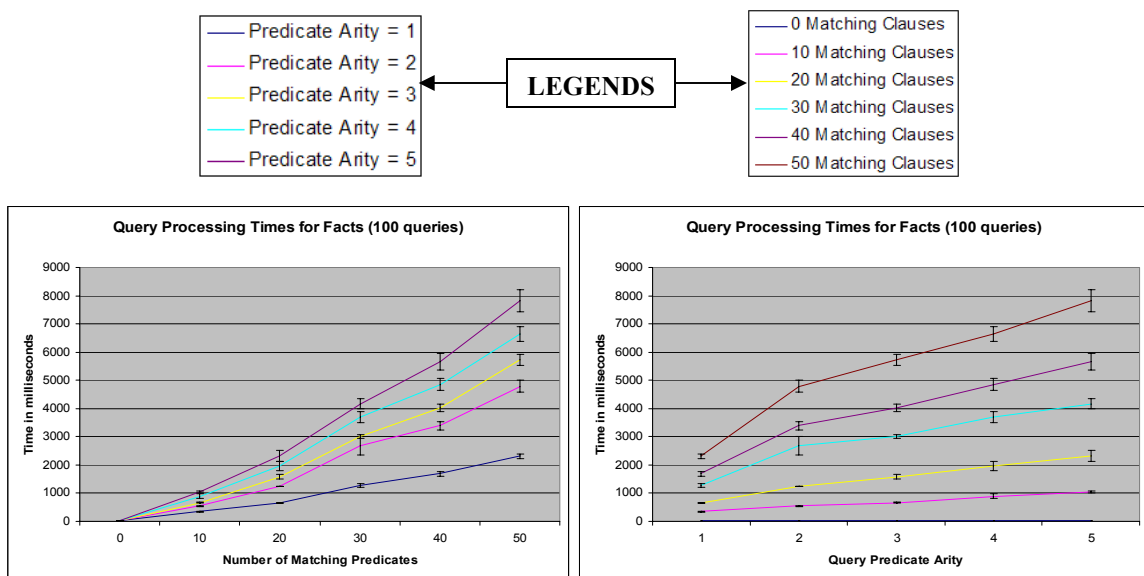
#### 9.1. Overhead of Commonly Invoked Policy Engine Functions

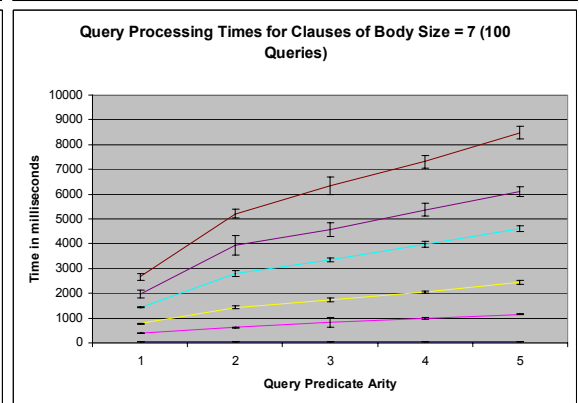
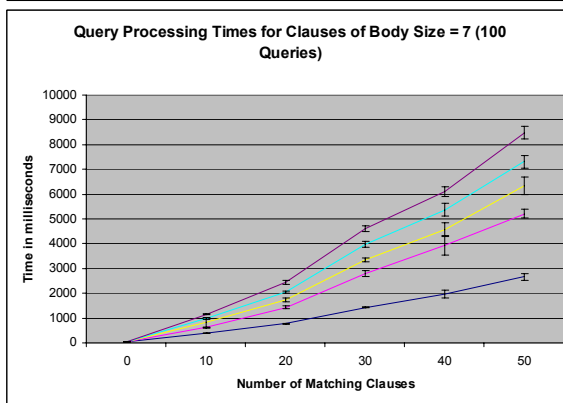
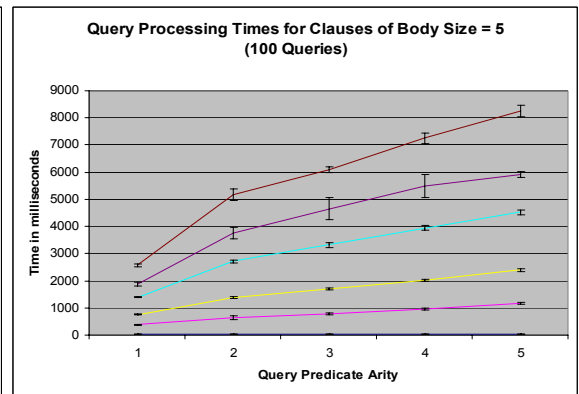
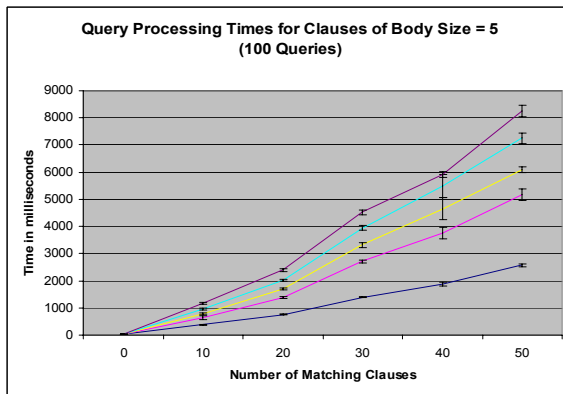
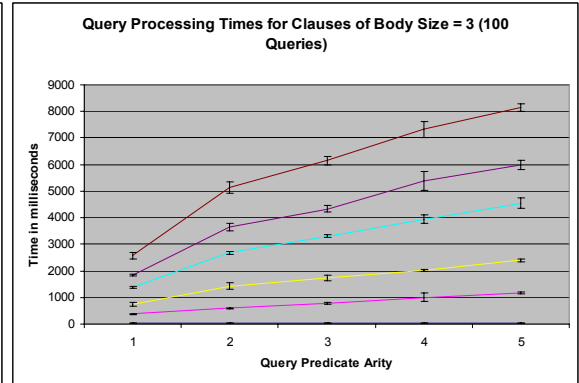
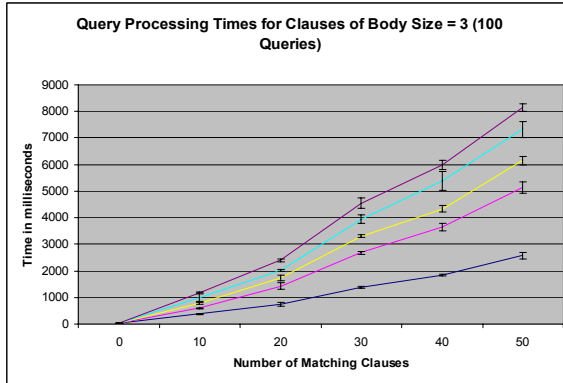
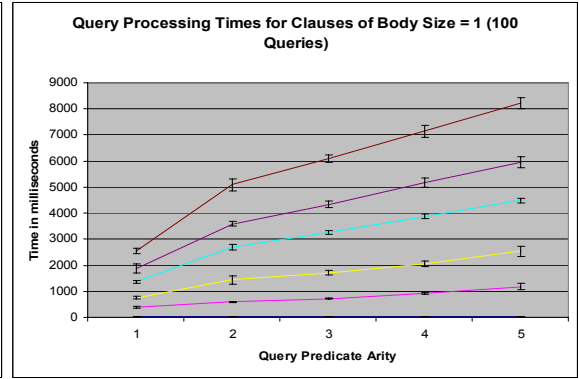
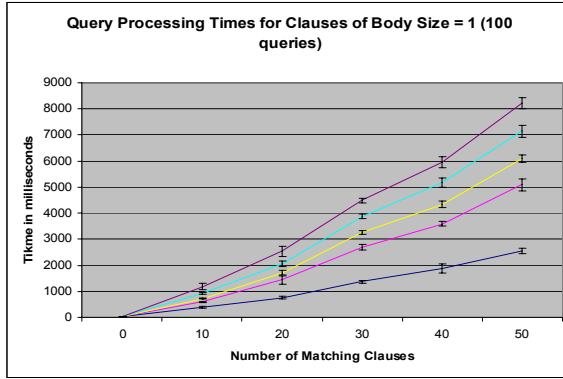
We described the key operations offered by the policy engine through an API in Chapter 4. These operations invoke basic SWI-Prolog functions through the JPL (Java-to-Prolog) interface. These operations do not just translate the SWI-Prolog output into Java objects and data types, but perform additional tasks for the purpose of formatting the results and maintaining the integrity of the policy database. Our measurements below indicate how fast such operations typically are. All measurements were performed on an IBM Thinkpad T42 (1.7 GHz, 512 MB) laptop running Fedora Core 3 Linux.

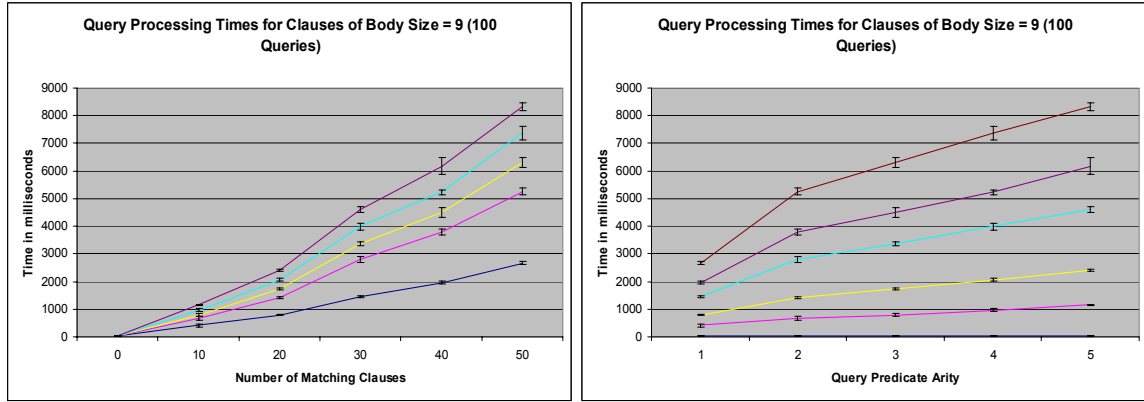
**Running Database Queries**—SWI-Prolog offers a simple query procedure that returns all valid variable bindings. Our query procedure performs additional tasks: i) checking whether or not the predicate names are *special* predicates, and ii) formatting each query string in such a way as to catch exceptions rather than letting the program crash. Also,

Prolog backward chaining may return (depending on the nature of the policies) duplicate sets of variable bindings. In practice, we found a huge number of such duplicates being returned, which forced us to add duplicate set elimination in the query procedure. This operation incurs significant overhead, but this additional overhead is still lower than would be incurred if the problem of duplicates was deferred to the calling procedures.

Since the time to run each query is very small (milliseconds or less) and the variance is large, we report the time taken to run bunches of 100 queries each. First, we asserted a set of facts and clauses in the Policy Engine database, the facts and heads of clauses being predicates of arities varying from 1 to 5, and clause bodies containing 0 to 9 predicates. The policy databases we used in our applications and tests lay within these bounds, and we believe these numbers are representative of real policy databases. Even if some databases contain policies that fall outside these bounds, reasonable query processing overhead estimates can be projected from the charts (shown below in Figure 22). All numbers are reported with 99% confidence intervals.







**Figure 22.** Query Processing Times along Various Dimensions

What we can observe from the graphs in Figure 22 is that the query processing times seem not to vary at all when the sizes of the clauses (number of predicates in the bodies) change. On the other hand, the times vary significantly with predicate arity and the number of possible solutions. The processing time appears to increase either linearly or in a gentle exponential way with an increase in the number of matching facts or clauses. This is expected, given the way the backward chaining procedure works. With an increase in arity of the query predicate, the rate of increase in processing time seems to decrease (or equivalently, a roughly logarithmic increase in processing time) or remains constant. This variation (linear increase in processing time) is consistent with the nature of the unification algorithm for matching predicates, which primarily affects performance in this case. Therefore, our results above seem to roughly meet our expectations.

**Table 2.** *Variation of Query Processing Times*

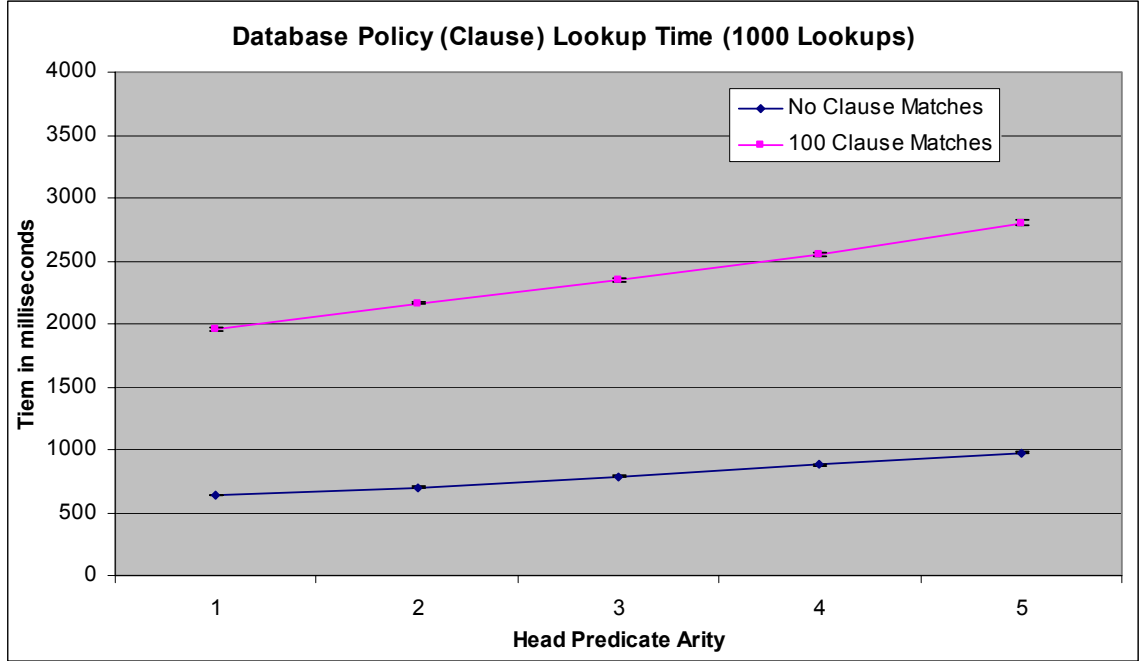
		Predicate Arity				
		1	2	3	4	5
Number of Matching Facts/Rules	0	25.25 $\pm$ 0.13	27.61 $\pm$ 0.85	30.69 $\pm$ 2.86	31.39 $\pm$ 0.60	33.31 $\pm$ 0.77
	10	385.32 $\pm$ 29.51	610.15 $\pm$ 51.34	757.82 $\pm$ 65.93	953.06 $\pm$ 45.68	1145.59 $\pm$ 50.97
	20	745.48 $\pm$ 53.66	1393.31 $\pm$ 86.47	1699.44 $\pm$ 74.55	2031.94 $\pm$ 44.56	2418.72 $\pm$ 78.98
	30	1383.58 $\pm$ 72.90	2727.46 $\pm$ 63.91	3272.64 $\pm$ 146.99	3910.80 $\pm$ 130.17	4487.28 $\pm$ 188.58
	40	1871.76 $\pm$ 116.17	3682.65 $\pm$ 213.38	4399.40 $\pm$ 253.86	5250.24 $\pm$ 256.75	5965.99 $\pm$ 199.60
	50	2555.63 $\pm$ 150.66	5110.94 $\pm$ 194.99	6118.54 $\pm$ 253.82	7173.34 $\pm$ 314.04	8208.24 $\pm$ 249.30

Table 2 summarizes the chart results (see Figure 22) by computing the average processing time for each  $\langle predicate\ arity, \#matching\ clauses \rangle$  pair. The most heavy operation here (predicate of arity 5 with 60 matching policies) takes  $\sim 8$  milliseconds. Having more than 20 matching facts/rules is extremely unlikely in a real policy database, which would indicate that the typical query processing time would take  $\sim 2$  milliseconds.

**Policy Statement Lookup**—Clause lookup is one of the most commonly performed Policy Engine operations. This is particularly the case in the counter-request and alternative offer generation procedures, where all clauses whose heads match the argument predicate must be examined. The SWI-Prolog meta-predicate `clause(ClauseHead, ClauseBody, ClauseReferenceNumber)` is evaluated with `ClauseHead` replaced by the relevant argument predicate. The Policy Engine gets the list of matching clauses by invoking the JPL *Query* method with a suitable *clause* predicate as argument. Each query lookup operation is carried out in milliseconds or less time. Therefore, we measured the times taken for 1000 lookups for two cases: i) when no matching clauses exist (as a base case), and ii) when 100 matching clauses exist. These prove adequate to make proper judgments, since the number of matching clauses for an



arbitrary predicate of interest in a real database is likely to be much fewer than 100. The trends are illustrated in Figure 23.



**Figure 23.** Clause Lookup Performance

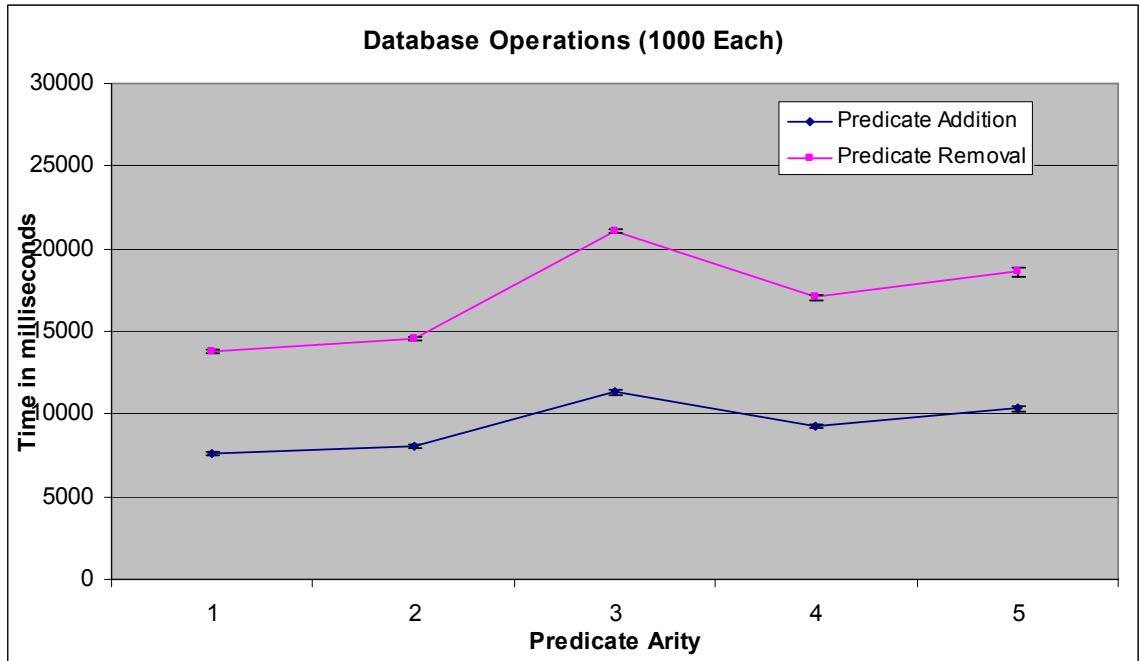
As we can see, the lookup time seems to increase roughly linearly with an increase in arity of the argument predicate. As the lookup procedure utilizes the unification algorithm, this linear increase is an expected result. Table 3 summarizes these results below.

**Table 3.** *Clause Lookup Times*

	Predicate Arity				
	1	2	3	4	5
<b>0 Matching Clauses (<math>M_0</math>)</b>	638.09	700.34	786.92	877.93	973.074
<b>100 Matching Clauses (<math>M_{100}</math>)</b>	1957.2	2164.2	2343	2546.9	2799.2
<b>Ratio (<math>M_{100}/M_0</math>)</b>	3.0673	3.0902	2.9774	2.9011	2.87665

As we can see, lookup time for a predicate when 100 matching clauses are available is ~2.8 milliseconds in the worst case (when arity=5). Therefore, this operation is very efficient from the point of view of the policy manager.

**Database Modification Operations**—We also measured the time taken to add and remove statements from the policy database. SWI-Prolog offers meta-predicates for addition and removal of facts and clauses: `assert`, `retract` and `abolish`. But the policy engine modification operations perform more tasks than simply call `assert` or `retract` queries. First, the predicates in a database fall into two classes: *static* and *dynamic*. The modification predicates only apply to dynamic predicates. To assert or retract static predicates, we need to first `abolish` them and re-assert them as dynamic predicates. Second, both addition and removal operations must be followed by the `chainEventUpdate` method call, which queries all `update` predicates (See Chapter 4). Therefore, these operations are somewhat heavier than they would be if we only had to make a single JPL query. The removal operation also has a higher performance cost than the addition operation. This is because when we remove a policy fact or rule, a simple string comparison to the existing statements in the policy database does not suffice; we must invoke the `identicalTerms` method for this purpose, incurring extra overhead. This is reflected in the graph in Figure 24.



**Figure 24.** Database Modification Overhead

The above graph indicates the performance cost of adding and removing 1000 predicates of varying arities. As we can see, the general trend seems to be a linear increase with increase in arity. When the arity is 3, we observe somewhat anomalous behavior. Informal measurements for arities above 5 indicate a general linear increase, but with this kind of anomalous behavior replicated at certain points. We believe this is an artifact of the SWI-Prolog and JPL implementations, and not reflective of any significant issue in our implementation. The actual values themselves are not too high or too low to merit further investigation. The performance numbers from the above graph are summarized in Table 4 below.

**Table 4.** *Overhead of Database Modification Operations*

	Predicate Arity				
	1	2	3	4	5
<b>Addition</b>	7590.6 $\pm$ 129.71	8095.57 $\pm$ 118.62	11307.8 $\pm$ 145.19	9267.56 $\pm$ 114.26	10336 $\pm$ 152.96
<b>Removal</b>	13779 $\pm$ 107.82	14564.2 $\pm$ 97.92	21097.2 $\pm$ 133.91	17049.14 $\pm$ 163.66	18615 $\pm$ 280.13

As we can see, the maximum overhead is  $\sim 11$  milliseconds for an addition and  $\sim 21$  milliseconds for a removal. These operations dominate event processing times and therefore might be performance bottlenecks. In practice though, such operations will be few and far between, with the highest concentration occurring during a negotiation.

## 9.2. Negotiation Overhead

In the above section we saw how policy engine methods typically invoked during the course of policy management perform on average. Though negotiations also involve a combination of these operations at their core, it would be more illuminating to see how much time is consumed by two typical devices negotiating with each other. We limit our case studies here to fairly short negotiations, but these results can be projected onto longer negotiations. These negotiations also involve helper functions; the overhead incurred by these functions could vary greatly depending on the task being performed. In a later section, we will see how negotiation times vary with the sizes of policy databases and the number of steps, factoring out helper functions completely.

Below we show the results of tests conducted between two different sets of negotiating devices for simple characteristic scenarios. Without perfect synchronization, it is more meaningful to consider the negotiation times at each end separately. The tables with the performance numbers below indicate the times taken to complete negotiation,

from the first message sent or received until termination. Each negotiator spends part of its time processing and sending messages, and the remaining time waiting for a message from its opposite party.

- P is the local processing time
- W is the wait time, and
- Total is the total time (Total=P+W)

Negotiator 1 (N1) initiates the protocol by sending a request message, and Negotiator 2 (N2) receives that message and resumes negotiation. The protocol time measurement starts for N1 when it starts to generate its request message and ends when it either sends out or receives a termination message. The time measurement for N2 starts when it receives the first negotiation message and ends when it either sends out or receives a termination message.  $R(k)$  indicates that  $k$  is sending a request to its negotiator,  $AO(k)$  indicates an affirmative offer sent by  $k$ ,  $NO(k)$  a negative offer, and  $T(k)$  indicates termination. ‘3 C- $R(k)$ ’ indicates three counter-requests sent by  $k$ . In each case, the original request contained a single entry, for *membership*. The two alternative counter-requests posed were as follows. Set 1 contains a single request to set up firewall rules to block a certain port. Set 2 contains 3 requests, one being a request for the identity of N1’s operating system version, another being a request for N1’s current location, and the last being a request for a social voucher previously granted to N1 by N2. Policies at N1 are formulated in a way to reject the request in Set 1 and to accede to the requests in Set 2. Processing in the case of Set 1 involves port scanning, and in the case of N2 involves verifying a voucher’s digital signature.

**Setup 1:** The negotiation was conducted between two spheres, negotiator 1 (N1) running on an IBM Thinkpad T42 (1.7 GHz, 512 MB RAM) laptop and negotiator 2 (N2) running on an Intel P4 (2.53 GHz, 512MB RAM) desktop; both machines ran Linux. Messages were exchanged through a TLS connection running over an 802.11b wireless channel. The results are reported in the table below (and were published in [Ramakrishna2007]).

**Table 5.** *Sample Negotiation Performance Measurements (in milliseconds)*

Case I: R(N1) → AO(N2) → T(N1)

Case II: R(N1) → NO(N2) → T(N1)

Case III: R(N1) → 3 C-R(N2) → 3 AO(N1) → AO(N2) → T(N1)

Case IV: R(N1) → C-R(N2) → NO(N1) → 3 C-R(N2) (alternative) → 3 AO(N1) → 1 AO(N2) → T(N1)

	Negotiator 1 (N1) Timing			Negotiator 2 (N2) Timing		
	P	W	Total	P	W	Total
<b>I</b>	180.5 ± 7.9	1081.2 ± 10.6	1261.7 ± 13	97.5 ± 7.4	1958.8 ± 12.7	<b>2056.3 ± 11</b>
<b>II</b>	8.7 ± 0.4	1124.6 ± 32.7	1133.3 ± 32.7	112 ± 16.1	1897.8 ± 40.8	<b>2009.8 ± 43.3</b>
<b>III</b>	387.8 ± 17.3	4251.2 ± 113.2	4639 ± 121.2	2231.1 ± 110.2	3167.8 ± 38.5	<b>5398.9 ± 121.1</b>
<b>IV</b>	504 ± 24.2	6871.5 ± 367.3	7375.5 ± 376.6	3178.7 ± 356.7	4886.5 ± 46.1	<b>8065.2 ± 370.4</b>

All numbers are reported with 99% confidence intervals. The numbers in the “Wait” and “Total” columns include the message processing overhead introduced by the Panoply middleware, which explains the discrepancy between the processing and total times. Entries in boldface indicate which negotiator’s time dominates the other. As we can see, the most complex negotiation terminates in ~8 seconds, and we can project that additional steps will introduce a small linear increase in overhead.

Our counter-request generation algorithm introduced reasonable overheads, at N2, of  $71.92 \pm 6.6$  milliseconds in Case II,  $1710.5 \pm 81.4$  milliseconds in Case III and  $2573.8 \pm 341.5$  milliseconds in Case IV. Using external methods to verify vouchers took

$13.4 \pm 2.3$  msec, which is small, but running shell commands and executing code will incur larger overhead, as we learned from the QED framework, an early Panoply application [Eustice2003b]. In practice, a few seconds of overhead for negotiation will not be noticed by users in a majority of ubicomp scenarios.

**Setup 2:** The negotiation was conducted between two spheres, negotiator 1 (N1) running on an IBM Thinkpad T42 (1.7 GHz, 512 MB RAM) laptop and negotiator 2 (N2) running on a Sony Vaio MicroPC UX280P (Intel Core Solo U1400 processor, 1.2 GHz, 1 GB RAM); both machines ran Linux. Messages were exchanged through a TLS connection running over an 802.11b wireless channel.

**Table 6.** *Sample Negotiation Performance Measurements (in milliseconds)*

Case I:  $R(N1) \rightarrow AO(N2) \rightarrow T(N1)$

Case II:  $R(N1) \rightarrow NO(N2) \rightarrow T(N1)$

Case III:  $R(N1) \rightarrow C-R(N2) \rightarrow 3 AO(N1) \rightarrow AO(N2) \rightarrow T(N1)$

Case IV:  $R(N1) \rightarrow 3 C-R(N2) \rightarrow 3 AO(N1) \rightarrow AO(N2) \rightarrow T(N1)$

Case V:  $R(N1) \rightarrow C-R(N2) \rightarrow NO(N1) \rightarrow 3 C-R(N2) \text{ (alternative)} \rightarrow 3 AO(N1) \rightarrow 1 AO(N2) \rightarrow T(N1)$

	Negotiator 1 (N1) Timing			Negotiator 2 (N2) Timing		
	P	W	Total	P	W	Total
<b>I</b>	$126.2 \pm 1.8$	$704.8 \pm 35.1$	<b><math>830.9 \pm 35.4</math></b>	$501.5 \pm 18.4$	$164.3 \pm 3.5$	$665.8 \pm 19.2$
<b>II</b>	$30.7 \pm 1.2$	$716.5 \pm 33.5$	<b><math>747.2 \pm 33.3</math></b>	$584.9 \pm 22.5$	$95.3 \pm 12.2$	$680.1 \pm 24.6$
<b>III</b>	$227.0 \pm 2.1$	$1276.1 \pm 22.3$	<b><math>1503.1 \pm 22.2</math></b>	$458.4 \pm 5.5$	$1020.2 \pm 22.7$	$1478.6 \pm 22.3$
<b>IV</b>	$675.8 \pm 15.7$	$2429.6 \pm 37.2$	<b><math>3105.4 \pm 44.7</math></b>	$1693.8 \pm 27.7$	$1354.6 \pm 36.3$	$3048.5 \pm 44.3$
<b>V</b>	$1752.8 \pm 42.8$	$5614.8 \pm 53.5$	<b><math>7367.6 \pm 79.9</math></b>	$5463.1 \pm 51.8$	$1845.8 \pm 45.0$	$7308.9 \pm 78.1$

Similar negotiations seem to perform slightly better under this setup as compared to the previous one. This can be explained partly by the difference in the experimental

setup. Also, the two experiments were conducted at different times. The first experiment was conducted at a time when the alternative offer generation procedure and the fault tolerance mechanisms were not added to the policy manager. Also, the link characteristics appear to have been much better under the second setup, leading to vastly improved communication times; this explains why the total processing time for each case under Setup II is much closer to the total time than in the equivalent case under Setup I.

Still, as we can see, the actual times are of the same order as in the case of the first setup. The maximum time taken by the longest negotiation (Case V) is  $\sim 7.4$  seconds, which is almost the same as the  $\sim 8$  seconds reported in the other setup.

### **9.3. Event Filtering through the Policy Manager**

In this section we discuss the overhead incurred by passing events through policy engine filters as a means of access control (see Section 6.3.2). Performing access control impacts all communication involving sphere applications, since every event is inspected for policy compliance before being allowed to pass through to an application. The performance numbers for event propagation are presented in Eustice's dissertation [Eustice2008b]. Here, we focus on the extra overhead incurred by invoking policy engine mechanisms. We examine the overhead measurements drawn from two scenarios. All numbers in both tables below are reported with 99% confidence intervals.

**Scenario 1:** A sphere (S) and two applications (A and B) run on the same device, an IBM Thinkpad T42 (1.7 GHz, 512 MB RAM) laptop running Linux. Application A sends a set



of events sequentially to application B that has subscribed for events of that type. All events are propagated through the sphere S. The event flow path is  $A \rightarrow S \rightarrow B$ . Since the time taken to process each event on its way to the application is too small to measure without high variance, we measured the total time taken to process a set of events as well as the total overhead introduced by passing the events to the policy manager and getting back replies. We considered three cases:

1. Each event triggers a policy engine query which returns a negative response.
2. Each event triggers a policy engine query which returns an affirmative response.
3. Only the first event triggers a policy engine query, which is saved in the policy cache; this value is looked up for all subsequent events.

The values are indicated in Table 7 below. The left-hand column indicates only the policy manager overhead, whereas the right-hand column indicates the total time taken from when the first event was received to when the last event is dispatched to the application.

**Table 7.** *Event Filtering Performance on a Single Device (in milliseconds)*

	<b>Policy Manager Overhead</b>	<b>Total Time</b>	<b>Policy Manager Overhead Per Event</b>
<b>(1) 1000 Events</b>	$2473.43 \pm 21.30$	$3188.93 \pm 34.90$	$2.47 \pm 0.02$
<b>(2) 1000 Events</b>	$3244.66 \pm 31.05$	$4567.10 \pm 42.99$	$3.24 \pm 0.03$
<b>(3) 10000 Events</b>	$261.79 \pm 7.68$	$18584.13 \pm 62.78$	$0.03 \pm 0.00$

As expected, the overhead is least in case (3), where the policy cache is consulted rather than passing the event to the policy manager, a much more expensive task. In case (3), the time taken to pass each event is  $\sim 1.8$  milliseconds, with the overhead being negligible. In the first two cases, though, the policy filter contributes significantly to the total event processing time,  $\sim 77\%$  and  $\sim 71\%$  respectively. Therefore, even though the

policy filter overhead is only ~3 milliseconds per event, it becomes a performance bottleneck if the cache is not used. Also, we can see that the average time to process a valid query (~4.2 milliseconds) is higher than the time to process an invalid query (~2.5 milliseconds).

**Scenario 2:** A sphere (S) runs on an IBM Thinkpad T42 (1.7 GHz, 512 MB RAM) laptop and joins (becomes a member of) another sphere (T) running on a Sony Vaio MicroPC UX280P (Intel Core Solo U1400 processor, 1.2 GHz, 1 GB RAM); both machines ran Linux. Messages were exchanged through a TLS connection running over an 802.11b wireless channel. Application A running on the laptop associates with S and sends a number of events sequentially, the destination being application B running on the MicroPC and associating with T. The event flow path is  $A \rightarrow S \rightarrow T \rightarrow B$ . We considered the total time for a set of events, and the values indicated below have the same meaning as in the case of Scenario 1. We considered four cases:

1. The policy engine filter is completely omitted.
2. The policy engine filter is used, but the policy cache is not; the policy governing the event evaluates to *true*.
3. The policy engine filter is used for the first event, and the saved value looked up for all subsequent events.
4. The policy engine filter is used, and a cache miss triggers a negotiation requiring S to produce a valid social voucher; T verifies the voucher and lets the event pass to B.

**Table 8.** *Event Filtering Performance when Two Devices Interact (in milliseconds)*

	<b>Policy Manager Overhead</b>	<b>Total Time</b>	<b>Policy Manager Overhead Per Event</b>
<b>(1) 10000 Events</b>	404.23 $\pm$ 5.26	31355.85 $\pm$ 487.70	0.04 $\pm$ 0.00
<b>(2) 1000 Events</b>	9351.55 $\pm$ 42.52	11623.34 $\pm$ 116.00	9.35 $\pm$ 0.04
<b>(3) 10000 Events</b>	452.10 $\pm$ 86.75	30941.15 $\pm$ 481.45	0.04 $\pm$ 0.01
<b>(4) 1 Event</b>	2575.93 $\pm$ 64.90	2575.95 $\pm$ 64.90	2575.93 $\pm$ 64.90

The numbers in cases (1) and (3) are statistically identical, as expected. The time indicated in the left-hand column for case (1) is the time taken by the *SphereAppChannel* module to dispatch events to the application, and looking up the policy cache incurs almost no extra overhead. In each case, an event takes  $\sim 3.1$  milliseconds to propagate on average. Running a policy engine query increases the total time  $\sim 4$  times to  $\sim 11.6$  milliseconds, and the overhead is  $\sim 9.3$  milliseconds per event, or  $\sim 80\%$  of the total event processing time. The policy engine processing time is different from that observed in Scenario (1) because the queries were processed on the MicroPC rather than the Thinkpad, and because the policies were different on the former (though both evaluated the event pass query to *true*). In case (4), the first event requires  $\sim 2.6$  seconds to process, but subsequent events will be processed in  $\sim 11.6$  milliseconds (no cache) or  $\sim 3.1$  milliseconds (with cache). The average performance will therefore not change appreciably even if a negotiation is required.

#### **9.4. Renegotiation Overhead**

When sphere S successfully negotiates with sphere  $B_1$ , it incurs some overhead in deciding whether or not a renegotiation is required with other spheres that S is currently interacting with. We investigated how this overhead varied with the number of spheres

that  $S$  is currently related to, as a parent or a child. First,  $B_1$  negotiates successfully for membership in  $S$ . Then sphere  $B_2$  negotiates successfully for membership in  $S$ . The latter then tries to infer whether or not a renegotiation is required with  $B_1$ . This incurs some overhead. Such negotiations continue (without resulting in renegotiations), until sphere  $B_k$  negotiates successfully. At that point,  $S$  initiates renegotiations with spheres  $B_1$  through  $B_{k-1}$ . We varied  $k$  from 2 to 5, and present the overhead measurements in Figure 25. The policies that govern these negotiations are stated as follows:

```

appProhibitionLimit(k).

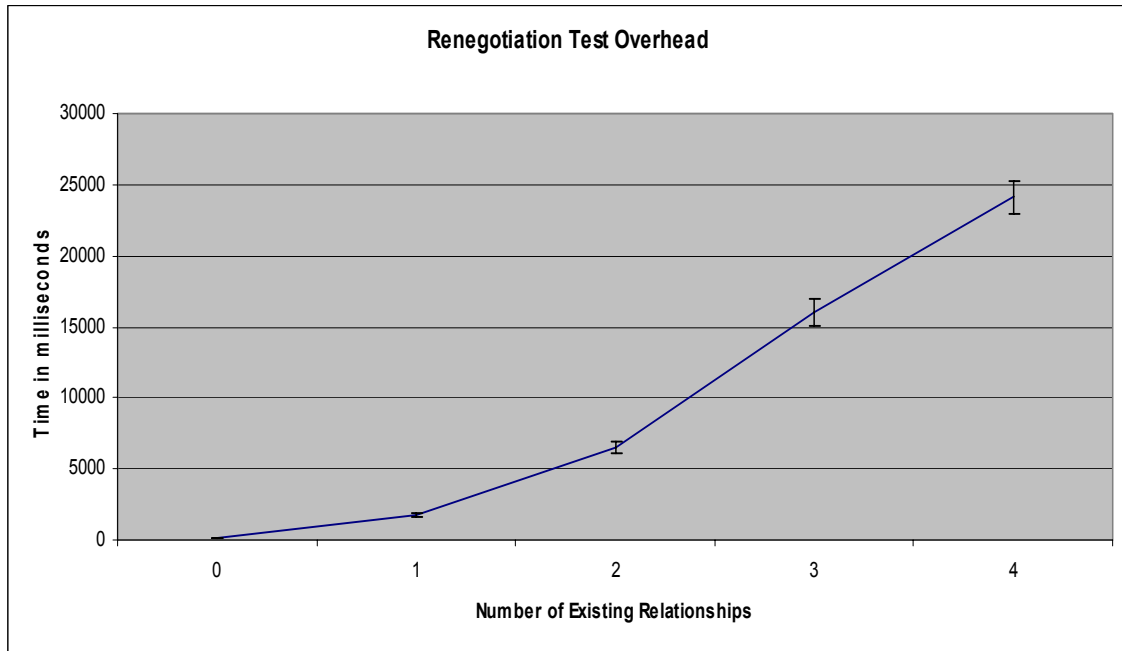
member(X) :- numChildren(N), appProhibitionLimit(L), N>=L,
             runApp(X, prohibit, App), networkApp(App, []), closedPort(X, 111).

member(X) :- numChildren(N), appProhibitionLimit(L), N<L,
             closedPort(X, 111).

member(X) :- childSphere(X), numChildren(N), appProhibitionLimit(L),
             N=L, closedPort(X, 111).

```

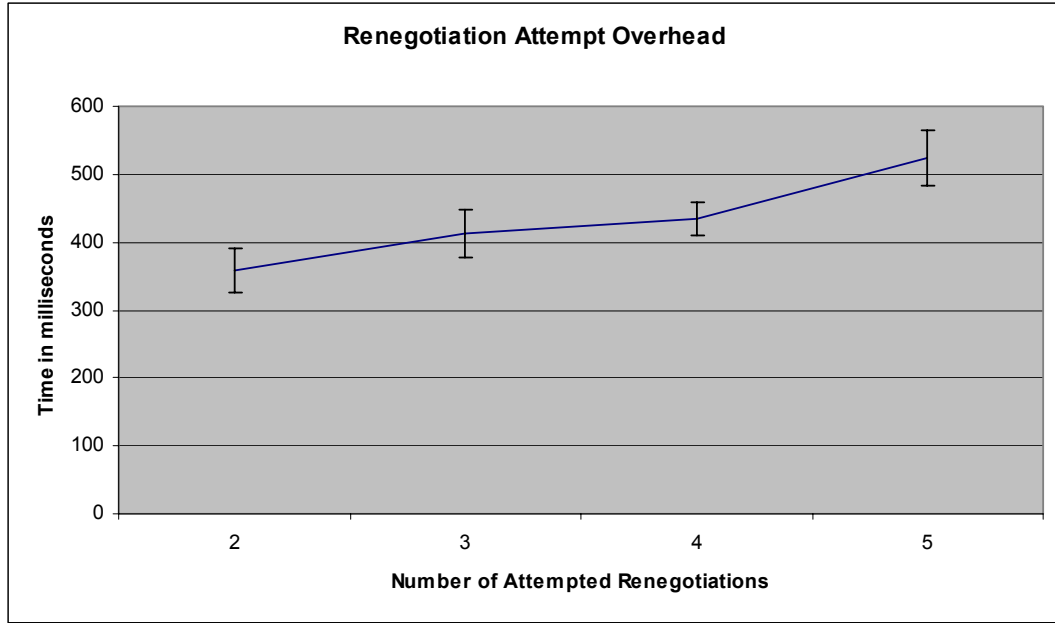
The graph in Figure 25 below indicates how the renegotiation overhead varies with the number of existing relationships. We found that the overhead for a given number of existing relationships remained almost constant with a change in the parameter  $k$ ; hence, the measurements in the graph are averaged over all values of  $k$ .



**Figure 25.** Variation of Renegotiation Overhead with the Number of Existing Associations

We can see a significant increase in the overhead as the number of existing associations increases; discounting the overhead at zero associations, this increase is roughly linear. This overhead is incurred by the process of running policy engine queries for every sphere that sphere A is currently associated with. The policies involve testing `closedPort(...)`, which includes the time taken to make operating system calls, which results in the overhead. This overhead increases (roughly linearly for  $r \geq 2$ ) with the increase in the number of spheres for which such function calls have to be made.

The overheads incurred when examination actually results in renegotiations are indicated by the graph in Figure 26.



**Figure 26.** Variation of Renegotiation Overhead with the Number of Attempted Renegotiations

The above graph indicates a gentle, and roughly linear, increase in overhead with the number of spheres that A must renegotiate with. In absolute numbers, the timings are significantly lower than in the table above because the queries that resolved to *false* took significantly less time than queries that resolved to *true*.

### 9.5. Comparing Negotiation to Centralized Policy Resolution

As we described in Chapter 8, it is possible to design an oracle that takes as inputs the policy sets and goals of the negotiators and outputs an optimal negotiation. Examining performance numbers and micro-benchmarks (see Sections 9.1 to 9.5) is useful, but a more realistic assessment of negotiation performance can be made by comparing its performance against the optimal case. Such comparisons vary widely with the characteristics of the input sets, and yield useful and illuminating results only when

conducted on a wide variety of inputs. Therefore, we are interested in obtaining statistical results from which we can draw useful conclusions and also predict policy resolution (both centralized and distributed) in the average case. We are primarily interested in how the number of negotiation steps (and predicted optimal number of steps) varies with different input sets, though we can observe other properties as well, such as the number of nodes in the policy resolution trees, the number of intermediate requests posed and granted, etc. Statistically, the nature of policy databases, as well as the number of optimal and negotiation steps in particular instances, affect the values of these metrics.

In this section we show how negotiation performance was measured by generating a large set of test cases and running both the oracle (centralized policy resolution) and the negotiation protocol (distributed policy resolution) for a large number of test goals/requests. First we list the evaluation metrics. Then we characterize test inputs and list the parameters that can be varied to produce meaningful results. Following that, we describe the procedure through which we generated a large number of parameterized test cases, and conclude with the measurement results.

### **9.5.1 Test Metrics**

- *Steps*: The number of steps taken for a negotiation can be compared to the number of optimal steps computed by an oracle for the same (or statistically similar) test case. The number of steps is related to the amount of network activity, and we would like the number of negotiation steps to be as close as possible to the number of oracular steps. We can also observe how the number of negotiation steps varies with the

characteristics of the policy databases (to be discussed later).

- *Size (number of nodes) of the Policy Resolution Tree:* The number of nodes in the tree that is generated, either by the oracle or collectively for the negotiators, is a measure of the amount of computation performed. An associated metric is the number of nodes per database unit ( $\text{\#nodes} / \text{\#policy rules in database}$ ), as the size of the database is likely to inherently affect the size of the generated tree.
- *Processing Time:* Measuring the actual system processing time is useful for the same reason as in the case of the nodes metric. Though this metric is expected to highly correlate to the number of nodes, differences are likely because the processing time for nodes may vary. The average time taken per step of negotiation (measured for each negotiation or oracular resolution instance) is a useful metric as well, because negotiation steps involve actions (such as counter-request generation and offer evaluation) that could vary widely in the amount of time taken. As in the case of the nodes metric, we also need to observe the time (total, and average per step) taken per database unit.
- *Number of Alternatives:* In a negotiation, the counter-request generation procedure produces a number of alternatives, some of which may succeed (i.e., sending them as requests may yield an affirmative offer). Were the negotiation protocol to select the correct alternative at every step, it would terminate in the number of steps predicted by the oracle. But in the absence of complete knowledge, this is not possible. We must therefore investigate how many alternatives are examined on average compared to the number of alternatives generated. This is a measure of protocol *efficiency*. The



fewer the number of alternatives examined, the better. Our negotiation procedure uses a simple heuristic of selecting the alternative (set of requests) of least cardinality.

- *Position of Valid Alternatives:* In the alternatives metric, there might be various cases when all alternatives are examined, none yielding an affirmative offer. This implies that a particular branch of the policy resolution tree failed. To observe how many alternatives are examined on average before a valid one is found, we count only those examinations that eventually lead to an affirmative offer.
- *Intermediate Requests Granted:* A number of intermediate requests and offers are generated by the negotiation protocol. The number of such requests satisfied (by either negotiator) gives a measure of the *privacy* maintained. Observing the number of intermediate requests generated, the number of requests granted, and the fraction of the requests generated that are granted, tells us how good the privacy-maintaining properties of negotiation are.

We did not consider the nature of the generated agreement as a test metric, because it is not very meaningful in this context. This is because the negotiation protocol is agnostic of the nature of the policy rules and the goals. Negotiation finds one among a number of equivalent goal assignments; i.e., if an oracle can find a way to satisfy a set of goals, a complete negotiation protocol must do so as well. A different but equally “good” agreement is acceptable within this definition. For example, in our ubiquitous conference room (see Section 7.2), any one among a set of printers may be offered, since the guest device simply requested a printer. To a user, a color printer may be qualitatively superior to a black-and-white printer. But the negotiation protocol cannot distinguish between

them unless the user explicitly expresses a preference in his goals. For every scenario, a theoretically optimal negotiation exists, whereby negotiators can reach one among a number of qualitatively equivalent agreements in the least number of steps. The number of steps in an actual negotiation depends on the heuristic used to select an alternative counter-request at any step. In an optimal negotiation, the first alternative selected always succeeds, whereas multiple “false leads” are followed in real negotiations, resulting in increase in the number of steps.

### **9.5.2 Characterization of Test Inputs**

The key challenge in producing statistically meaningful performance results for negotiations is the generation of test cases whose characteristics can be controlled through well-defined parameters. Our test cases consist of policy databases and initial goals, though once we generate the databases, we could generate a set of goals simply by examining the list of policies and extracting the heads of clauses. In randomly generated databases, a large fraction of such goals are likely to result in failure, but a significant fraction (the remainder) will result in success. Given a suitably large number of test cases generated randomly (through a procedure we will describe later), we can generate a large number of negotiation scenarios that produce successful negotiations of varying length. For simplicity, each negotiation instance consists of exactly one initial goal/request. The metrics described above can be measured using single-request negotiations. Though some multiple request negotiations will shed some light on the solution quality (certain intermediate requests may conflict, as we discussed in Chapter 8), the vast majority of

negotiations will only tell us how processing overhead increases with more requests. Also, the time taken to run our tests was quite large for single request negotiations, and the running time would have been prohibitively high for multiple request negotiations.

Policy databases consist of statements, i.e., facts and rules. One measure of database size is simply the *total number of facts and rules* in the database. Another measure is the total size of all policy rules. But these parameters, though relevant, may have little or no correlation with the actual performance of policy resolution (either centralized or decentralized) as reflected by our metrics. This is because not all facts and rules are relevant to a particular negotiation goal. Indeed, it is hard to characterize a policy database (or pair of databases given as inputs) quantitatively in an exact manner. Therefore, we turn to the next best option—the expected size of a derivation tree (the number of tree nodes) that results from providing a random input (requests) to a policy resolver that knows what facts and rules are present in one or both databases. Whether the resolution is centralized or distributed, each node in the tree (which represents a database element, either a fact or a rule) is examined once, so the number of nodes has a direct impact on performance. We implicitly assume uniform processing time per node (i.e., time varies linearly with number of nodes). This does not hold true in real-life scenarios, as helper functions associated with requests and offers increase processing time arbitrarily, but the assumption is reasonable for testing purposes. We did not use or simulate helper functions, which is a simplifying assumption but one that preserves the theoretical correctness and completeness properties of the negotiation protocol. To generate non-trivial database pairs, we set bounds on the maximum size of derived trees.

A tree can be bounded by two parameters: *breadth*, or branching factor, and *depth*. Random database pairs that generate policy resolution trees whose sizes are bounded by a given  $\langle \textit{breadth}, \textit{depth} \rangle$  tuple can be generated. Negotiation and oracular policy resolution characteristics (the metrics listed above) will vary directly with variations in breadth and depth of the input databases.

Our performance measurement strategy involved generating a large number of database pairs for a range of breadth and depth values. Both oracle and negotiation tests were conducted for a set of requests for each input, and readings recorded. (*Note: Using tree sizes (particularly depth) to characterize the performance of systems is not unique [Minami2006] to our framework, though our experience with generation of random test cases provides valuable lessons to other researchers. We are also hopeful that our experiences will make the case for establishing tree branching factors and depths as performance benchmarks for logic-based expert and semi-expert systems.*)

### 9.5.3 Test Parameters

To generate test cases (pairs of databases), we use the following parameters:

- *Maximum Branching Factor (b)*: Any database pair constructed using this parameter can never result in a policy resolution tree that has a higher branching factor (number of immediate descendants, or children, of a node) than  $b$ . This parameter indicates a bound rather than an exact or approximate characteristic. Also, the trees are not guaranteed to be balanced; nodes at an intermediate level could have any number of children, varying in the range  $[1 \dots b]$ . In practice, a large number of trees with

branching factor much smaller than  $b$  will be generated as well, depending on the goal/request provided as input.

- *Maximum Depth ( $d$ ):* Any database pair constructed using this parameter can never result in a policy resolution tree that has a higher depth (distance from the root node to a leaf node) than  $d$ . This parameter indicates a bound rather than an exact or approximate characteristic. Also, the trees are not guaranteed to be balanced; leaf nodes could lie at any distance from the root, varying from 1 to  $d$ . In practice, a large number of trees with maximum depth less than  $d$  will be generated as well, depending on the goal/request provided as input.

*Why Bounds?*—A database consists of facts and rules, whose relations are specified through the Horn clause syntax. It is possible to generate a set of random facts and rules that will result in exactly one tree with uniform  $b$  and  $d$ . But this tree would manifest itself in practice only if the posed request corresponds to its root. In other words, guaranteed values of  $b$  and  $d$  are characteristic of a tree and not of a database. A database, on the other hand, can only be characterized through bounds or averages. A random request will result in the generation of a proof tree with branching factor at most  $b$  and depth at most  $d$ . Ensuring that a self-contained set of predicates will always produce a tree of guaranteed branching factor and depth is not possible. Generation of rules will result in cycles, and our query procedure would fail. Hence, we use bounds rather than guarantees on  $b$  and  $d$ . (*Note:* We conjecture that average values of  $b$  and  $d$  may be useful characteristic parameters as well. For our study, we just happened to pick maximum bounds, and we intend to use averages as our parameters in future work.)

The test results for all the database/goal scenarios can also be collected and classified on the basis of the following parameters:

- *Predicted Number of Steps for an Optimal Negotiation*: The least number of negotiation steps computed by the oracle provides a good classification basis. For example, we generate statistical results on all result sets (based on the metrics listed earlier) for which the oracle-predicted number of steps is 3; likewise for 5, 7, etc. (*Note*: The number of steps is always an odd number because every request must have a corresponding offer and every negotiation ends with a termination message.) Statistical results could be compiled for the metrics listed above. Instead of using predicted bounds on the generated trees, we use actual bounds on the generated trees (though the database characteristics could vary across the range of maximum branching factors and depths).
- *Number of Negotiation Steps*: The actual number of negotiation steps also provides a good classification basis. Statistical results are compiled for the metrics listed above.

#### **9.5.4 Generation of Test Cases**

To run a test scenario for policy negotiation as well as centralized policy resolution using an oracle, the following must be provided as inputs:

- Names of the two negotiators, expressed in the form of Panoply sphere IDs
- An initial request posed by one negotiator to the other (multiple such requests can be posed for the same set of policy databases, as we will see later)
- A pair of policy databases

- A policy database (set of policies) consists of a set of facts ( $f$ ) and rules ( $r$ ).
- Each fact will be of the form:
  - $\text{pred}(\text{arg}_1, \text{arg}_2, \text{arg}_3, \dots, \text{arg}_n).$
- Each rule will be of the form:
  - $\text{pred}(\text{arg}_1, \text{arg}_2, \text{arg}_3, \dots, \text{arg}_n) \text{ :- } \text{pred}_1(\text{arg}_{11}, \text{arg}_{12}, \text{arg}_{13}, \dots, \text{arg}_{1n}), \dots$

The following are inputs to the test generation procedure:

- The **size** of a database pair:
  - $b_{max}$ : Maximum branching factor of derived policy resolution trees  
 Random rules (Prolog clauses) are generated so that the body of the clause does not contain more than  $b_{max}$  predicates.
  - $d_{max}$ : Maximum depth of derived policy resolution trees  
 Random rules are generated so that a predicate will be added to the body of a clause only if the depth of any proof tree generated using the database pair does not exceed  $d_{max}$ .
  - **initialNumRules**: Initial number of rules/clauses in both databases combined  
 This does not indicate the final database size. During database generation, more rules will be generated to increase the number of available negotiation alternatives.
- Policy statement ontology:
  - As described in Chapter 4, we have a pool of designated (keyword) predicates and argument names drawn from our policy language ontology that represent object

types, action types, states, requests, characteristics of objects, and instances of objects. The predicates have fixed arities (number of arguments). These predicates and arities are given below:

- *requestNames* = {"accessInfo/2", "obey/2", "access/2-3", "member/1"}
- *stateNames* = {"possess/1-2", "action/2", "memberIn/1"}
- *queryNames* = {"location/1", "tim/1", "displayName/1", "printerName/1", "groupName/1", "groupSize/1", "bandwidth/1", "storage/1", "parentName/1", "childName/1"}
- *objectNames* = {"voucher/1", "printer/1", "file/1", "disp/1", "door/1"}
- *characteristicNames* = {"type/2", "directory/2", "group/2", "brand/2"}
- All predicate names are drawn from this pool of keywords. This pool is representative of real-world scenarios and is sufficient for generating meaningful test cases for the range of breadth and depth values we are interested in.
- Most arguments of the facts are randomly generated Prolog constants (based on the Prolog syntax). These constants could be strings as well as integers. Some arguments are drawn from a fixed pool, such as the following:
  - *actionNames* = {"run", "closePort", "prohibit", "play", "open", "close"}
  - *possessionNames* = {"diskSpace"}
  - *constantNames* = {"type-color", "type-bw", "brand-hp4150", "brand-hp7100", "group-ucla", "group-acm", "group-ieee"}
- These random constants introduce variations in the nature of policy databases, in addition to the quantitative parameters.



- All arguments of predicates in either the head or body of a clause are Prolog variables (random strings that begin with a capital letter).

*Algorithm:*

1. Drawing from the object name pool, generate at most  $b_{max}$  “ground predicates” that will be asserted as facts. All arguments will be stub variables.

For example: `printer(X), door(X).`

2. Count the total number of arguments in this set of ground predicates; let it be equal to  $numArgs$ . Generate  $numArgs$  random Prolog constants. Substitute the stub variables in the ground predicates with these constants.

For example: `printer(o), door(zp).`

3. Drawing from the query name pool, generate at most  $b_{max}$  query predicates and assign arguments as above.

For example: `groupSize(d), displayName(q).`

4. Drawing from the characteristic name pool, generate at most  $b_{max}$  predicates corresponding to characteristics of objects and actions, and assign arguments.

For example: `type(uf,color), brand(e1, hp4150).`

5. Associate every characteristic predicate with a ground predicate.

For example: `{printer(o), brand(o, hp4150)}, {voucher(ec), group(ec, acm)}.`

6. Drawing from the state name pool, generate at most  $b_{max}$  state predicates. Associate ground predicates with each state predicate. Substitute the arguments in the ground predicates within the corresponding state predicates.

For example: `{possess(o), printer(o), brand(o, hp4150)}`.

7. Split the set of state predicates generated above into two equal halves, one set to each policy database. Assign each state predicate to one set or another in a random manner. For every ground predicate associated with the state predicates, assign each to the corresponding policy database. Also assign it to the other policy database with a probability 0.5.

8. Drawing from the request name pool, generate at most *initialNumRules* predicates corresponding to heads of clauses.

For example: `obey(X, closePort)`.

9. For each clause, add a suitable state predicate (and associated ground predicates) or a query predicate to the body.

For example: `access(X, VAR) :- voucher(VAR), type(VAR, VAR1)`.

10. Assign each clause to the same database that its corresponding state predicate is assigned to.
11. Associate a depth value to each clause and initialize it to -1. Then, for each clause, associate a depth value of 0 with probability  $1/d_{max}$ . The depth value of each clause indicates the maximum height of a proof tree rooted at the head of that clause (see Figure 21). Here, we force a fraction of the clauses to be leaves. The remaining clauses are initialized to have ambiguous depth so that they can be augmented.
12. Enhance each clause by adding more predicates to its body. Again, based on how a proof tree is derived, children of a tree node represent other clauses. Therefore, when we add predicates to the body of a clause during test generation, we are effectively

adding children in a potential policy resolution tree. We must also prevent cycles in proof resolution. Therefore, the set of candidate “child” clauses from which body predicates can be drawn is restricted.

13. Initialize the set of candidate children for each clause with depth -1. This set contains all clauses whose heads are not similar to the head of the clause under consideration. For example: if both clauses  $C_1$  and  $C_2$  are of the form `access (VAR1, VAR2) :- ...,` neither of them can be used to augment the body of the other. On the other hand, if  $C_1$  is of the form `access (VAR1, VAR2) :- .....` and  $C_2$  is of the form `obey (VAR1, run) :- .....`,  $C_1$  would be a candidate child of  $C_2$  and vice versa.
14. For each value in the range 1 to  $b_{max}$ :
  - a. For each clause  $C$ , attempt to augment the body with a certain probability (we selected a probability of 0.5).
  - b. To augment the body, draw a clause  $C'$  from the candidate children set of  $C$ . If the candidate children set is empty, do not augment the body of  $C$ . Format the state predicate associated with the  $C'$  and add it to the body of  $C$ . Formatting of the clause involves adding extra arguments (variables) to predicates that are drawn from clauses that have been assigned to a different database.
  - c. Update the depth of the  $C$  in the following way:  $\text{depth}(C) = \max(\text{depth}(C), \text{depth}(C') + 1)$ .
  - d. Update the list of candidate children of every clause. Children include those clauses that are not potential parents in a proof tree (we keep track of all descendants of each clause for this purpose), are not similar Prolog predicates and

whose depth value does not exceed  $d_{max}$ . The last condition enforces the depth bound criterion on the generated test database pair.

- e. For every augmented clause  $C$ , if the selected child clause ( $C'$ ) belongs to the same database, add an extra clause by duplicating  $C$  and modifying the predicate associated with  $C'$  by formatting it as if it belonged to the other database. This creates two clauses from one, and results in more available alternatives in negotiation scenarios. We added this feature after discovering that our preliminary attempts at generating database pairs resulted in few or no alternatives during actual negotiations.

*Realism of Generated Policy Databases:* The policy databases generated by the above algorithm reflect the types of databases we have used in our example applications (see Chapter 7) and also other databases that ubicomp domains will use in practice, were they to run a policy manager built along our design. Our test case vocabulary (described earlier) is almost identical to the actual vocabulary (see Appendix A). We did use a few extra keywords for designated (global) predicate and parameter names (e.g., {groupSize/1, type/2, brand/2}). But this does not detract from the realism of the generated databases, as the policy engine algorithms and the negotiation protocol itself is agnostic of these names. We described in Chapter 4 how different domains could choose different local vocabularies and agree on a shared global vocabulary. This agreement is domain- and application-dependent, and virtually any names consistent with the SWI-Prolog syntax will suffice. Apart from the names, our databases contain facts and Horn

clauses, using the same operators that we would use in real policy databases. Hence, we can confidently assert that our test case generator produces realistic test scenarios.

#### *Generation of Initial Requests:*

The initial requests are drawn from the state predicates and their associated ground predicates generated in step 6 above. Therefore, the number of initial requests generated for each database pair is equal to the number of state predicates generated in step 6. Requests are generated for both negotiators, so that we obtain multiple test scenarios with the same policy database pair. This enables the generation of a much larger number of test cases than the number of database pairs. Since database generation is a time-consuming process, especially as the breadth and depth bounds increase, generating multiple requests for each pair is practically useful. Below, we list the initial requests generated for the pair of databases whose contents are listed above.

Appendix B lists an example policy database and request set pair generated by the test case generation algorithm described above.

### **9.5.5 Test Results**

*Platform:* The experiments were conducted on an Intel P4 (2.53 GHz, 512MB RAM) desktop. Though negotiations in real-world scenarios would be conducted between two computers connected by a network, they can be conducted between two distinct spheres running on the same computer as well, the only difference being the low communication

latency in the latter case. Because we were interested only in the performance of the negotiators and their processing times, running both negotiating spheres on the same computer made sense, especially as it eliminated network latency and yielded quicker results. Each negotiator process waits for a response while the other is running. Since the processing times don't overlap, we were able to record accurate readings.

*Test Cases:* We generated test cases for the following parameter values:

- $b_{max}$  varying from **1 to 10**
- $d_{max}$  varying from **1 to 20**
- ***numRules* = 28**

Note that this is the initial number of rules input to the test case generators. The final number of rules, after generation of alternative clauses, is variable, though always greater than *numRules*. We selected this value based on the size of the request predicate pool (which is equal to 4) and used trial and error, whereby 7 clauses of each request type will be generated initially. We chose 7 simply because we found lower values provided less diverse and meaningful test cases, whereas higher values resulted in higher running times than we were willing to tolerate.

The alternative selection heuristic used by each negotiator running our protocol was the *least cardinality of the request set*.

For each value of  $\langle b_{max}, d_{max} \rangle$ , **40** test cases (database pairs) were generated. Therefore, a total of  $200 \times 40 = \mathbf{8000}$  database pairs were generated. The total number of

policy statements in all these databases is equal to **988263**, or an average of **~62** statements per database.

A variable number of initial requests were generated for each test case. The total number of scenarios ( $\langle \text{database-pair}, \text{initial-request} \rangle$ ) generated for any breadth and depth value was equal to **194953**, or an average of **~24** requests per database pair.

### 9.5.5.1 Metric Results Based on Parameters

We ran tests for every database pair and our recorded results are associated with the corresponding pair. To make sense of the results, we aggregated them based on certain parameters and ran statistical measurements. The parameters and the measured metrics are listed in Table 9 below. Following that, we present and analyze the results (in the same row and column order as shown in the table):

**Table 9.** *Roadmap to Results of Optimality Measurements*

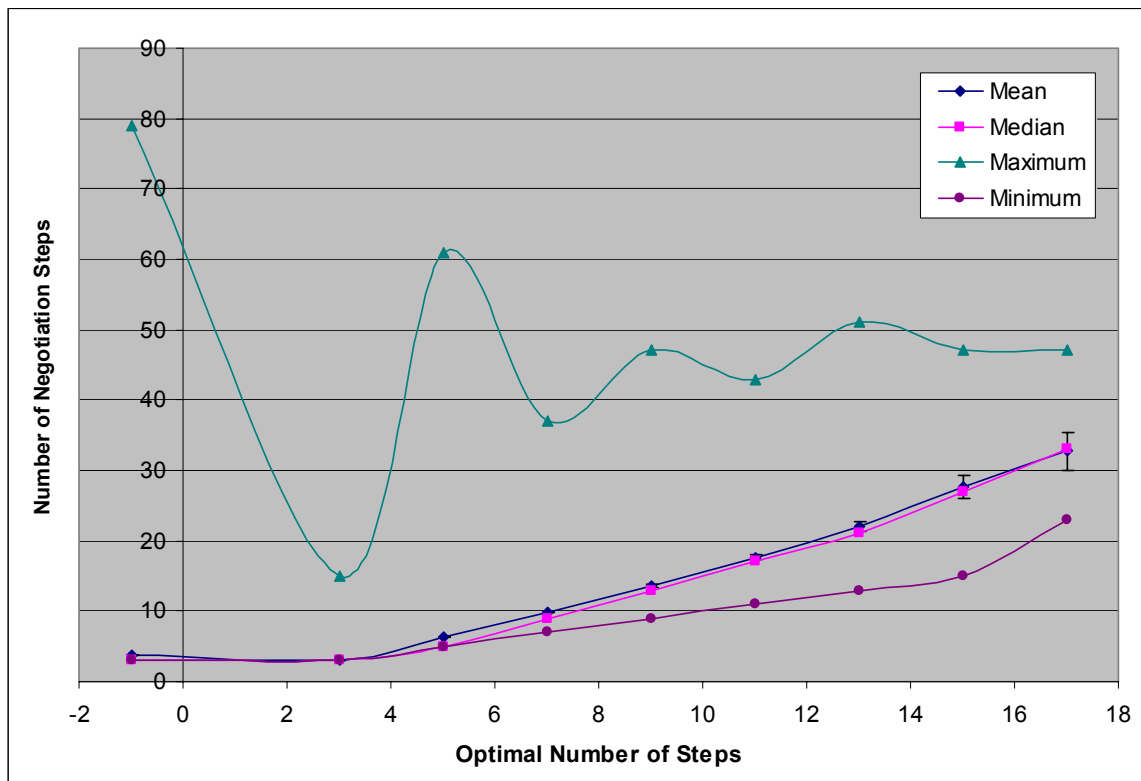
	Aggregation Parameters	Measured Metrics				
		Number of Negotiation Steps	Size of Policy Resolution Tree	Processing Time	Intermediate Requests	Examined Alternatives
(a)	Length of Optimal Negotiation ( $l_{min}$ )	✓	✓	✓	✓	✓
(b)	Length of Actual Negotiation ( $l_{neg}$ )	✗	✓	✓	✓	✓
(c)	Database Branching Factor Bound ( $b_{max}$ )	✓	✓	✓	✓	✓
(d)	Database Depth Bound ( $d_{max}$ )	✓	✓	✓	✓	✓

#### 9.5.5.1.1 Length of Optimal Negotiation ( $l_{min}$ )

We first aggregated all the results obtained for all  $\langle b_{max}, d_{max} \rangle$  pairs and classified them based on the optimal negotiation length as estimated by the oracle. From our

measurements, we were able to obtain test cases with optimal negotiations in the following set:  $\{-1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$ . The -1 value indicates a failed negotiation, i.e., no valid derivation tree exists for the particular goal. All other numbers are odd because negotiations typically consist of request-offer pairs and always end with a termination message. Results based on individual metrics are indicated below.

*Number of Steps:* Figure 27 and Table 10 below show how the number of actual negotiation steps compares with  $l_{min}$  (the mean is reported with 99% confidence intervals).



**Figure 27.** Comparison of Actual and Optimal Negotiation Steps



As the values indicate, the mean and median curves almost coincide; therefore, we can characterize the number of negotiation steps tightly. The number of negotiation steps seems to increase linearly with increase in  $l_{min}$  on the average. This conclusion is validated by the fact that running a linear regression on the mean negotiation steps data set gives us an  $R^2$  value of 0.99, and the slope is 2.11. The minimum number of negotiation steps seems to increase linearly as well until  $l_{min}=15$ . In fact, the minimum number of negotiation steps is identical to  $l_{min}$  for each value; this indicates that negotiations are sometimes optimal in real-world situations. The reason for convergence of the curves is simply because the number of data points (frequency) is rather small for higher values of  $l_{min}$ ; in fact, we have only one data point for  $l_{min} = 19$ . The maximum number of negotiation steps seems to oscillate somewhat between alternate data points, yet seems to follow a roughly increasing trajectory, being approximately 5 to 12 times the mean (or median) number of negotiation steps.

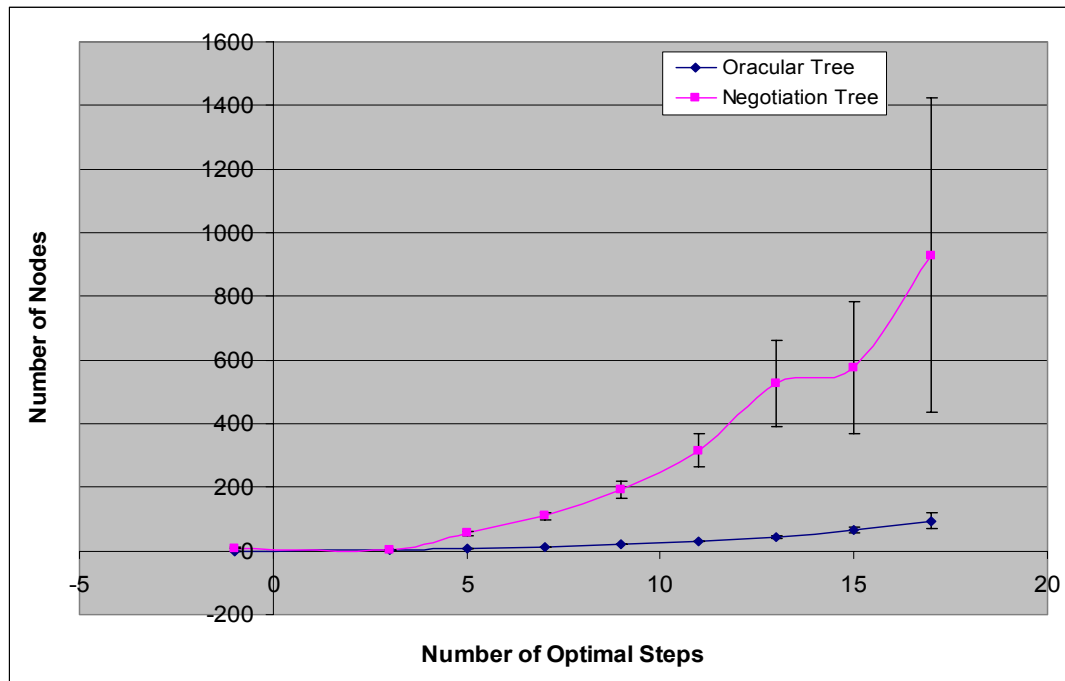
**Table 10.** *Comparison of Actual and Optimal Negotiation Steps*

OPTIMAL NEGOTIATION STEPS	ACTUAL NEGOTIATION STEPS			
	<i>Mean</i>	<i>Max</i>	<i>Min</i>	<i>Median</i>
-1	3.74	79	3	3
3	3.00	15	3	3
5	6.33	61	5	5
7	9.84	37	7	9
9	13.54	47	9	13
11	17.69	43	11	17
13	21.97	51	13	21
15	27.64	47	15	27
17	32.71	47	23	33

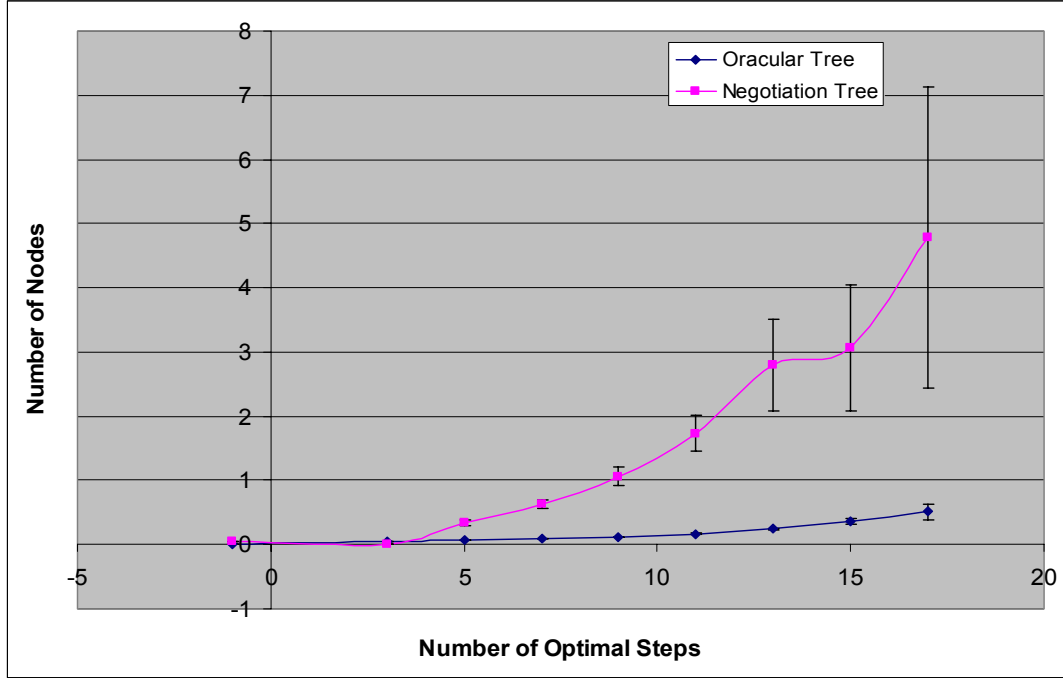
The special case of  $l_{min} = -1$  is also interesting to observe. As would be expected, more than half the negotiations (~55%) failed, so we obtained a very large number of

data points. But both the mean and median values were very close to 3 (actually 3.74), which is the shortest negotiation one could hope for even when the result is failure. On the other hand, the wide variation can be observed from the maximum number of negotiation steps, which was found to be 79; this was a case where the policy databases yielded a large number of alternatives that had to be explored.

*Number of Policy Resolution Tree Nodes:* Figures 28 and 29 indicate how the average number of nodes examined varies with the optimal negotiation length. The numbers are reported with 99% confidence intervals.



**Figure 28.**  $l_{min}$ : Number of Nodes Examined During Policy Resolution



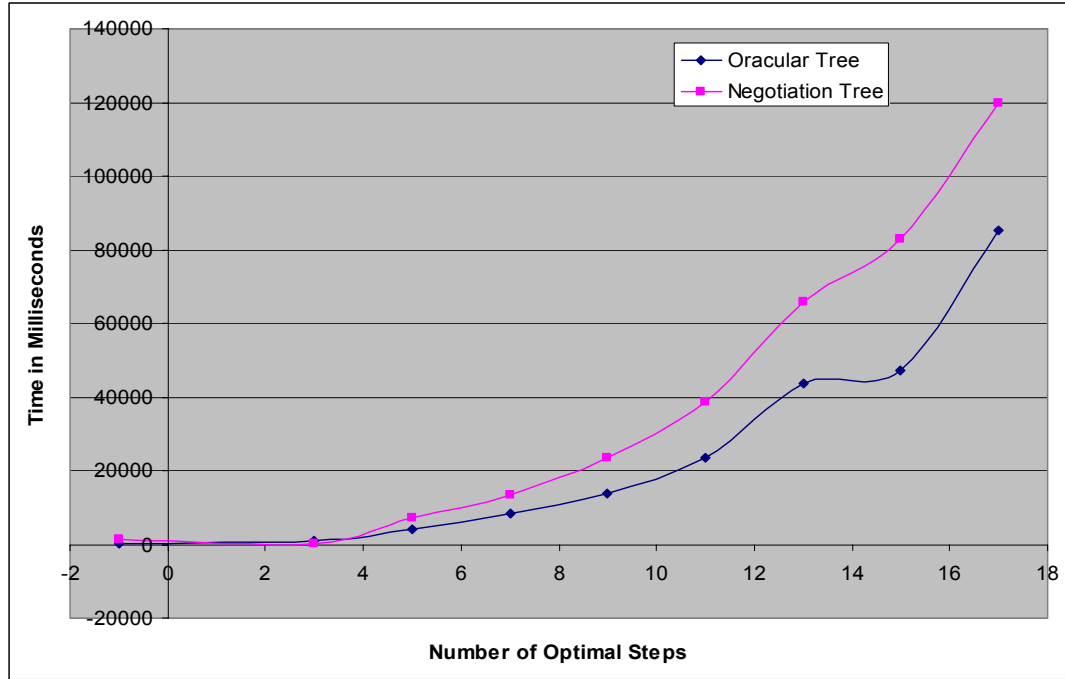
**Figure 29.**  $l_{min}$ : Number of Nodes Examined per Unit Database (Number of Nodes / Number of Policy Statements in the Database)

The confidence intervals for negotiations are particularly large for higher values of  $l_{min}$ . This is simply because there are fewer instances (data points) for such negotiations, leading to high variance. Given more time and more instances, we would likely obtain tighter bounds. Indeed, based on the shapes of the curves up to  $l_{min} = 13$ , we can extrapolate a roughly exponential curve with a high degree of confidence. An exponential curve is expected, given that trees grow exponentially with increase in depth (which correlates with number of steps). Also, the oracular tree curves indicate a much gentler exponential rise compared to negotiation trees. Given that the number of steps for a typical negotiation is much larger than the corresponding value of  $l_{min}$ , this is an expected result as well. But the ratio between the number of nodes rises more gently, in a roughly linear way; the ratio of negotiation tree nodes to oracular tree nodes is  $\sim 7$  for  $l_{min}$

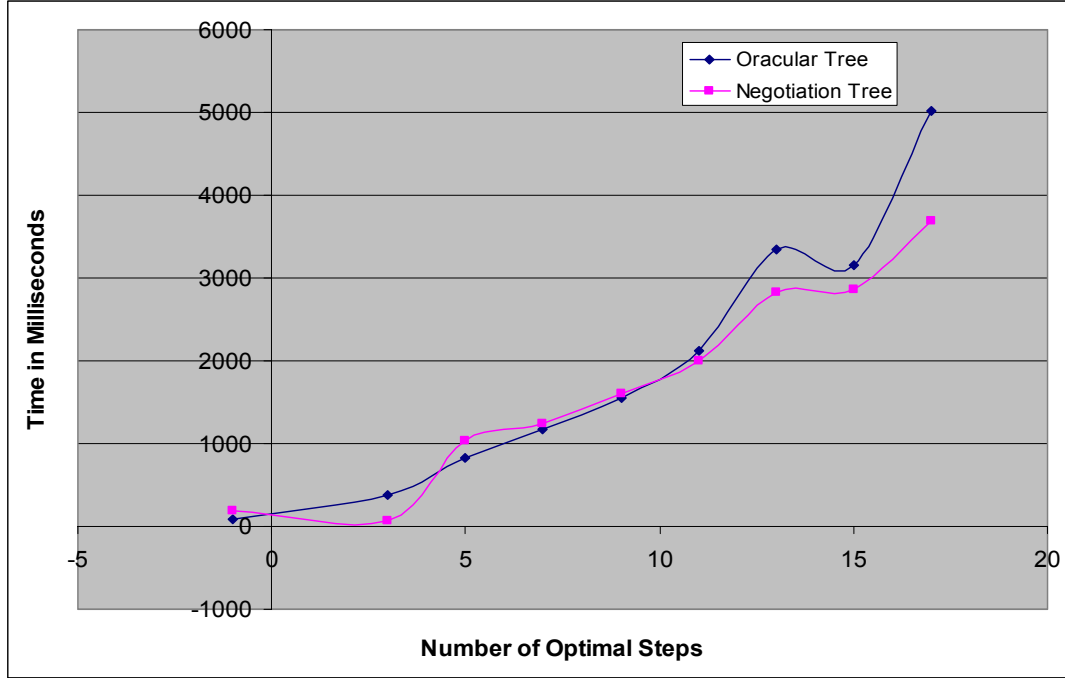
= 5 and is  $\sim 12$  for  $l_{min} = 13$ . We cannot control the length of the optimal negotiation, but this is an encouraging result for the prospect of optimizing negotiations in the future. At the very least, it indicates that negotiation performance does not suffer too badly in comparison to oracular policy resolution.

The curves in both Figures 28 and 29 have virtually identical shapes. Though that does not tell us anything more about how tree size varies with  $l_{min}$ , it does indicate that an increase in the database size implies a similar proportional increase in the number of tree nodes for both kinds of policy resolution.

*Processing Time:* Figures 30 to 33 indicate how the processing time for policy resolution varies with the optimal negotiation length.



**Figure 30.**  $l_{min}$ : Total Processing Time for Policy Resolution

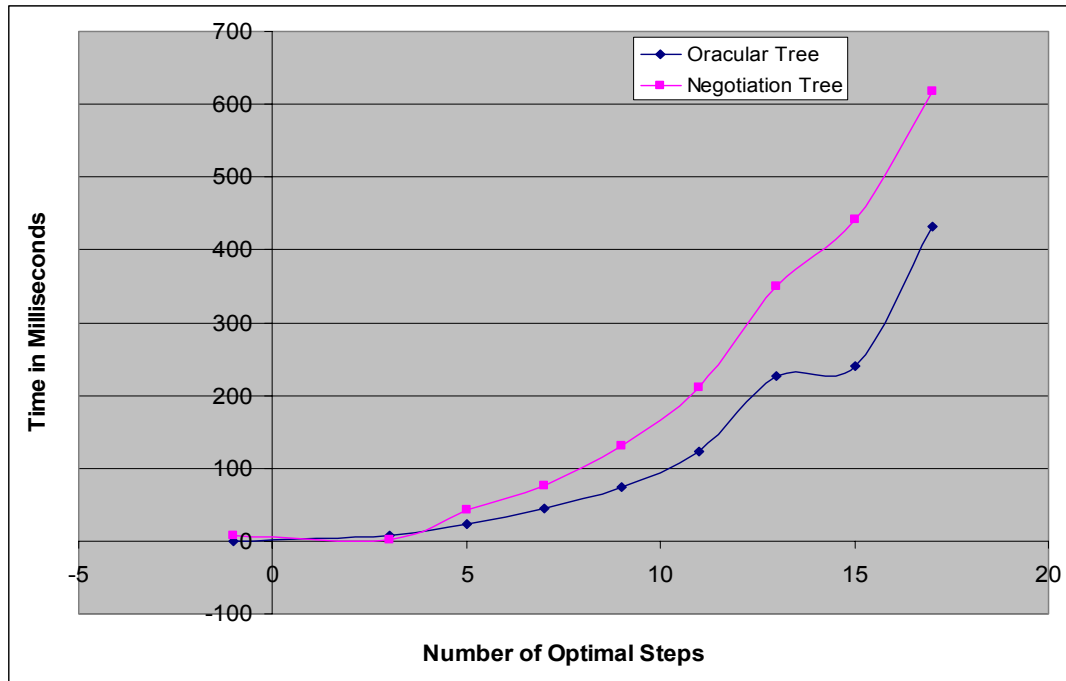


**Figure 31.**  $l_{min}$ : Average Processing Time for Policy Resolution per Negotiation Step

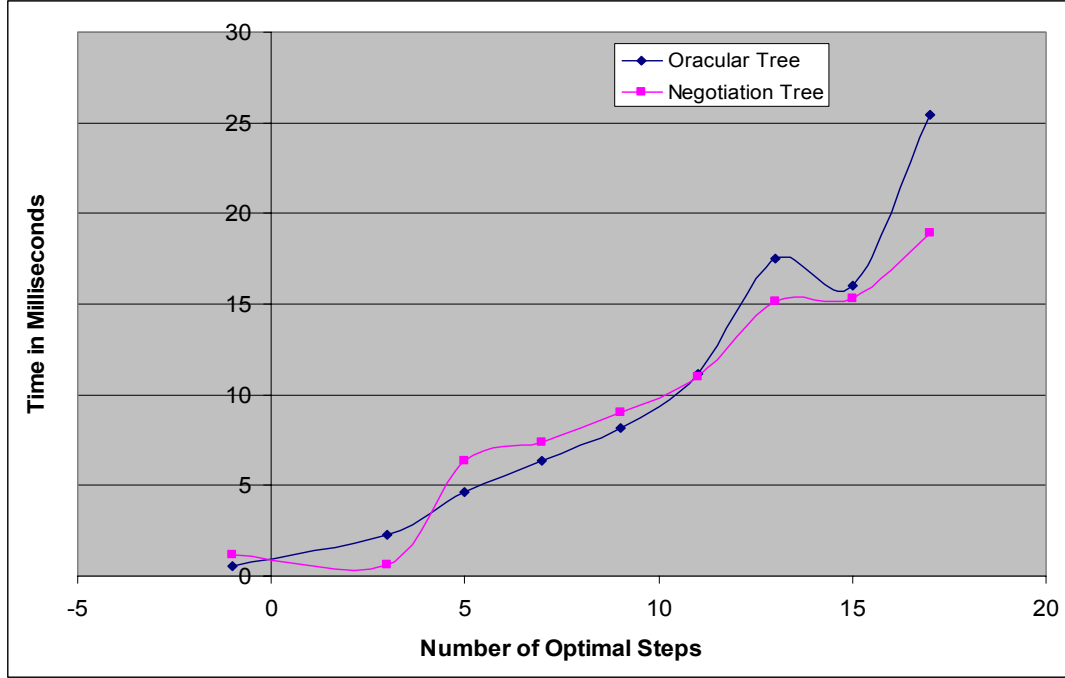
Figures 30 and 31 show similar trends to that observed in the case of the number of nodes in a policy resolution tree. We see a similar exponential rise up till  $l_{min} = 13$ , and variable behavior beyond, owing to lack of data points. On average, the slowest policy resolution occurs at  $l_{min} = 17$ , where the oracle takes  $\sim 85$  seconds to complete, whereas the negotiation protocol terminates in  $\sim 120$  seconds. A more typical negotiation would occur at  $l_{min} = 11$ , where the negotiation time is 40 seconds. Though tolerable in many mobile and pervasive computing situations, this is nevertheless quite slow and offers room for optimization. Faster processors would no doubt help, but one fears that limitations in AI techniques (expert and semi-expert systems, which emulate search, have inherently exponential growth) are a larger bottleneck.

The graph in Figure 31 is more encouraging, and indicates that the time taken per negotiation step is lower (and increases at a lower rate) than the corresponding time taken

per unit step by an oracle. The reason may be straightforward. Processing during a negotiation session involves running queries to check whether requests comply with policies, and generation of counter-requests and alternative offers. An increase in the number of steps would likely result, on average, in a proportional increase in processing time. On the other hand, an oracle would have to exhaustively examine a large number of tree paths (including paths that eventually lead to failure). The processing time may increase, even for a short negotiation, thereby increasing the average processing time per step. *Therefore, if we were able to optimize negotiations by finding heuristics that could decrease the number of negotiation steps on average, we could end up with significant savings in system processing time.*



**Figure 32.  $I_{min}$ :** Total Processing Time for Policy Resolution per Unit Database (Total Processing Time / Number of Policy Statements in the Database)

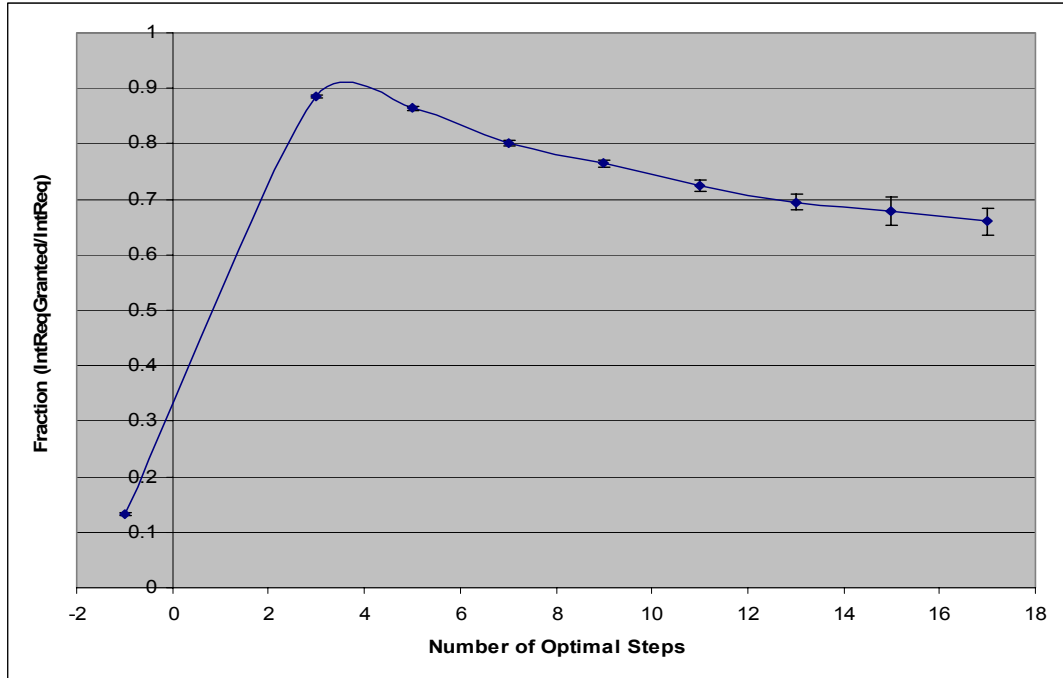


**Figure 33.  $l_{min}$ :** Average Processing Time for Policy Resolution per Negotiation Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database)

The curves in Figures 32 and 33 virtually replicate the results indicated in the earlier curves (Figures 30 and 31). As in the case of the tree size, this simply confirms that processing time increases proportionally with respect to both the number of tree nodes and the size of the policy database.

The examined nodes and processing time graphs (Figures 28 to 33) indicate that failed negotiations ( $l_{min} = -1$ ) result in small (or minimal) trees on average. The processing time for such negotiations is  $\sim 1$  second on average. This is a good result, as it indicates that our negotiation protocol does not waste processing time when the result is a likely failure, irrespective of the database size or the breadth and depth bounds.

*Number of Intermediate Requests:* We measured the number of intermediate requests posed by a negotiator to another (*intReq*) as well as the number of such requests that resulted in an affirmative offer (*intReqGranted*). For each test scenario, the fraction of requests granted (*intReqGranted/intReq*) was measured. We only considered those test cases for which a non-zero value of *intReq* was recorded. Averages of these fractions were computed for each unique value of  $l_{min}$ . Figure 34 shows these fractions with 99% confidence intervals.



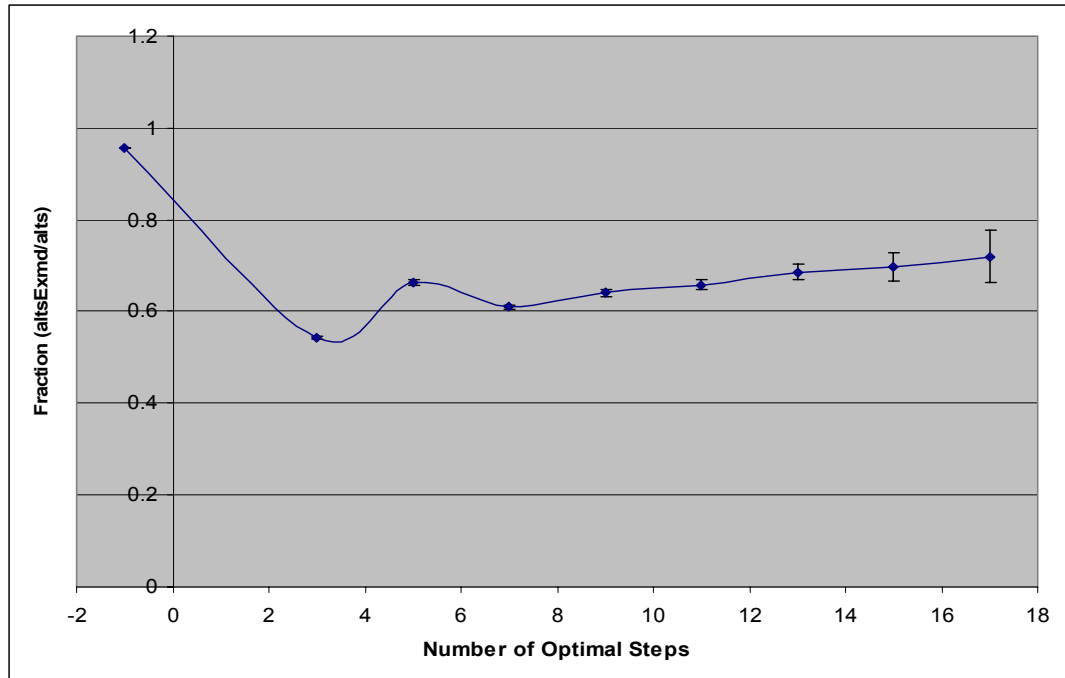
**Figure 34.**  $l_{min}$ : Average Fraction of Intermediate Requests Granted

We can see that the fraction of intermediate requests granted decreases monotonically (disregarding the point on the curve corresponding to failed negotiations:  $l_{min} = -1$ ) with increase in  $l_{min}$ . The maximum value (0.89) occurs at  $l_{min} = 3$ . Extrapolating to higher values of  $l_{min}$ , we may expect the curve to flatten out, eventually reaching a



terminal value of about 0.6. The nature of the curve makes this assertion plausible. Lower values of  $l_{min}$  seem to be outliers. The reason why the fraction is high for lower values of  $l_{min}$  may simply be because the number of intermediate requests itself is fairly low. Negotiations where  $l_{min} = 3$  would mostly be of length 3 or 5, as is evident from the average number of negotiation steps. Successful negotiations of length 5 comprise one intermediate request which results in an affirmative offer, resulting in a fraction value equal to 1. The larger number of 1s at lower values of  $l_{min}$  increases the average. This does not happen at higher values of  $l_{min}$ . The monotonic decrease of this fraction indicates that the negation protocol is effective at maintaining privacy to some extent.

As the vast majority of intermediate requests in failed negotiations result in declined offers, and a large number of fraction values are expected to be 0, it is no surprise that the average fraction at  $l_{min} = -1$  is very close to 0 ( $\sim 0.13$ ).



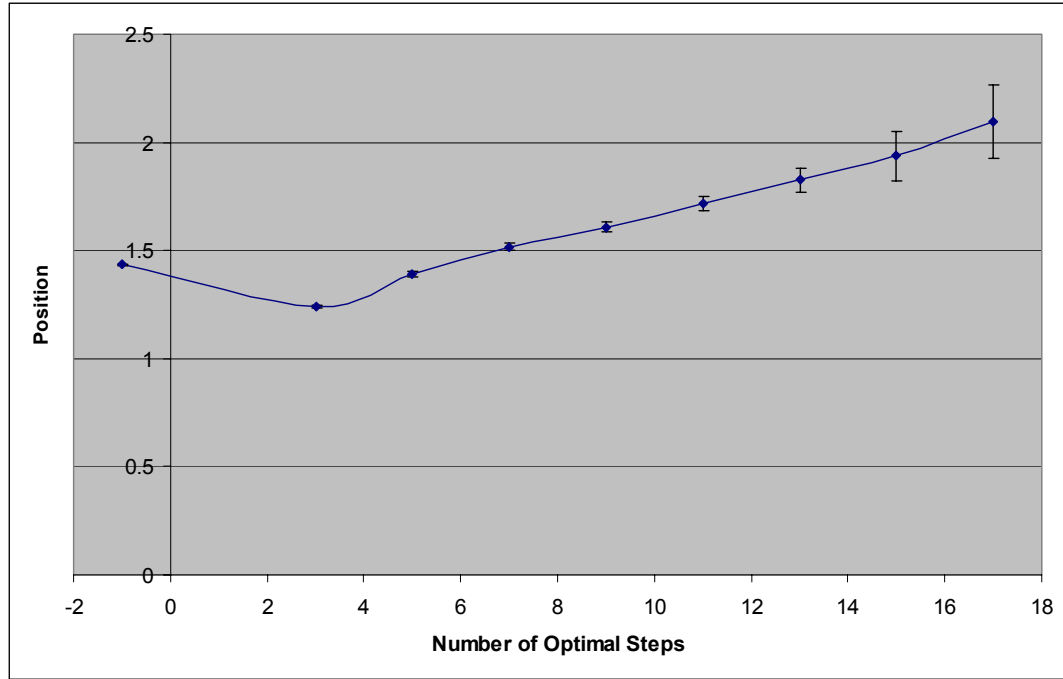
**Figure 35.**  $l_{min}$ : Average Fraction of Alternatives Examined

*Number of Alternatives:* We measured the number of alternatives generated by negotiators (*alts*) as well as the number of these alternative requests that resulted in an affirmative offer (*altsExamined*). For each test scenario, the fraction of alternatives examined (*altsExamined/alts*) was measured. Only those test cases were considered for which a non-zero value of *alts* was recorded. Averages of these fractions were computed for each unique value of  $l_{min}$ . The graph in Figure 35 shows these fractions with 99% confidence intervals.

We can see a roughly linear (though gentle) increase in the fraction of alternatives examined with an increase in  $l_{min}$ , though the curve may eventually flatten out in a way similar to the curve representing the fraction of intermediate requests granted. At lower values of  $l_{min}$ , fewer alternatives exist, and the correct alternative apparently gets examined earlier, leading to a smaller fraction value. In fact, this is reflected in the graph below as well, which indicates the average number of alternatives examined before a valid one is obtained. When  $l_{min} = -1$ , since negotiations fail, almost all fraction values are equal to 1; this is because the negotiation protocol engine attempts an exhaustive search. To make this clearer, some alternatives may succeed, but the satisfaction of a few affirmative offers may not be enough to allow a negotiation to succeed. Our policy resolution tree is an AND-OR tree, and all branches of an AND node must be satisfied for the policy resolution to succeed. Therefore, the average fraction of alternatives examined is  $\sim 0.96$ .

The fractions of alternatives examined in Figure 35 between 0.5 and 0.7. But these cover a range of negotiations whose steps vary from 3 to 79. For realistic

negotiations, the fraction is therefore significantly below 1. Better heuristics based on extra knowledge of the negotiating domains may improve the efficiency.



**Figure 36.**  $l_{min}$ : Average Position of the First Valid Alternative

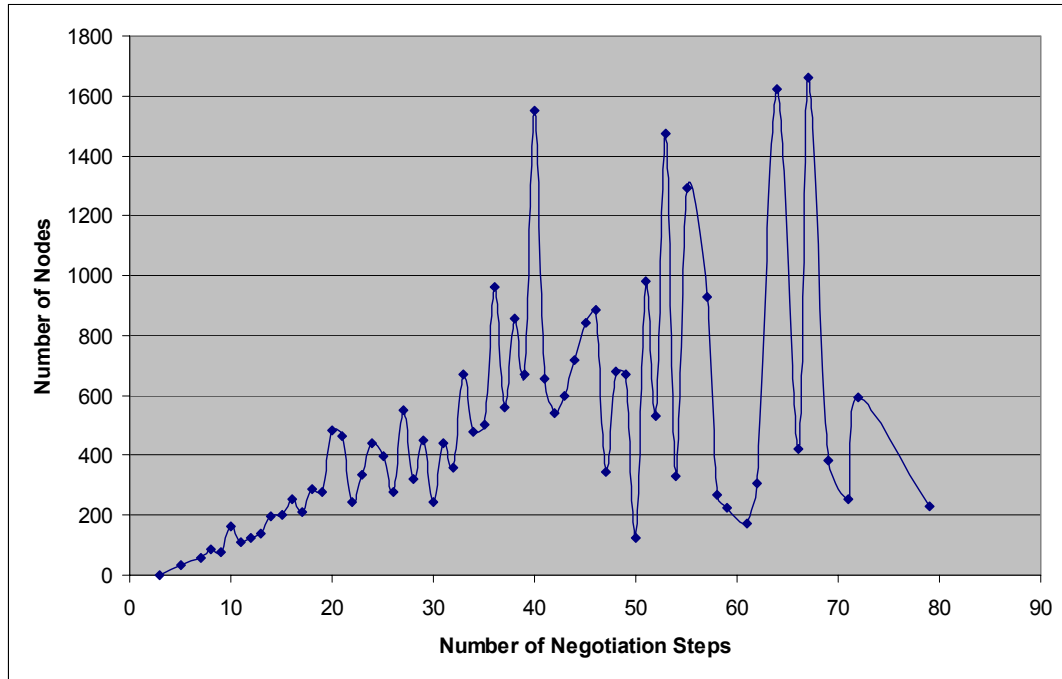
Given a set of generated alternatives ( $S_A$ ), a negotiator sends  $P_A$  ( $P_A \leq S_A$ ) in the form of requests. Therefore,  $P_A$  is the position of the first valid alternative. The above graph (Figure 36) indicates the average value of  $P_A$ ; roughly, we can see it increasing linearly with an increase in  $l_{min}$ . This conclusion is validated by the fact that running a linear regression on the average position data set gives us an  $R^2$  value of 0.99 (the slope is 0.06). But for the wide range of measured negotiations (a large number of which lie beyond our estimated “realistic” range), the second alternative appears to be the correct one on average. *Note:* These results are drawn only from scenarios that actually contain

valid alternatives; there are a number of cases where no valid alternatives were found, and these cases are not represented in the above graph.

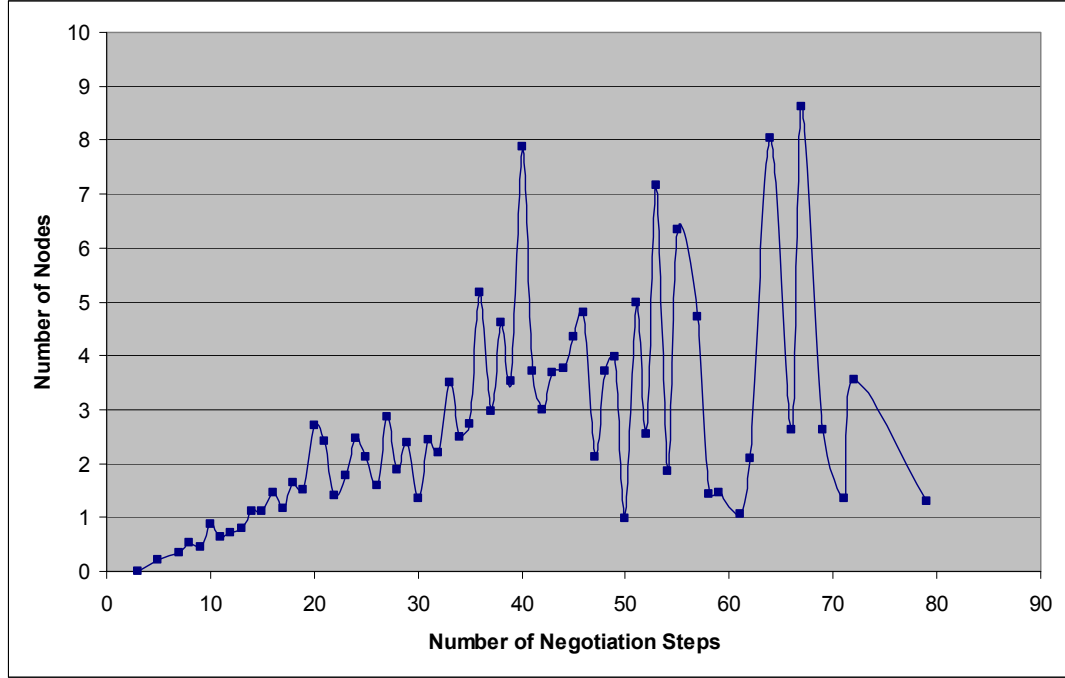
#### 9.5.5.1.2 Length of Actual Negotiation ( $l_{neg}$ )

We aggregated all the results obtained for all  $\langle b_{max}, d_{max} \rangle$  pairs and classified them based on the actual number of steps used by the negotiation protocol. The shortest negotiation (either successful or failed) obviously consisted of three steps. Results based on individual metrics are indicated below.

*Number of Policy Resolution Tree Nodes:* Figures 37 and 38 indicate how the average number of nodes examined varies with  $l_{neg}$ .



**Figure 37.**  $l_{neg}$ : Number of Nodes Examined During Policy Resolution



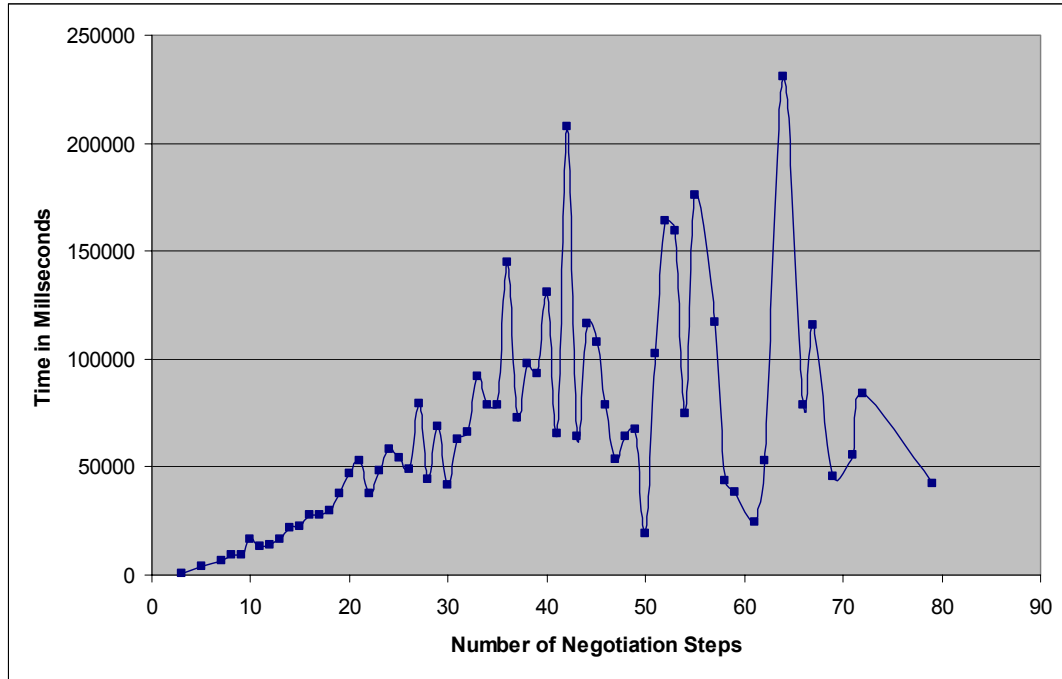
**Figure 38.**  $l_{neg}$ : Number of Nodes Examined per Unit Database (Number of Nodes / Number of Policy Statements in the Database)

The graphs in Figures 37 and 38 show meaningful trends for values of  $l_{neg}$  lower than 40-45. This limit well exceeds the length of negotiations in realistic scenarios. The number of nodes seems to increase in a roughly linear, or a very gentle exponential, manner. Since the number of steps correlates with the depth of the distributed policy resolution tree, we would expect an exponential increase in the number of nodes. Therefore, an increase that appears to be linear to the naked eye reflects well on the performance of the negotiation protocol.

The reason why the points at higher values of  $l_{neg}$  seem to be scattered randomly is because the number of data points is very small, and our confidence in those values being representative is fairly low.

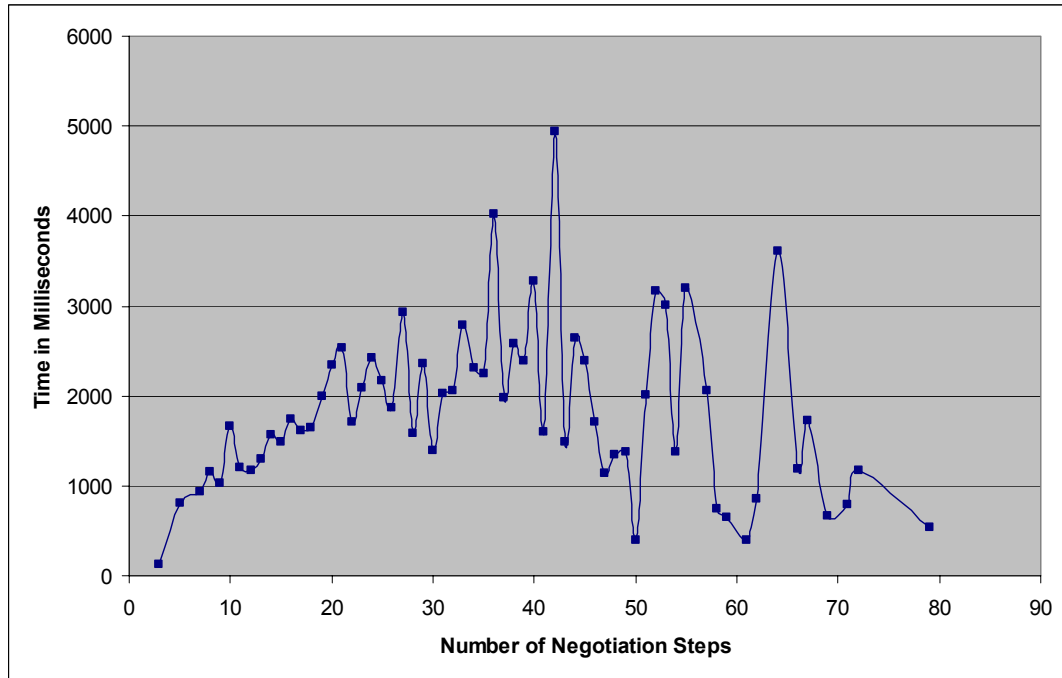
Normalizing the number of nodes with the size of the database yields a virtual replica of the earlier curve, conveying no new information in the process.

*Processing Time:* Figures 39 to 42 indicate how the processing time for policy resolution varies with the actual negotiation length.



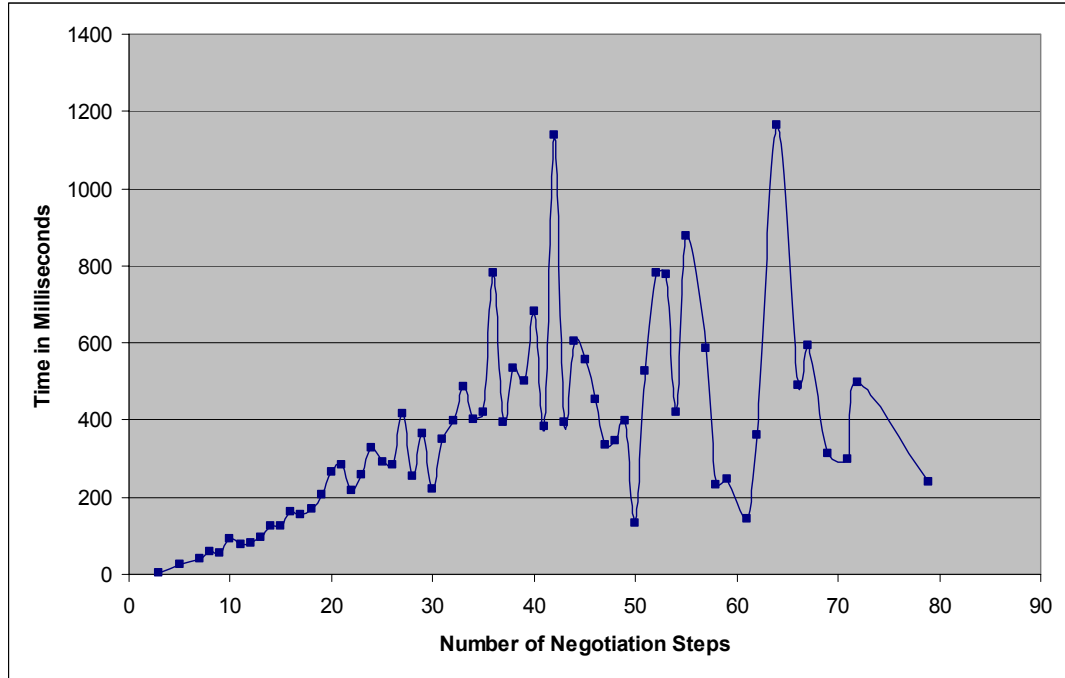
**Figure 39.**  $l_{neg}$ : Total Processing Time for Policy Resolution

The graph in Figure 39 is, not surprisingly, very similar to the graph in Figure 37 (number of nodes versus negotiation length). An exponential increase in the processing time with an increase in  $l_{neg}$  is more evident here. The portion of the curve beyond  $l_{neg} = \sim 40$  does not yield any meaningful information due to lack of data points.

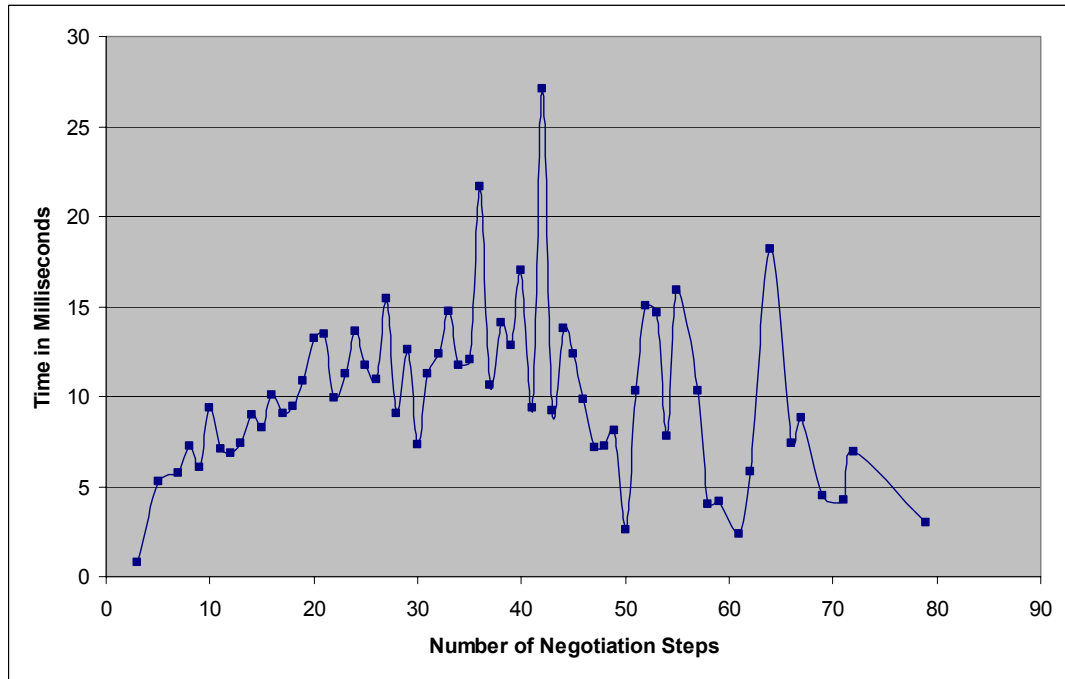


**Figure 40.**  $I_{neg}$ : Average Processing Time for Policy Resolution per Negotiation Step

The curve in Figure 40 indicates that the processing time per negotiation step increases at a much slower rate asymptotically compared to the total time. This is an encouraging result, as it means that longer negotiations are more efficient than shorter ones, even though they will end up consuming more time and bandwidth just because they happen to be longer.



**Figure 41.**  $I_{neg}$ : Total Processing Time for Policy Resolution per Unit Database (Total Processing Time / Number of Policy Statements in the Database)

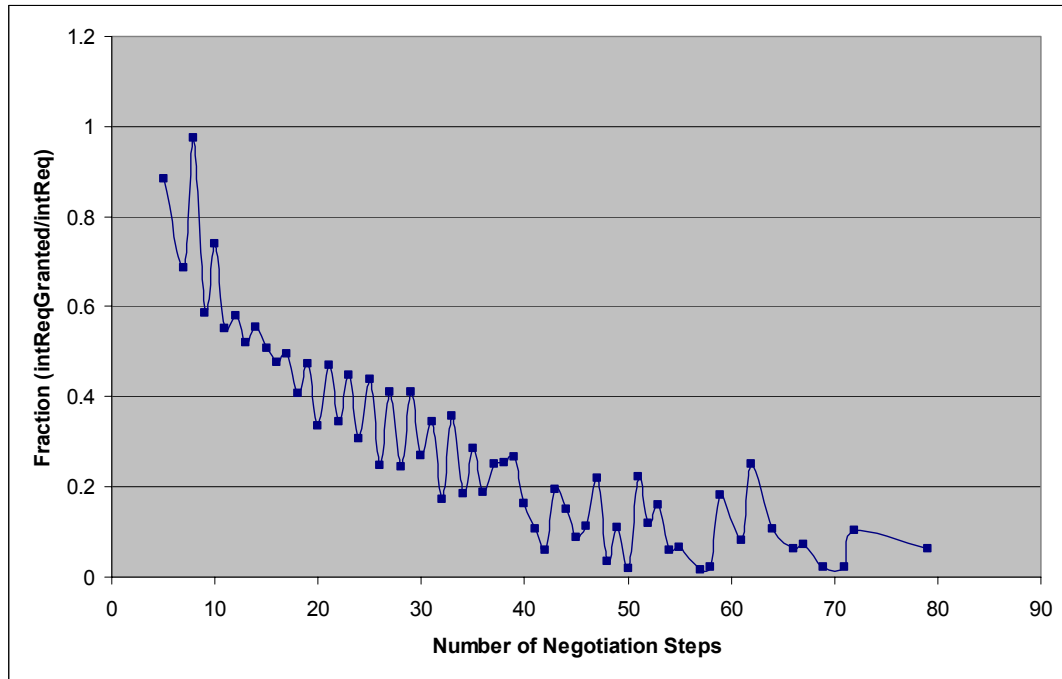


**Figure 42.**  $I_{neg}$ : Average Processing Time for Policy Resolution per Negotiation Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database)



Normalizing the processing time with the size of the databases again seems to yield no new information, as is evident from the Figures 41 and 42.

*Number of Intermediate Requests:* We measured the number of intermediate requests posed by a negotiator to another (*intReq*) as well as the number of such requests that resulted in an affirmative offer (*intReqGranted*). For each test scenario, the fraction of requests granted (*intReqGranted/intReq*) was measured. We considered only those test cases for which a non-zero value of *intReq* was recorded. Averages of these fractions were computed for each unique value of  $l_{neg}$ .

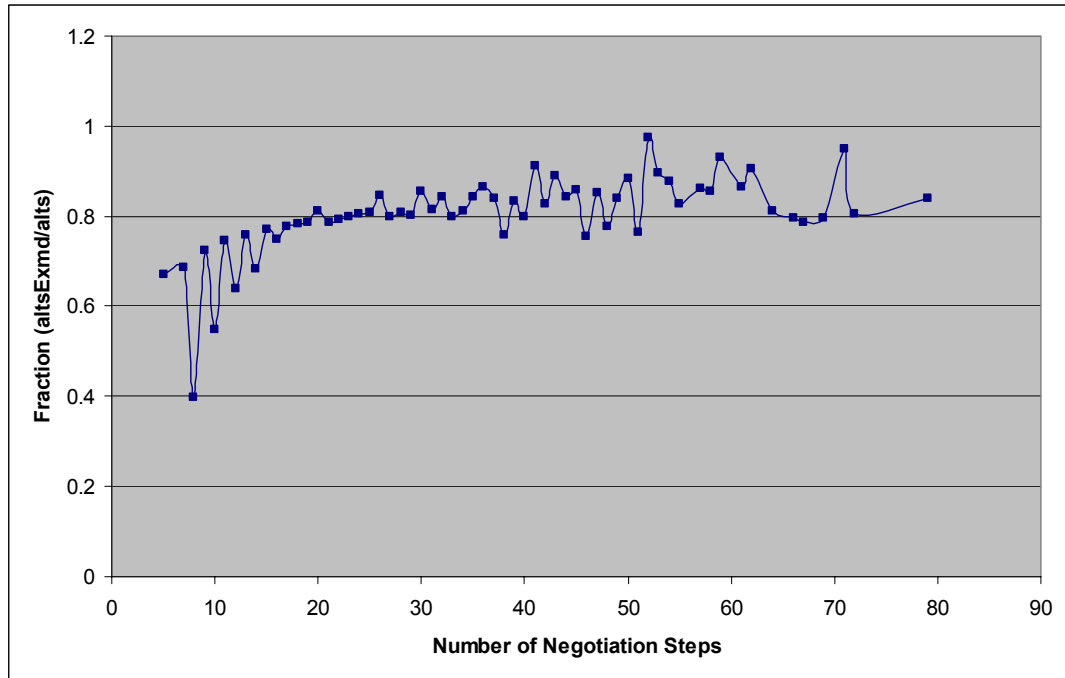


**Figure 43.**  $l_{neg}$ : Average Fraction of Intermediate Requests Granted

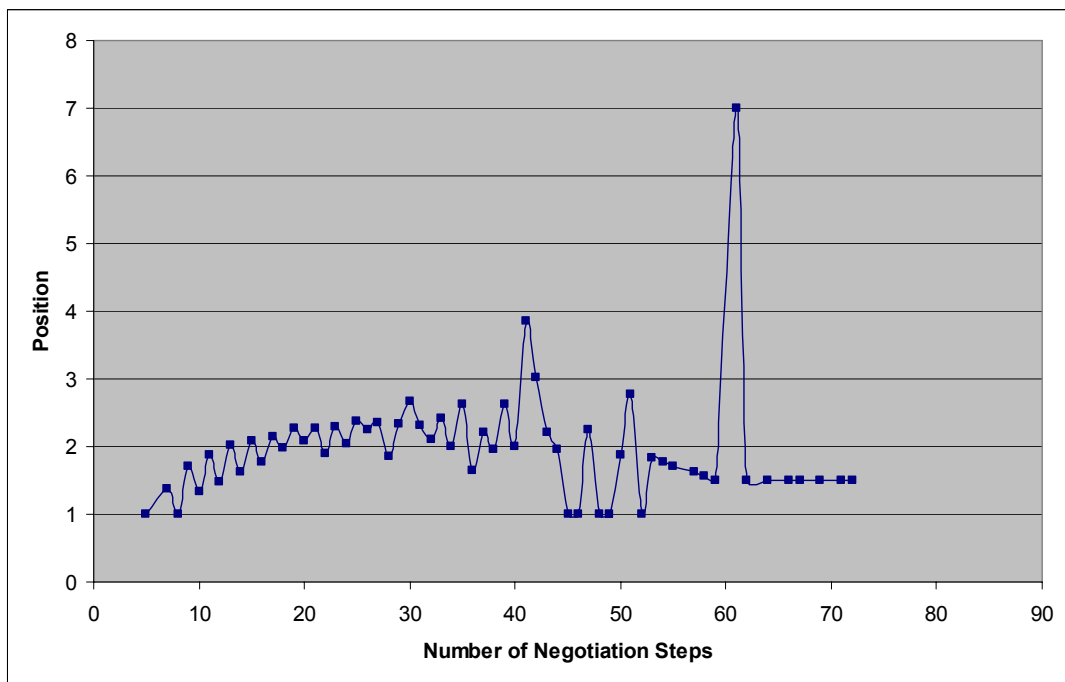
The graph in Figure 43 indicates a general decreasing trend in the fraction of intermediate requests granted, though we can observe oscillations between adjacent data

points. The oscillations are probably because adjacent negotiations are often distinguished by an extra request-affirmative offer pair of steps, thereby changing the average fraction value. The fraction falls from a peak of almost 1 (for a short negotiation) to almost 0 (for negotiations longer than 50 steps). One reason for this trend may be that longer negotiations tend to be failed negotiations, in which a large number of intermediate requests are made, but virtually none granted. Still, this curve does indicate that the negotiation protocol is fairly good at maintaining privacy, because even though the number of intermediate requests granted increases with an increase in the number of steps, it is only a small fraction of what could potentially be granted.

*Number of Alternatives:* We measured the number of alternatives generated by negotiators (*alts*) as well as the number of such requests that resulted in an affirmative offer (*altsExamined*). For each test scenario, the fraction of alternatives examined (*altsExamined*/*alts*) was measured. Only those test cases were considered for which a non-zero value of *alts* was recorded. Averages of these fractions were computed for each unique value of  $l_{neg}$ .



**Figure 44.**  $I_{neg}$ : Average Fraction of Alternatives Examined



**Figure 45.**  $I_{neg}$ : Average Position of the First Valid Alternative

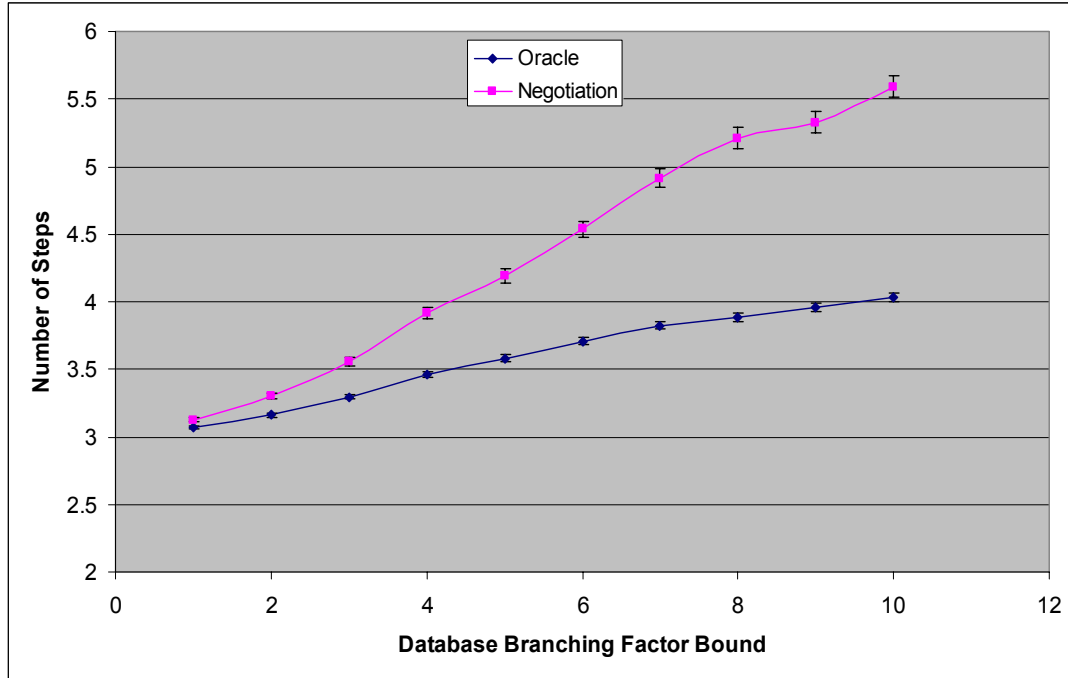
From the graph in Figure 44, we can see that the fraction of examined alternatives seems to rise till  $l_{neg} = \sim 20$  and remain constant ( $\sim 0.8$ ) beyond that. *Note*: most of the data points lie below  $l_{neg} = 20$ . This indicates that longer negotiations tend to examine a roughly constant percentage of the number of alternatives generated. As a metric of efficiency, this is an encouraging result as it indicates that using better heuristics would result in performance gains (i.e., a lower fraction of alternatives will be examined).

On average, the graph in Figure 45 indicates the position of the first valid alternative roughly increases with an increase in  $l_{neg}$  for the range  $l_{neg} \leq \sim 20$ . Beyond that, the position seems to be roughly constant, and is  $\sim 2$ . This means that even with an increase in the number of negotiation steps, the probability of hitting a valid alternative after examining two alternatives are high on average. *Note*: These results are drawn only from scenarios that actually contain valid alternatives; there are a number of cases where no valid alternatives were found, and these cases are not represented in the above graph.

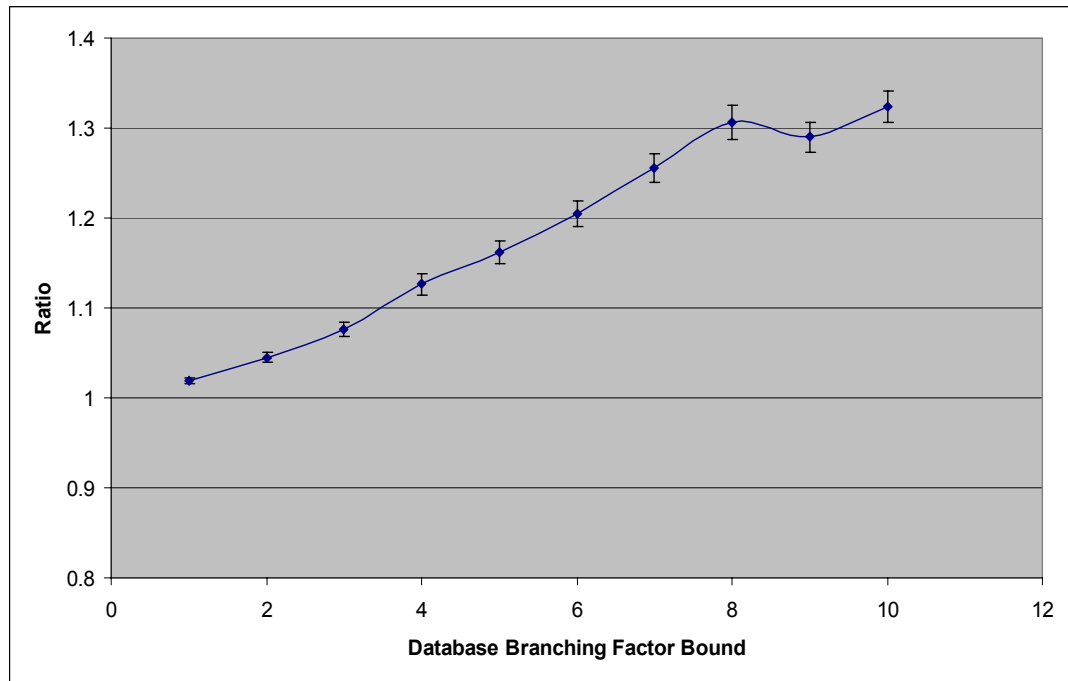
#### 9.5.5.1.3 Bound on Tree Branching Factor ( $b_{max}$ )

We aggregated all results and classified them based on the branching factor of the test scenarios that yielded them. Results based on individual metrics are indicated below.

*Number of Steps*: Figures 46 and 47 show how the number of optimal and actual negotiation steps increase with an increase in  $b_{max}$ . We can observe not only how the number of steps in each type of policy resolution varies (Figure 46) but also how the ratio



**Figure 46.**  $b_{max}$ : Average Number of Policy Resolution Steps



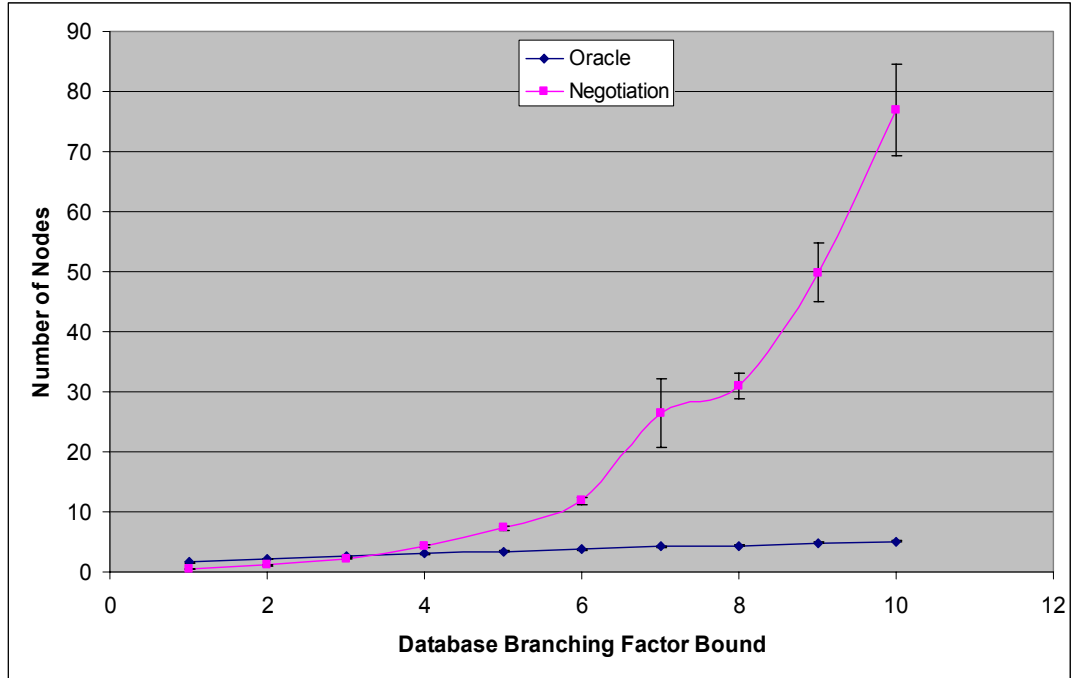
**Figure 47.**  $b_{max}$ : Policy Resolution Step Ratio (Number of Negotiation Steps / Optimal Number of Steps)

of actual and optimal (least number of) negotiation steps in each test scenario varies (Figure 47) with an increase in the branching factor bound. For proper comparison, the oracle considers the number of steps in a failed negotiation to be equal to 3, as this is the least number of steps in which a negotiation can fail. All quantities are reported with 99% confidence intervals.

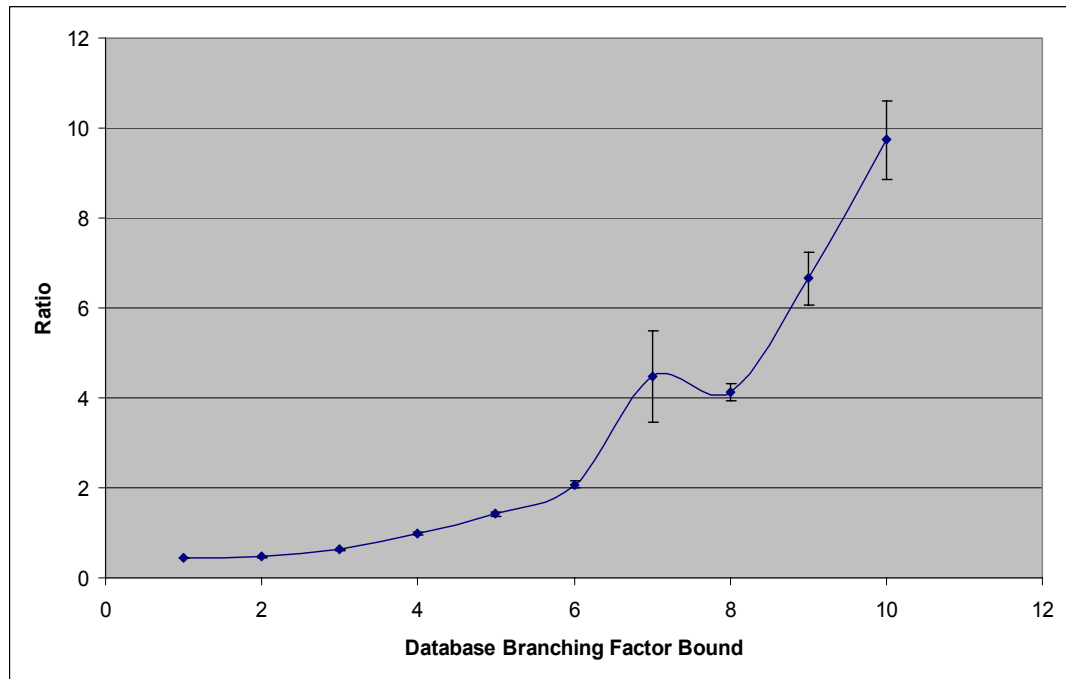
We can observe a monotonic increase in the number of optimal steps, actual negotiation steps, and the ratio of actual-to-optimal negotiation steps, with  $b_{max}$ . The increase of each curve appears to be roughly linear. This conclusion is validated by the following regression statistics:

- Optimal Policy Resolution Steps (Figure 46):  $R^2 = 0.98$ , slope = 0.11.
- Negotiation Steps (Figure 46):  $R^2 = 0.99$ , slope = 0.3.
- Policy Resolution Step Ratio (Figure 47):  $R^2 = 0.97$ , slope = 0.04.

The overall variation with respect to the database breadth bound is quite low. The average optimal length of a negotiation increases from 3 ( $b_{max} = 3$ ) to 4 ( $b_{max} = 4$ ). Likewise, actual negotiation lengths increase from 3 to  $\sim 5.5$  over the same range. The ratio increases to a maximum of  $\sim 1.3$ , which means that even for test cases with the largest branching factor bound, a negotiation protocol will take at most 1.3 times the least possible number of steps on average.



**Figure 48.**  $b_{max}$ : Average Number of Nodes Examined



**Figure 49.**  $b_{max}$ : Ratio of Number of Nodes Examined (Negotiation Tree Nodes / Oracular Tree Nodes)

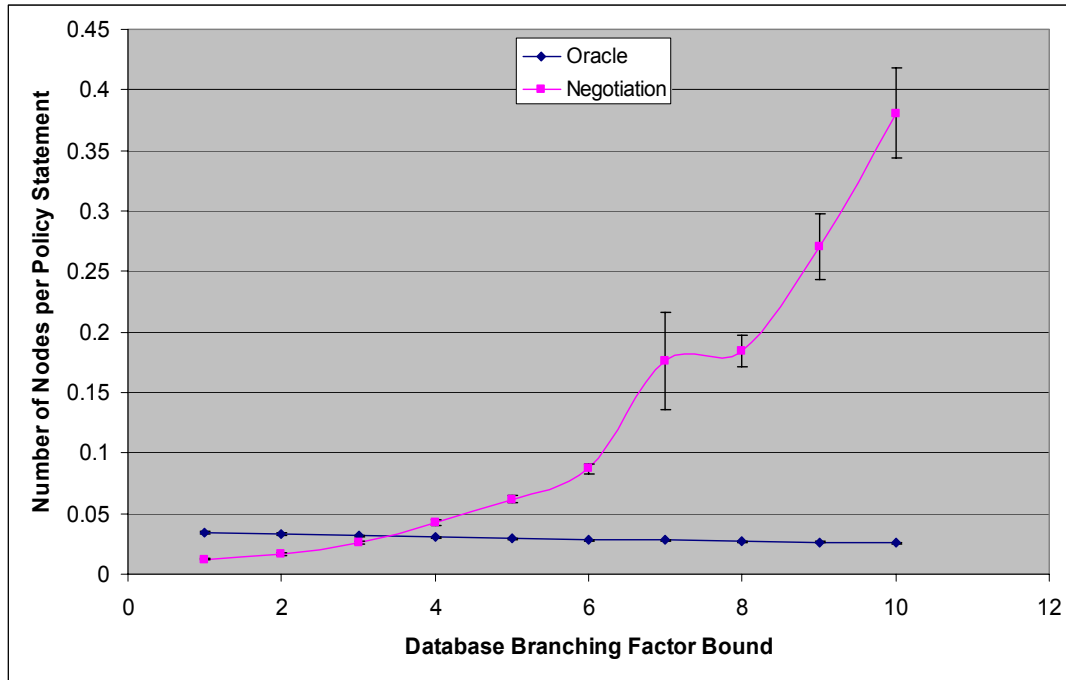
*Number of Policy Resolution Tree Nodes:* Figures 48 and 49 show how the number of nodes examined increases with increase in  $b_{max}$ . The graph in Figure 48 compares the numbers obtained for both kinds of policy resolution individually. The graph in Figure 49 indicates how the ratio (#Negotiation nodes/#Oracular nodes) varies with  $b_{max}$ . All quantities are reported with 99% confidence intervals.

The negotiation curve in Figure 48 indicates a super-linear increase, with a sharp increase beyond  $b_{max} = 6$ . We would expect the growth to be polynomial, with the exponent lying somewhere between 1 and 20, given that the size of a tree is polynomial in terms of its branching factor. The number of nodes examined by the oracle, on the other hand, seems to increase very gently (almost linear to the naked eye, though a polynomial increase would be a more plausible explanation, because the number of nodes in a tree varies as a polynomial function of its branching factor, everything else being constant). The ratio curve in Figure 49 is similar in shape to the negotiation curve in Figure 48. It grows sharply from a factor of 2 (when  $b_{max} = 6$ ) to  $\sim 10$  ( $b_{max} = 10$ ). These results show that the tree branching factor may seriously impact performance at moderately high values ( $b_{max} > 6$ ). In our test scenarios,  $b_{max}$  never reached a value of 6 or higher, but there is no guarantee that more complex policy databases in larger ubiquitous domains will not reach higher values of  $b_{max}$ . Therefore we seek to find ways to mitigate this performance hit as part of future work.

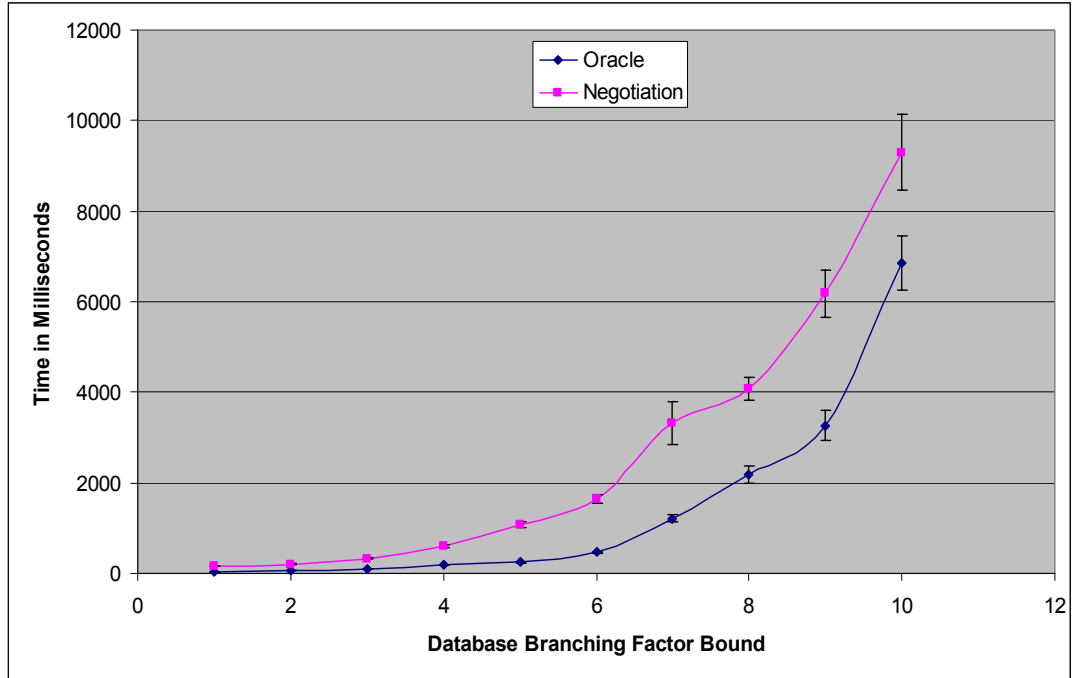
When we normalize the number of nodes with the size of the database (see Figure 50), the negotiation protocol curve retains its shape. On the other hand, the oracular curve indicates a decreasing trend. A plausible explanation for this is as follows. Database sizes



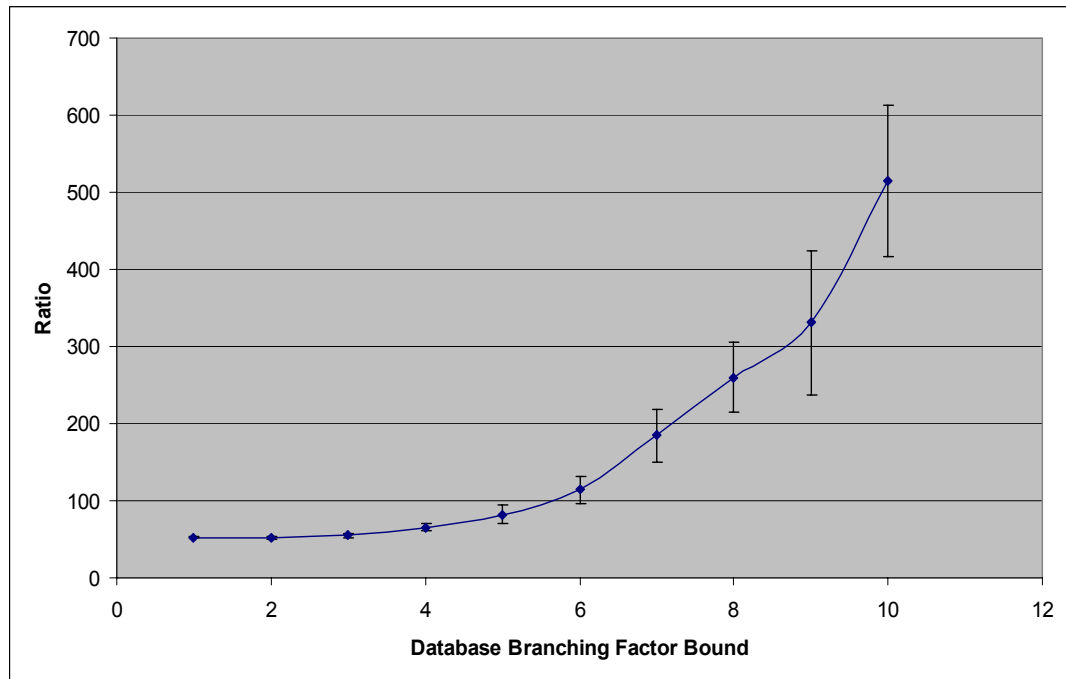
increase with an increase in  $b_{max}$ , but the number of irrelevant statements also increases in significant numbers. Especially, failed negotiations involve few or no node examinations by the oracle. Since the negotiation protocol exhaustively examines the relevant search space until failure, it cannot escape a large number of node examinations at higher values of  $b_{max}$ . Therefore, dividing the number of examined nodes by the database size tends to have a much more positive effect on oracular performance as compared to the negotiation performance.



**Figure 50.  $b_{max}$ :** Average Number of Nodes per Unit Database (Number of Nodes / Number of Policy Statements in the Database)



**Figure 51.**  $b_{max}$ : Processing Time for Policy Resolution

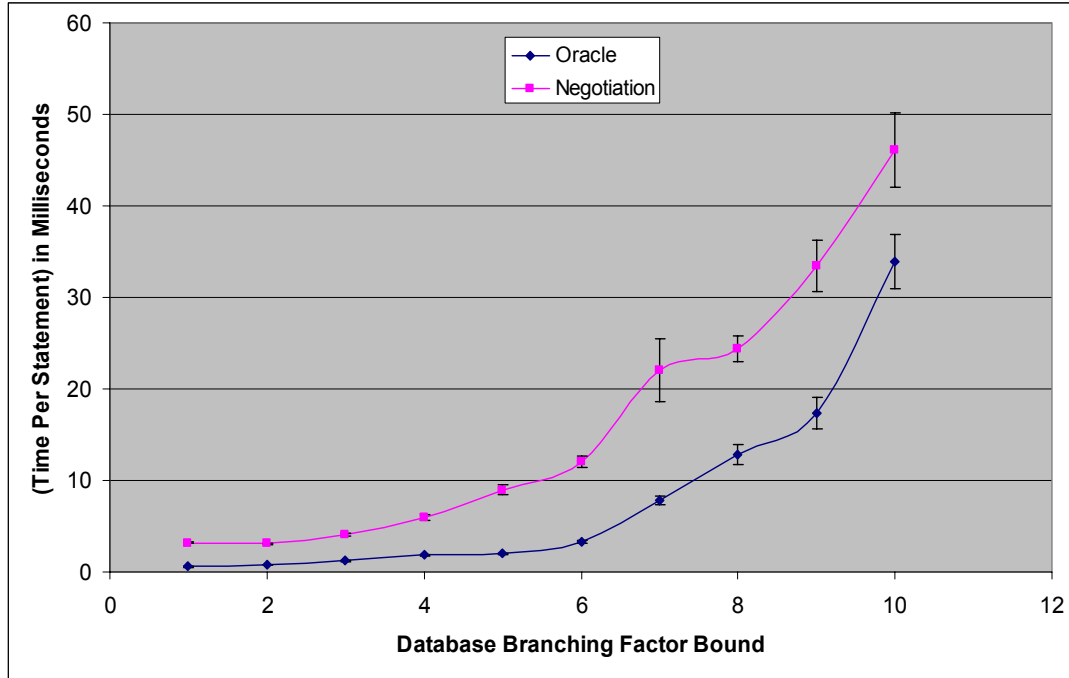


**Figure 52.**  $b_{max}$ : Ratio of Processing Times for Policy Resolution (Negotiation Time / Oracular Time)

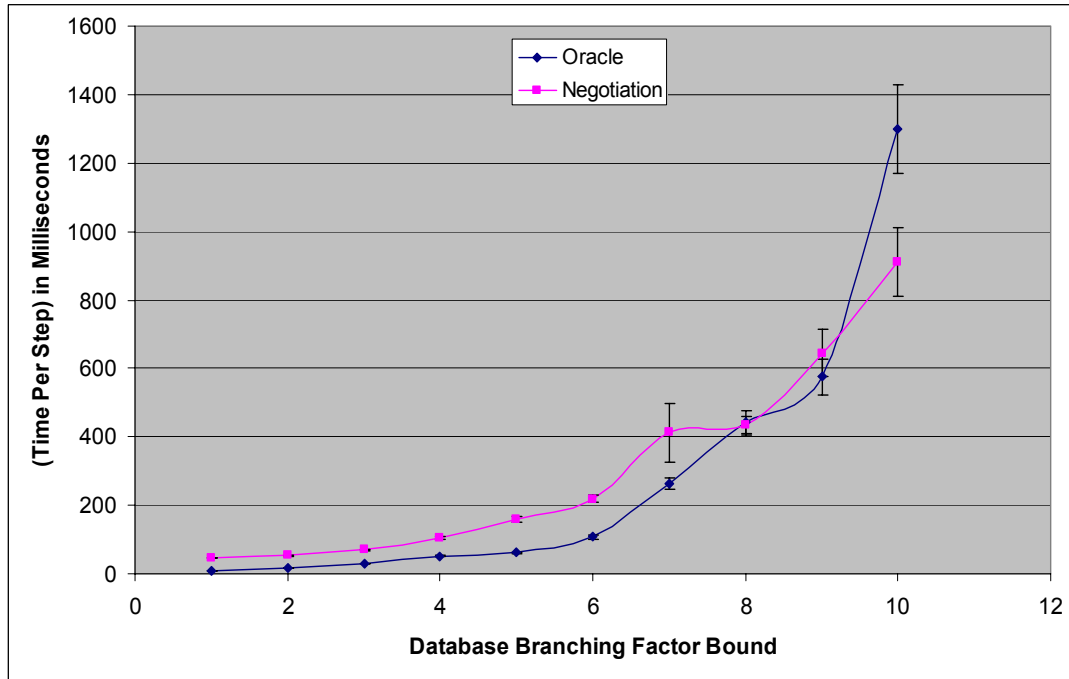
*Processing Time:* Figures 51 to 55 show how the processing time increases with an increase in  $b_{max}$ . The graph in Figure 51 compares the numbers obtained for both kinds of policy resolution individually. The graph in Figure 52 indicates how the ratio (Negotiation time/Oracular time) varies with  $b_{max}$ . All quantities are reported with 99% confidence intervals.

The processing time for negotiation (see Figure 51) is strongly correlated with the number of examined nodes (see Figure 48), i.e., it increases sharply beyond  $b_{max} = 6$ ; our conjecture is that this is a polynomial, and not an exponential, increase as tree sizes are polynomial in terms of their branching factors. The increase in oracular processing time increases is not too much lower than the negotiation processing time; in contrast, the number of nodes examined by an oracle is much fewer than the number of nodes examined by the negotiation protocol (see Figure 48). The average of processing time ratios measured for each scenario also increases in a polynomial (super-linear) manner, as we can observe from Figure 52. The maximum average time does not exceed 10 seconds, even for  $b_{max} = 10$ , which is significantly higher than we would observe in real-world scenarios. Realistic scenarios, where  $b_{max} \leq 6$ , have significantly low processing times on average for both oracular and distributed policy resolution.

Normalizing the processing times by database sizes, as indicated in the graph in Figure 53, seems to make no difference to the nature of the curves. Database sizes therefore seem to have no effect on the time comparisons between centralized and distributed policy resolutions.

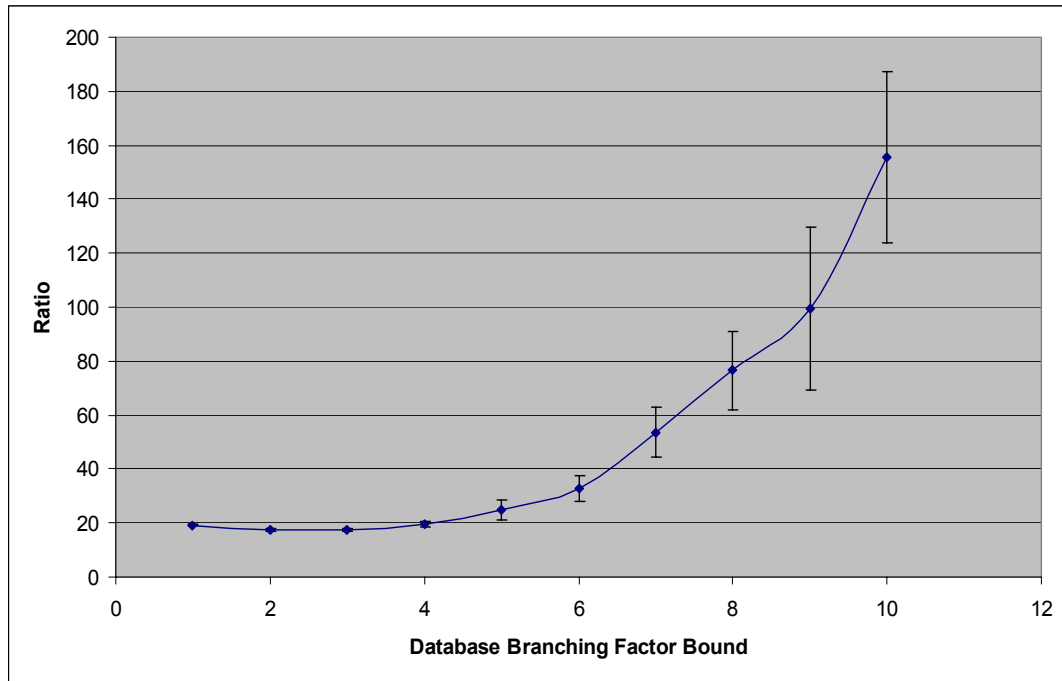


**Figure 53.**  $b_{max}$ : Total Processing Time for Policy Resolution per Unit Database (Processing Time / Number of Policy Statements in the Database)

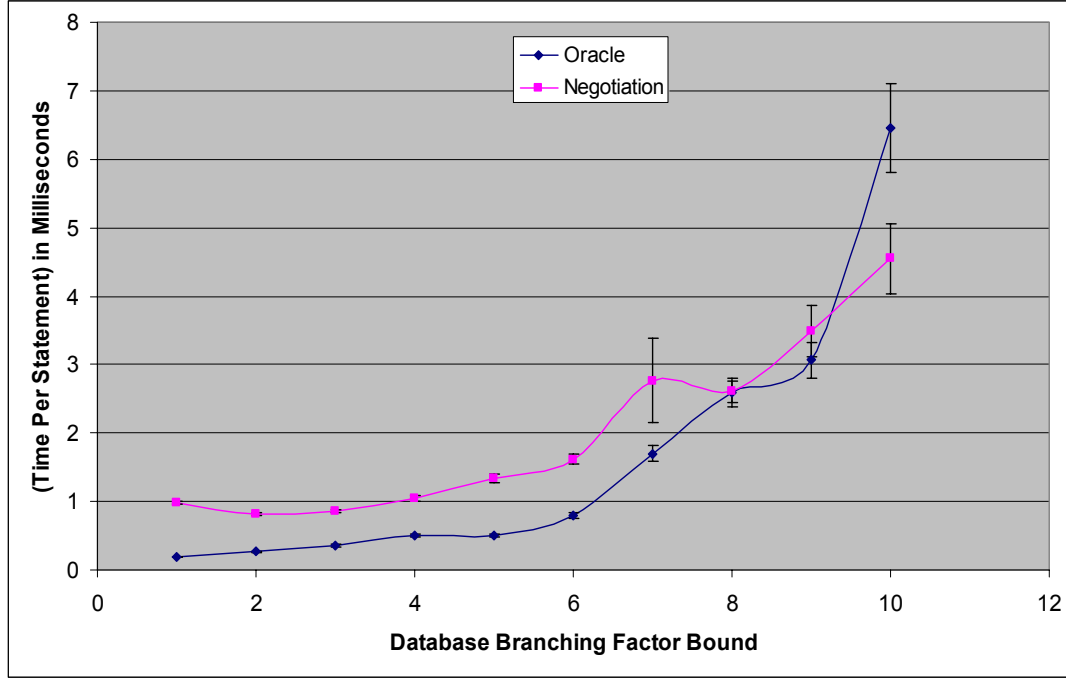


**Figure 54.**  $b_{max}$ : Average Processing Time for Policy Resolution per Policy Resolution Step

The graph in Figure 54 shows curves that are much closer to each other than the corresponding curves representing total processing times. Indeed, at higher values of  $b_{max}$ , the average processing time per step for an oracle is higher than the average processing time per step for a negotiation. For realistic branching factors ( $b_{max} \leq 6$ ), both the processing times per step and the ratio (see Figure 55) are fairly low. The graph in Figure 55, which indicates the averages of ratios of processing times per step for each test scenario, shows a surprisingly sharp increase beyond  $b_{max} = 6$ . As we can see, the confidence intervals for higher values of  $b_{max}$  are quite large; our conjecture is that certain individual results with high ratios end up pushing the averages (the average of ratios is always higher than the ratio of averages).



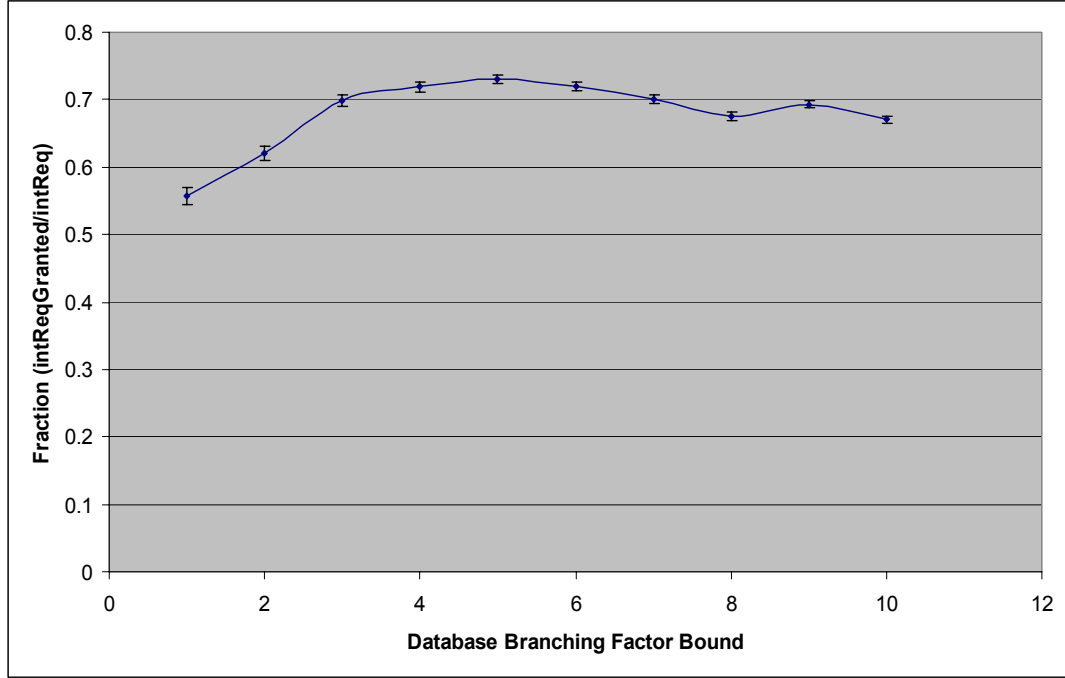
**Figure 55.**  $b_{max}$ : Ratio of Average Processing Time for Policy Resolution per Step (Negotiation Time / Oracular Time)



**Figure 56.  $b_{max}$ :** Average Processing Time for Policy Resolution Step per Unit Database (Average Processing Time / Number of Policy Statements in the Database)

As we have observed in the case of all other parameters and metrics, normalizing the processing times by database sizes, as indicated in the graph in Figure 56, seems to make no difference to the nature of the curves. Database sizes therefore seem to have no effect on these average time comparisons as well.

*Number of Intermediate Requests:* The graph in Figure 57 below shows how the fraction of intermediate requests granted ( $intReqGranted/intReq$ ) varies with  $b_{max}$ . Only those test cases were considered for which a non-zero value of  $intReq$  was recorded. All quantities are reported with 99% confidence intervals.

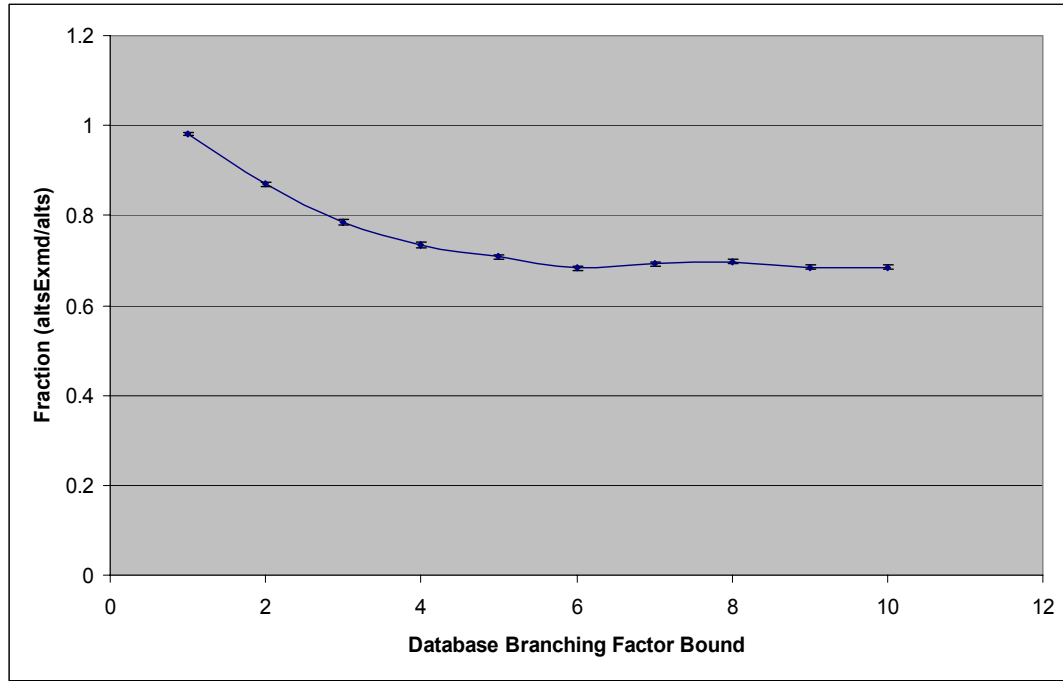


**Figure 57.**  $b_{max}$ : Average Fraction of Intermediate Requests Granted

We cannot discern any significant variation in the fraction of intermediate requests granted with change in branching factor, except for very low values of  $b_{max}$ . Overall, it appears that ~68% of intermediate requests posed will be granted on average, give or take a few percentage points. The fraction seems to be closer to 50% for test cases with low branching factor bounds. This may simply be because a larger number of failed negotiations occur at lower values of  $b_{max}$ , and has no bearing on the larger issue of privacy maintenance.

Therefore, we can conclude with high assurance that the privacy maintenance properties of the negotiation protocol does not vary with  $b_{max}$ .

*Number of Alternatives:* The graph in Figure 58 shows how the fraction of alternatives examined ( $altsExamined/alts$ ) varies with  $b_{max}$ . Only those test cases were considered for which a non-zero value of  $alts$  was recorded. All quantities are reported with 99% confidence intervals.

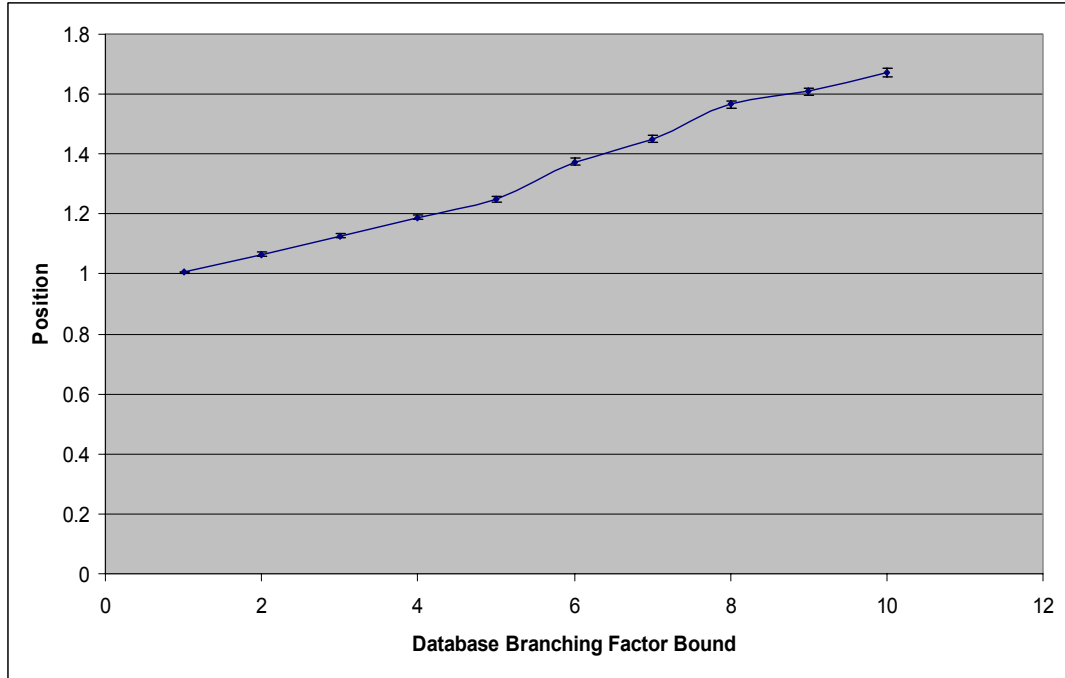


**Figure 58.**  $b_{max}$ : Average Fraction of Alternatives Examined

We can see that the fraction of examined alternatives seems to reach a constant terminal value of  $\sim 0.7$ . The reason for the fraction being close to 1 for lower values of  $b_{max}$  is probably because the test cases with lower branching factor bounds generate fewer alternatives, all or most of which are examined during the course of a negotiation.

We can conclude that the fraction of alternatives examined decreases roughly linearly for  $b_{max} \leq 5$  and remains constant at approximately 0.7 for  $b_{max} \geq 5$ .



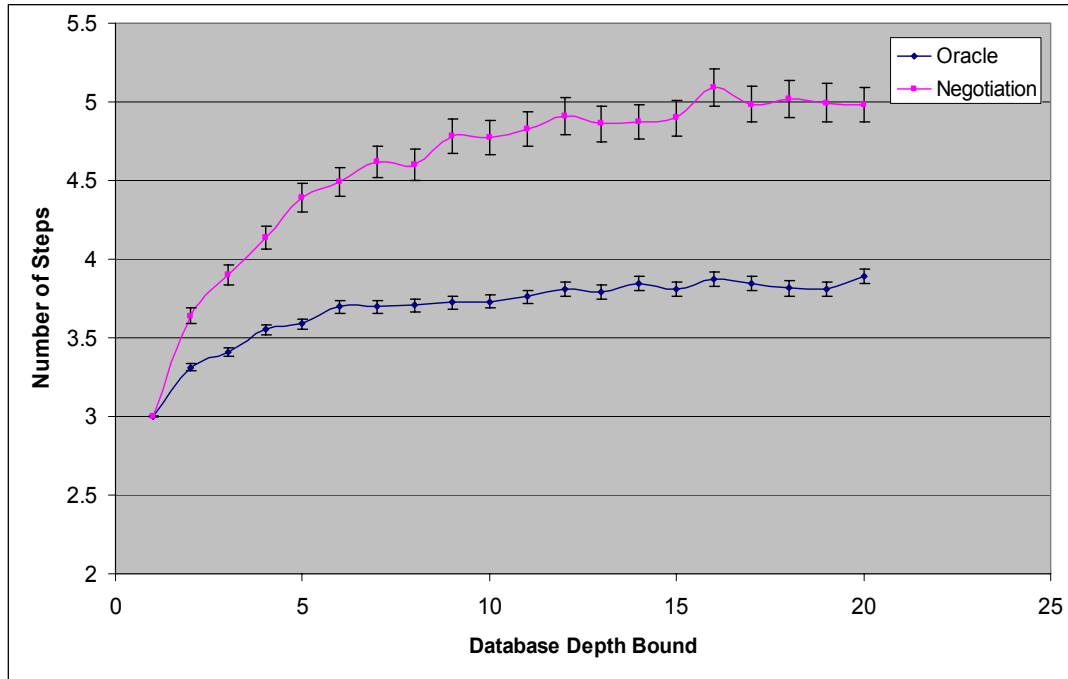


**Figure 59.  $b_{max}$ :** Average Position of the First Valid Alternative

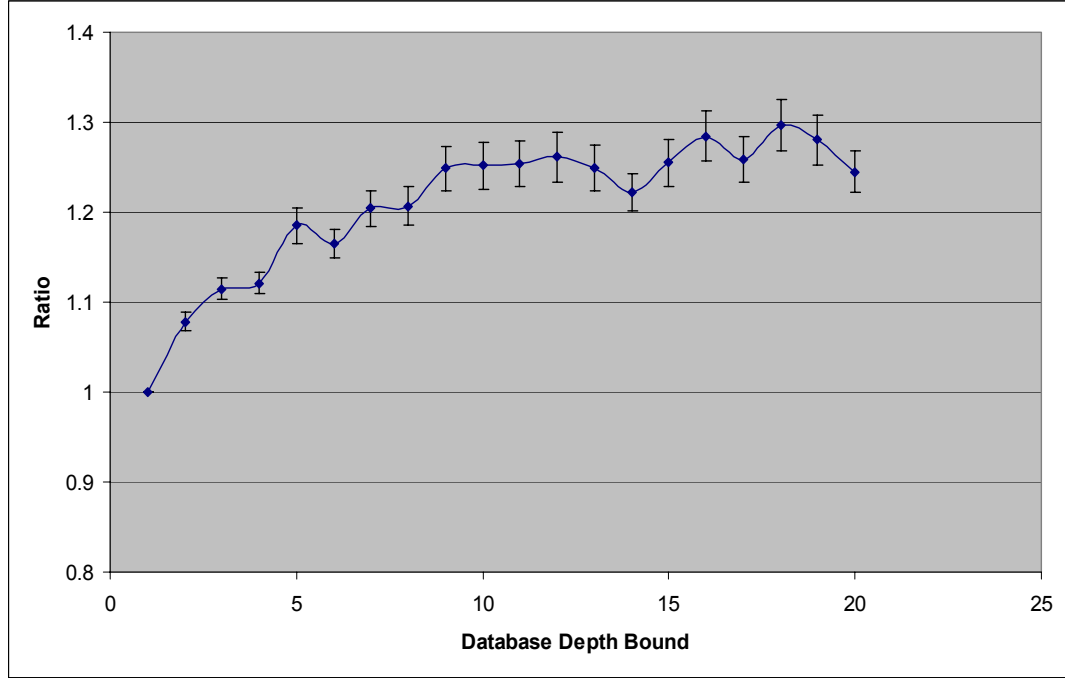
The average position of a valid alternative increases linearly with  $b_{max}$ , as is evident from the graph in Figure 59. This conclusion is also validated by the fact that running a linear regression on the average position data set gives us an  $R^2$  value of 0.99, (the slope is 0.08). The slope of the line is very low, and the position value does not exceed 1.7 on average even for the case with the highest branching factor bound ( $b_{max} = 10$ ). What this implies is that for  $b_{max} \leq 7$ , there is a higher probability of hitting a valid alternative at the first attempt, whereas for higher values of  $b_{max}$ , there is a higher probability that the first alternative will fail. *Note:* These results are drawn only from scenarios that actually contain valid alternatives; there are a number of cases where no valid alternatives were found, and these cases are not represented in the above graph.

#### 9.5.5.1.4 Bound on Tree Depth ( $d_{\max}$ )

*Number of Steps:* Figures 60 and 61 show how the number of optimal and actual negotiation steps increase with an increase in  $d_{\max}$ . We can observe not only how the number of steps in each type of policy resolution varies (see Figure 60), but also how the ratio of actual and optimal (least number of) negotiation steps in each test scenario varies (see Figure 61) with increase in depth. For proper comparison, the oracle was assumed to report the number of steps as 3 for failed negotiations. All quantities are reported with 99% confidence intervals.

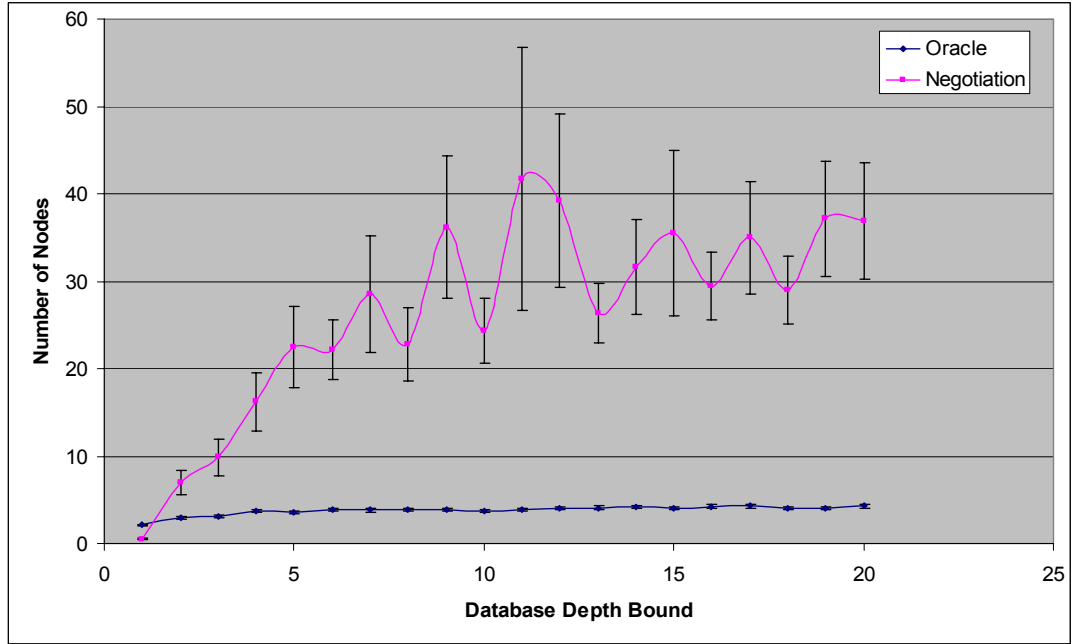


**Figure 60.**  $d_{\max}$ : Average Number of Policy Resolution Steps

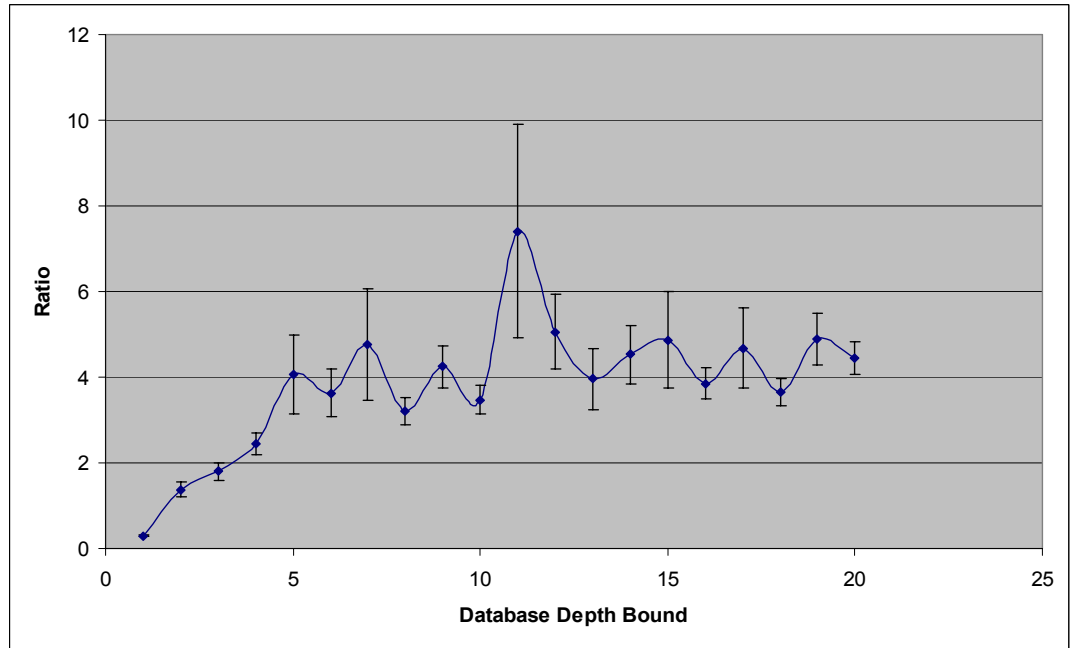


**Figure 61.**  $d_{max}$ : Policy Resolution Step Ratio (Number of Negotiation Steps / Optimal Number of Steps)

The number of steps can be seen to increase roughly monotonically with an increase in  $d_{max}$ . The increase is more pronounced at lower values of  $d_{max}$  (less than 10), and the curves almost flatten out for higher depth values. The graph in Figure 61 indicates that in the worst case, the number of steps taken for a negotiation is at 1.3 times the optimal (least) number of steps possible on average. As this ratio does not seem to increase (or increases very gently) for high depth values, we can conclude that an increase in  $d_{max}$  has virtually no negative effect on the length of a negotiation.



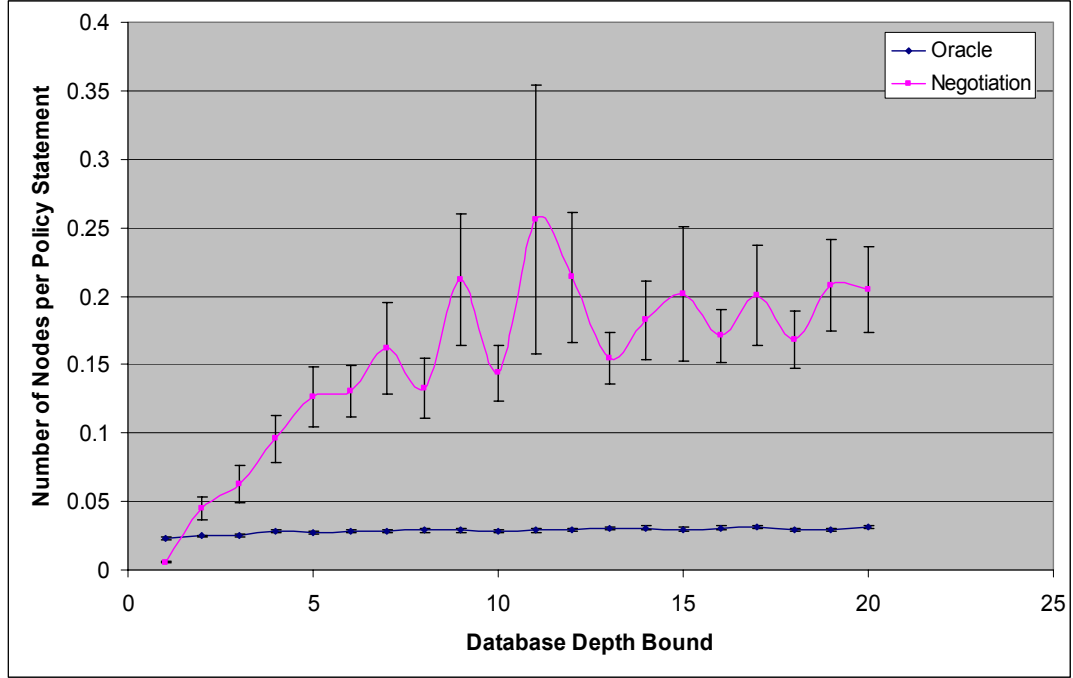
**Figure 62.**  $d_{max}$ : Average Number of Nodes Examined



**Figure 63.**  $d_{max}$ : Ratio of Number of Nodes Examined (Negotiation Tree Nodes / Oracular Tree Nodes)

*Number of Policy Resolution Tree Nodes:* Figures 62 and 63 show how the number of nodes examined increases with an increase in  $d_{max}$ . The graph in Figure 62 compares the numbers obtained for both kinds of policy resolution individually. The graph in Figure 63 indicates how the ratio (#Negotiation nodes/#Oracular nodes) varies with  $d_{max}$ . All quantities are reported with 99% confidence intervals.

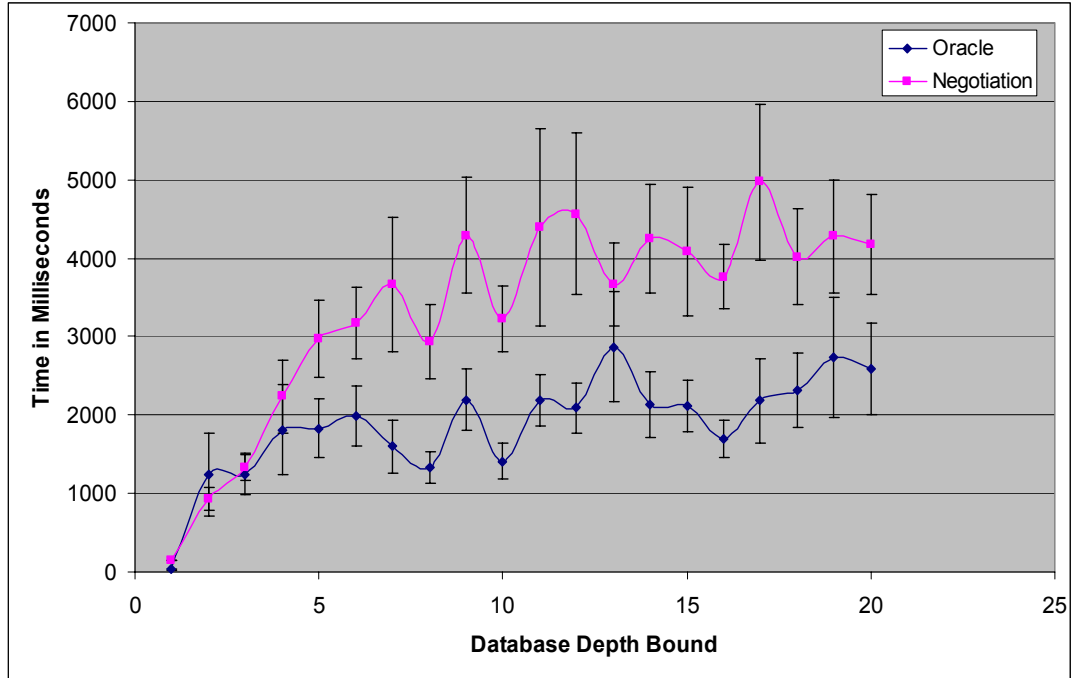
The size of a balanced tree increases exponentially with an increase in depth. The above graphs do not present a picture of exponential increase in the number of nodes examined, though. What we can observe is that the number of nodes examined during a negotiation increases linearly for smaller values of  $d_{max}$  (less than 10) and remains approximately constant (though our confidence intervals indicate significant margins of error) for higher values. Similarly, the increase in number of nodes examined by an oracle is only pronounced at very low depths, and is almost constant thereafter. The ratio curve (see Figure 63) is almost identical to the curve representing the number of nodes examined during negotiation (see Figure 62). This is because the number of nodes examined by the oracle is almost constant and is significantly lower (a factor of 4 to 10 at higher depths) than the number of nodes examined during negotiation.



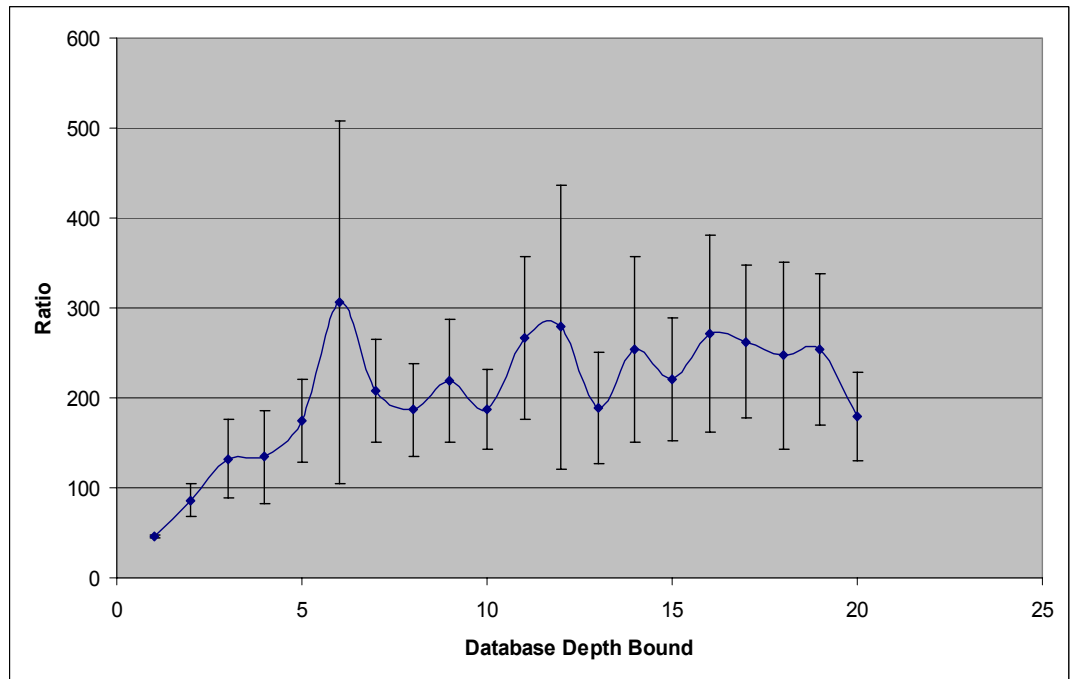
**Figure 64.  $d_{max}$ :** Average Number of Nodes per Unit Database (Number of Nodes / Number of Policy Statements in the Database)

From the graph in Figure 64, we can see that the number of policy statements in the database makes no difference to the comparison between the performance of the oracle and the negotiation protocol; increase in database size results in an increase in the number of examined nodes by an equal proportion for both.

*Processing Time:* The graphs below show how the processing time increases with an increase in  $d_{max}$ . The graph in Figure 65 compares the numbers obtained for both kinds of policy resolution individually. The graph in Figure 66 indicates how the ratio (#Negotiation time/#Oracular time) varies with  $d_{max}$ . All quantities are reported with 99% confidence intervals.

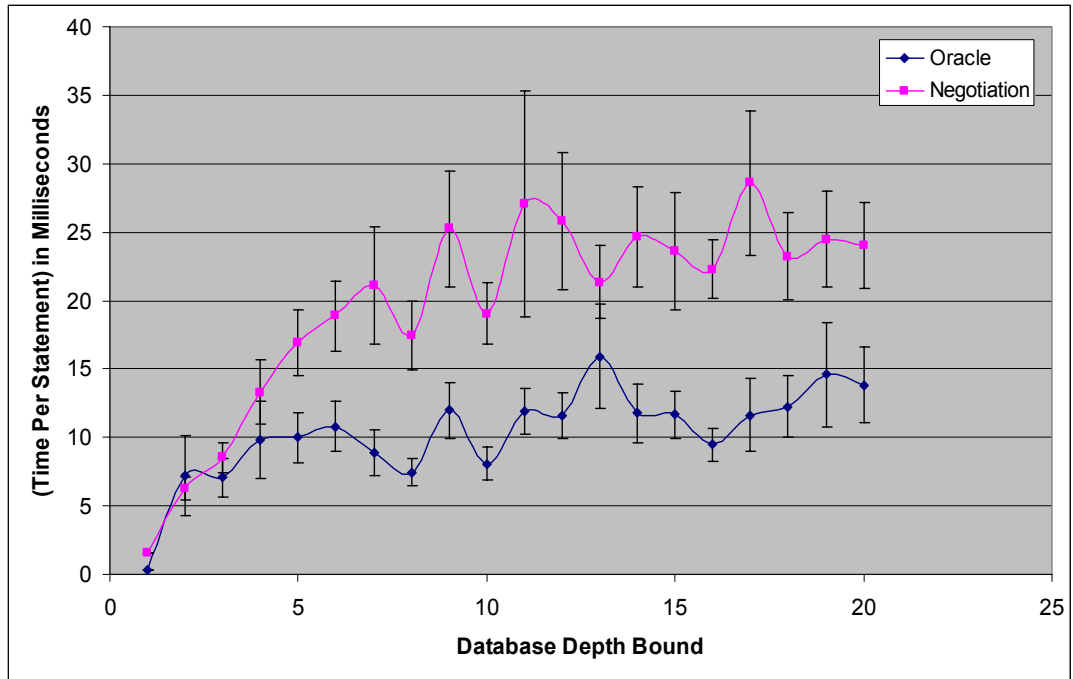


**Figure 65.**  $d_{max}$ : Processing Time



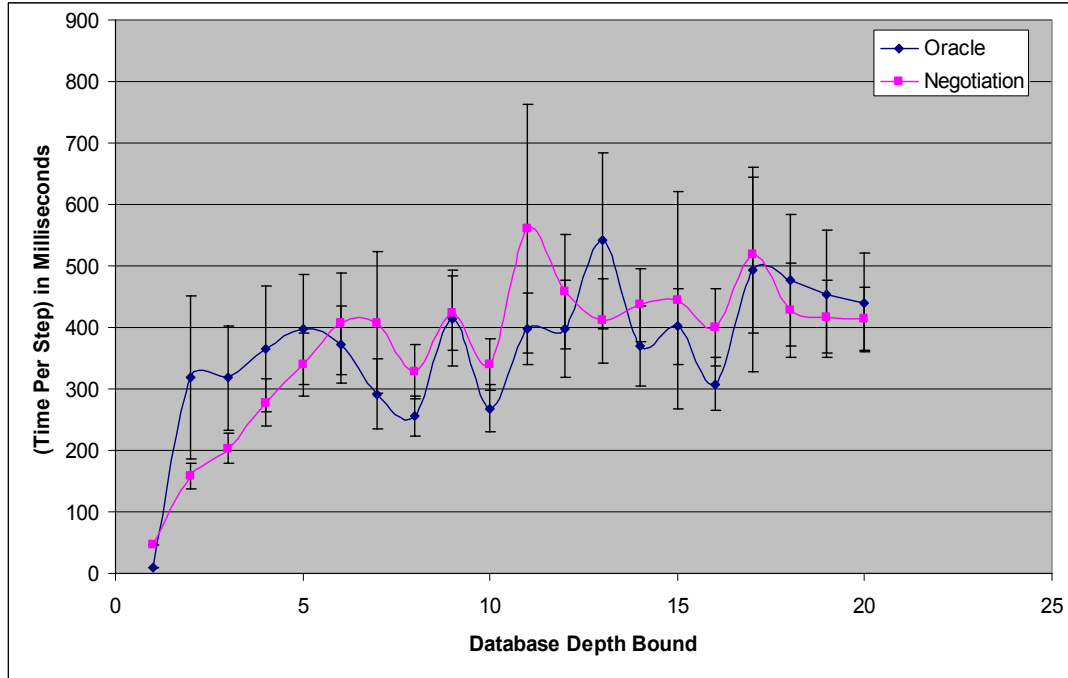
**Figure 66.**  $d_{max}$ : Ratio of Processing Times (Negotiation Time / Oracular Time)

We can draw the same conclusions that we did from the graphs representing the number of nodes examined (see Figures 62 and 63). The processing time for a test scenario increases linearly for lower depth values and remains roughly constant for higher depth values. This is probably because the parameter is an upper bound rather than a tight characteristic of the database pairs. As we increase  $d_{max}$ , a relatively low fraction of generated trees have truly high depth, resulting in the averages being almost constant. The depth parameter ( $d_{max}$ ) is therefore less meaningful in this respect as compared to the optimal negotiation length parameter ( $l_{min}$ ), with which the processing time has an exponential relation.

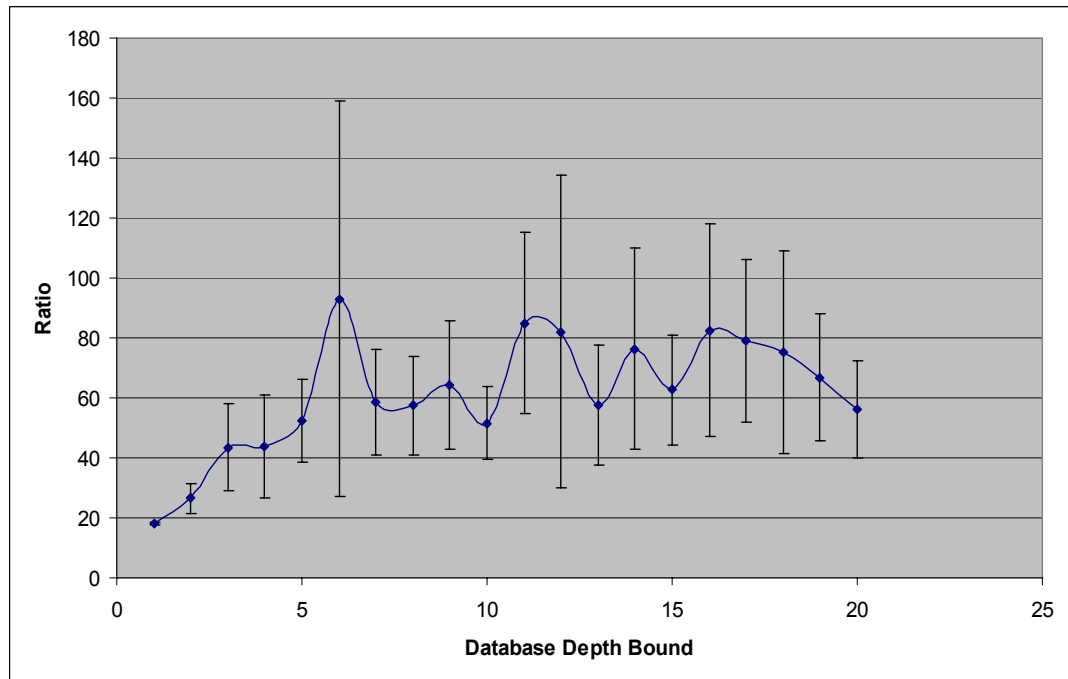


**Figure 67.  $d_{max}$ :** Total Processing Time per Unit Database (Processing Time / Number of Policy Statements in the Database)





**Figure 68.**  $d_{max}$ : Average Processing Time per Policy Resolution Step

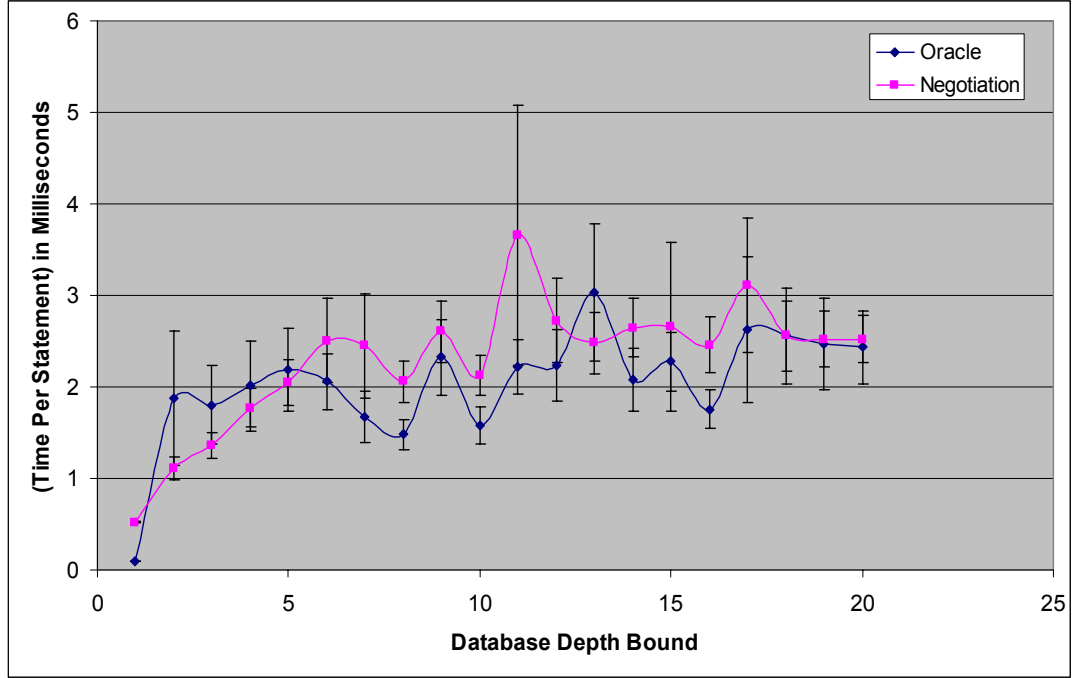


**Figure 69.**  $d_{max}$ : Ratio of Average Processing Times per Step (Negotiation Time / Oracular Time)

The effect of normalizing processing times with the database sizes has no meaningful effect, as the graph in Figure 67 indicates an increase in oracular and negotiation processing times in equal proportion (irrespective of the number of statements in the database.)

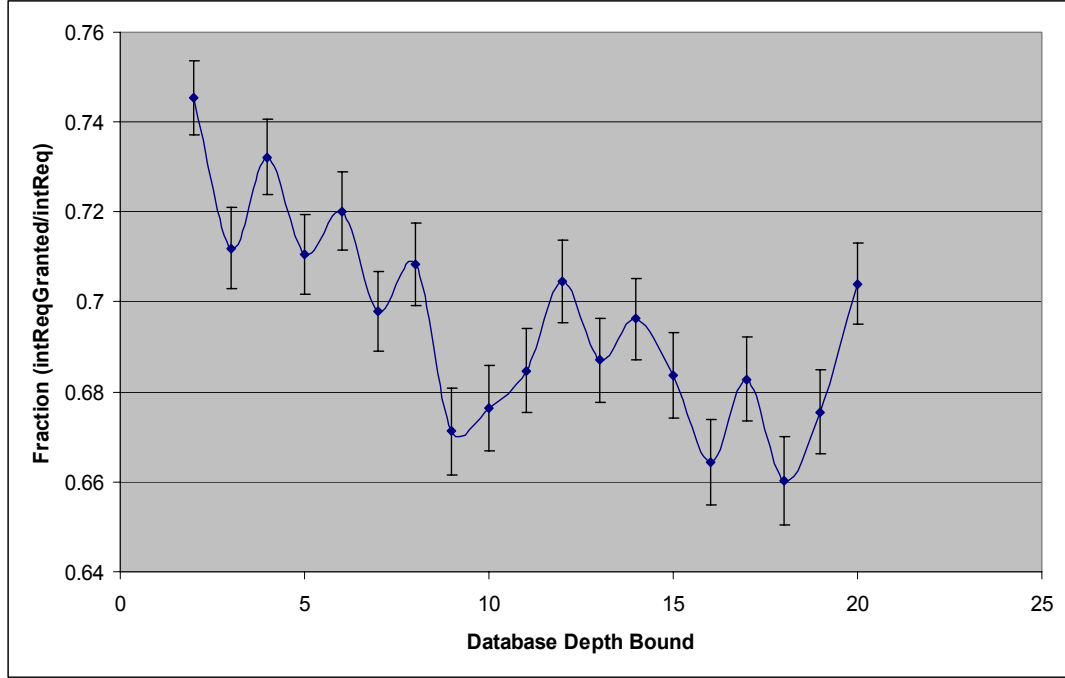
The average time taken per step presents a mixed picture of the relative performance of the negotiation protocol and the oracle. As we can see in the graph in Figure 68, the processing times per step are statistically identical, at least for higher depth values. At lower depth values, the increase is linear, and the time per step taken by the oracle is relatively higher than the corresponding time per step taken by the negotiation protocol. From a practical point of view, this implies that the efficiency of negotiation is higher than that of the oracle, and if we could find good heuristics (with low computation times) that result in a reduction in the number of negotiation steps, we would obtain significant performance savings.

As we observed in the case of all other metrics, the graph in Figure 70 indicates that the average oracular and negotiation processing times per step increase in equal proportion, irrespective of the number of statements in the database.



**Figure 70.**  $d_{max}$ : Average Processing Time per Policy Resolution Step / Database Size

*Number of Intermediate Requests:* The graph in Figure 71 shows how the fraction of intermediate requests granted ( $intReqGranted/intReq$ ) varies with  $d_{max}$ . We only considered those test cases for which a non-zero value of  $intReq$  was recorded. All quantities are reported with 99% confidence intervals.



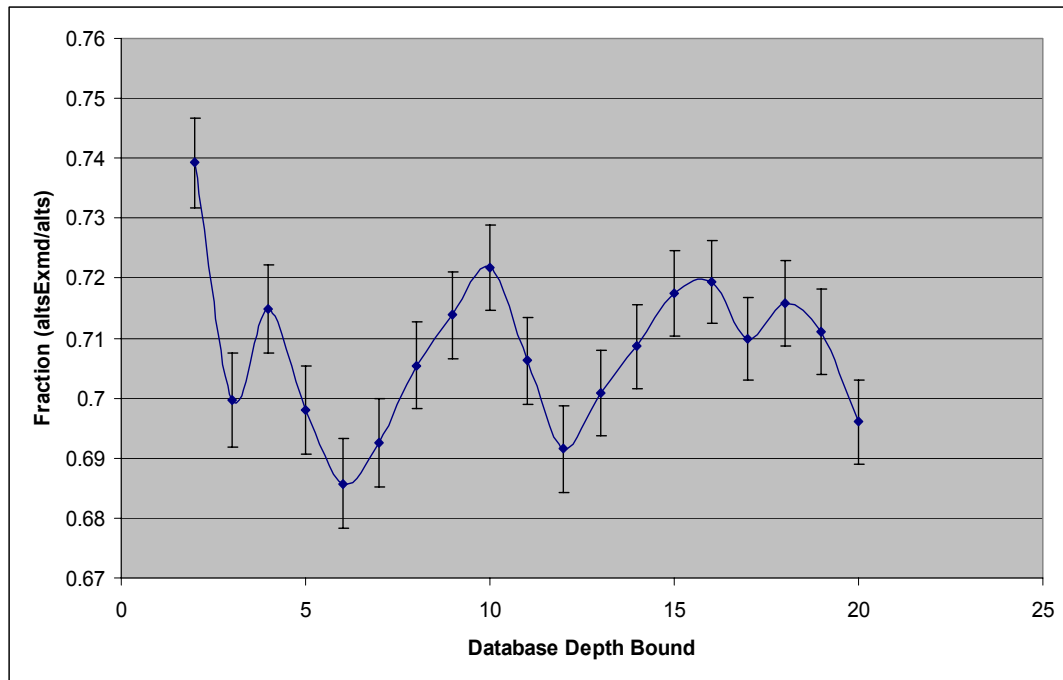
**Figure 71.  $d_{max}$ :** Average Fraction of Intermediate Requests Granted

Inferring a meaningful trend from this graph is difficult, since the fraction of intermediate requests granted varies within a range of 0.66-0.75, a 9% variation across a wide range of  $d_{max}$  values. We do observe that the fraction is highest at the least value of  $d_{max}$  (which is 2 in the above graph). This is because database pairs with low depths produce shorter negotiations on average. When shorter negotiations succeed, fewer intermediate requests are posed and a larger fraction of those is likely to result in an affirmative offer (for example: if  $intReq=1$  and the negotiation succeeds,  $intReqGranted/intReq$  cannot fall below 1; if  $intReq=5$ , the fraction could be anywhere in the range 0.2-1.)

The fraction of intermediate requests granted appears to decrease (albeit with oscillations) until  $d_{max} = \sim 10$  and then seems to be roughly constant within the margin of

error. To conclude, since this fraction represents information leakage (or degree of privacy), the fact that we can set an upper bound on it is an encouraging result.

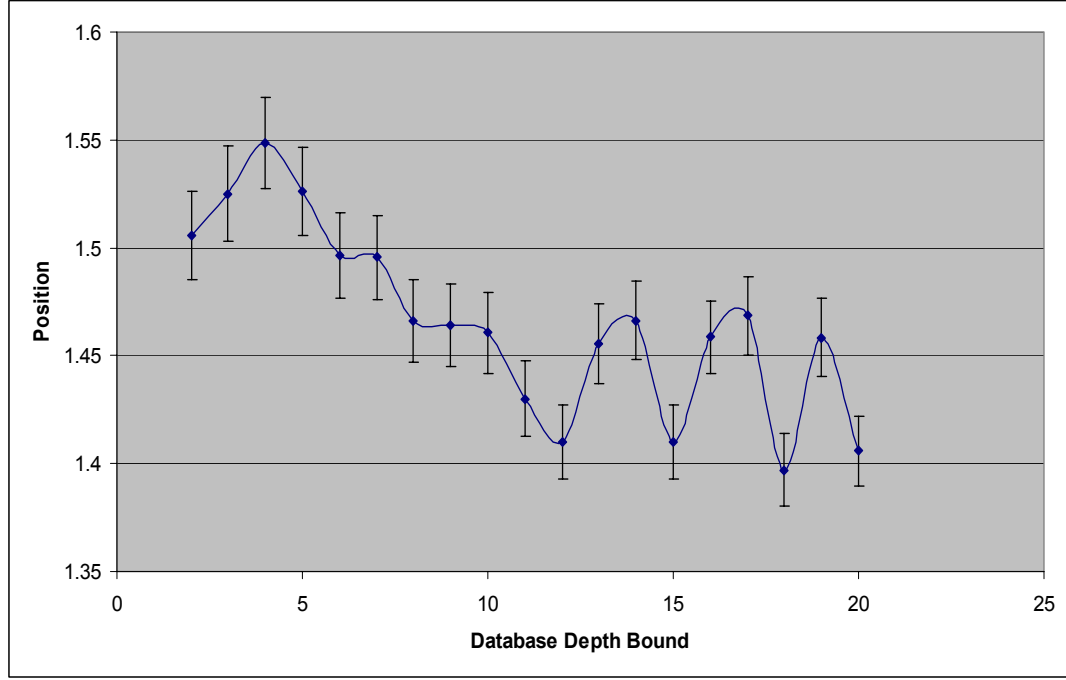
*Number of Alternatives:* The graph below shows how the fraction of alternatives examined ( $altsExamined/alts$ ) varies with  $d_{max}$ . We only considered those test cases for which a non-zero value of  $alts$  was recorded. All quantities are reported with 99% confidence intervals.



**Figure 72.  $d_{max}$ :** Average Fraction of Alternatives Examined

The fraction of alternatives examined is as difficult to characterize as the fraction of intermediate requests, as all the recorded values (see Figure 72) lie within a 0.6 range band (from 0.68 to 0.74.) Unsurprisingly, the fraction is highest (0.74) at the least value of  $d_{max}$  (equal to 2 in this graph), probably because a large percentage of results evaluated

to unity. On average, the fraction is approximately equal to 0.71 over the entire range of  $d_{max}$  values, which implies that not more than 7 out of 10 alternatives will be examined on average, irrespective of the depth bound.



**Figure 73.**  $d_{max}$ : Average Position of the First Valid Alternative

From the graph in Figure 73, we can observe a steady (roughly linear) decrease in the position where a valid alternative is hit for  $4 < d_{max} < 11$ . For  $d_{max} > 11$ , the results are less meaningful, though it is likely that the curve will asymptotically converge to a value close to 1. In the range  $d_{max} > 11$ , the position value appears to oscillate around the 1.45 mark. We can reasonably conclude from the above graph that the average position of a valid alternative does not increase with an increase in  $d_{max}$ . *Note:* These results are drawn only from scenarios that contain valid alternatives; there are a number of cases where no valid alternatives were found, and these cases are not represented in the above graph.

### 9.5.5.2 Conclusions

To summarize, the set of graphs and tables presented here support the following general conclusions:

- On average, the length (number of steps) of an actual negotiation increases linearly with the length of an optimal negotiation for the same test scenario.
- Failed negotiations (that are unable to satisfy the initial goal) are efficient on average, the length being under 4 steps (optimal length = 3 steps).
- Total processing times for negotiations tend to dominate total oracular processing times. But the average processing time per step for negotiation tends to be dominated by the average processing time per step for the oracle. The average processing time per step (or even per tree node) is lower on average for the negotiation protocol.
- The fraction of intermediate requests granted and the fraction of alternatives examined show meaningful (albeit low) variations with all four of our parameters.
- An increase in the branching factor bound ( $b_{max}$ ) produces a significant increase in processing costs (nodes examined and processing times). For realistic values ( $b_{max} \leq 6$ ) though, the performance does not exceed the tolerable range.
- An increase in the depth bound ( $d_{max}$ ) results in an observable change to our metric values only for lower values of  $d_{max}$  (typically less than or equal to 10). The values of the performance metrics (number of steps, nodes examined, and processing times) in this range increase at a slow rate, and almost seem to be constant for values of  $d_{max}$  greater than 10. Thus, the depth bound does not affect performance in a significant way, and is not a bottleneck in the way the branching factor bound appears to be.

## Chapter 10

### Related and Complementary Research

There is a significant body of research in the design of negotiation protocols, policy languages and ontology, service discovery and policy management in open environments, and distributed trust and access control frameworks. We discuss each of these in this chapter and either differentiate those contributions from ours or show how their results could complement ours in an effort to advance the state of ubicomp interoperation research.

#### 10.1. Negotiation Protocols

##### 10.1.1 Automated Trust Negotiation

The work that bears the closest resemblance to my research is *automated trust negotiation* [Winslett2003], through which web entities (typically client-server, but the model applies to peer communication as well) can establish trust in an automated fashion, the objective being access to a guarded resource. It is a flexible way of doing access control, where entities can control what private information is released at fine granularities, though it is only a special case of general resource negotiation. Negotiators request and expose sensitive credentials based on evaluation of per-credential access control policy rules. An earlier protocol, TrustBuilder [Winslett2002], implements trust negotiation as an extension of TLS. The policy language is not very expressive, being



limited to Boolean expressions of credentials. This has since been superseded by PeerTrust [Gavriloaie2004; Nejd12004], whose policy rules are written in a distributed logic programming language based on Prolog, and which targets semantic web applications. PeerTrust policy rules can be signed (equivalent to certificates) and their evaluation delegated to another peer, thereby enabling distributed and cooperative security.

Automated trust negotiation does not solve the more general problem dealing with simultaneous discovery and negotiation of multiple resources. Also, in ubicomp, trust is not restricted to the presence or absence of credentials but can have a much broader definition where risks can be analyzed using any subjective measure. Both PeerTrust and TrustBuilder suffer from drawbacks such as the lack of multiple goal support, lack of context-awareness, rigid policies and trust building restricted to exposure of cryptographic credentials. Both these systems fail to consider inter-dependence of multiple resource and security constraints, because they target explicit and addressable connections on the Internet rather than ad hoc ubiquitous connections. Still, our research has benefited to some extent from earlier work in trust negotiation, prominently with respect to policy language requirements [Seamons2002] and negotiation strategies based on policy and trust information.

Protune (Provisional Trust Negotiation) [Bonatti2005; Coi2007] is the closest working system to our own that we can find. It builds on PeerTrust, with declarative logic program-based policy rules written as Horn Clauses. The expressivity of their policy language as well as the negotiation dynamics are similar to those of our system, with

rules being scanned in order to determine counter-requests [Bonatti2005]. Protune uses metapolicies to filter policies whereas our protocol uses designated predicates to achieve the same effect. Both requests for credentials and requests to perform actions can be posed by Protune negotiators to each other.

In spite of the similarities, there are still salient differences between our system and Protune. First, an entire policy rule (though filtered) is sent in a request message in Protune, thereby placing the burden of interpreting the policy and satisfying it to the receiver. In our protocol, multiple alternative sets are generated and tried one by one. Bonatti et. al. claim that their approach is more efficient and that generating multiple alternatives will lead to exponential blowup, but our performance results (for realistic cases; see Chapter 9) do not seem to bear out their claim. Also, since neither negotiator knows about the policy facts or rules of the other, there is no guarantee that a Protune receiver will return a reply that the sender is satisfied with; in our protocol, requests are more specific and provide better guarantees of agreement. Our policy and negotiation framework is also more general than Protune. Negotiators using our protocol can trade a variety of possessions and data, and also communicate associated objects as attachments, compared to just credential objects and policy rules in Protune. Our negotiation protocol is built to handle multiple goals, goal revisions during negotiation, and determination of compromises using our alternate offer generation procedure. Protune does not provide these features yet, though it could conceivably be extended to support these functions. We also have extensive practical test results comparing centralized and distributed policy

resolution, whereas the designers of Protune have not conducted such tests (to the best of our knowledge).

### **10.1.2 Automated Negotiation for Services**

The growth of web services technology, grid computing [GRID], Web 2.0 and the semantic web has prompted significant research in enabling agents to collaborate and negotiate for the privilege of obtaining services. A variety of standards have evolved around negotiations for web services and in generating Service Level Agreements (SLAs). Service-oriented architecture, enabling interoperation of services for business applications, has also inspired research into automated negotiation. We describe negotiation research in these areas here and differentiate them with our research results.

Negotiation protocols developed for the grid typically involve matching the preferences of service owners and consumers, who often lie in different administrative domains. Negotiation steps consist of proposals and counter-proposals, which are evaluated against resource utility functions (specified in terms of maximum and minimum values) [Lawley2003]. Agreements are reached when no higher-utility counter proposal can be generated and basic constraints are met, otherwise the result is failure. Negotiation strategies could include heuristics such as the number of messaging rounds and could be guided by game-theoretic algorithms like Faratin's [Lawley2003], or by the use of genetic and evolutionary algorithms [Chao2002]. SNAP [Czajkowski2002] generates SLAs through a distributed resource allocation protocol, and is agnostic of the resource or application type. Policies are private, and clients make requests and counter-

requests for a *level* of a service till the server is able to satisfy or reject the request. Grid protocols offer valuable guides as far as negotiation heuristics and dynamic agreements are concerned, but they do not consider security constraints and service discovery. The grid also does not require expressive policy languages or reasoning mechanisms, since only the level of a single resource is being negotiated. Last, these protocols are employed in static situations, and cannot be used in scenarios that involve mobility and context changes.

More work has been done in developing standards like WS-Policy [WS\_Policy] and WS-Agreement [Andrieux2007], oriented towards enabling universal negotiation in the web services arena. Being based on the widely adopted SOAP and XML standards, interoperability is largely guaranteed. Other frameworks independent of WS-Agreement have also been proposed [Paurobally2007; Tsai2007]. But the scope of the negotiations is largely restricted to SLAs and auction-type applications, both cases involving known negotiators, known services and not much effort made to infer semantic information from policies in order to engineer a flexible protocol that will explore all ways to reach an agreement. Efforts have been directed towards securing legally binding contracts through negotiation [Parkin2006] or involving multiple resource requesters in order to ensure that needy consumers get a maximal level of service [Abdoessalam2004]. Efforts at advancing the WS-Agreement standard have focused on meta-protocols that help select a suitable negotiation type for a given instance (this type could be a one-to-one bargain, an auction, etc.) [Hudert2006] or act as a broker between a web service and consumers in a service-oriented architecture [Ardagna2004]. These efforts largely ignore the actual

negotiations themselves, which are deemed to be closely tied to the applications. Our negotiation protocol provides more richness and flexibility without sacrificing generality through the use of declarative logical policies. Therefore, even though WS-standards offer significant advantages, we did not consider them a suitable basis for our research.

Similarly, Dang and Huhns [Dang2006] propose a protocol to manage multiple concurrent negotiations for services among service providers and consumers. Negotiation agreements are based on utility functions that must be reconciled, whereas our framework is based on security and resource usage policies. Andreoli and Castellani [Andreoli2001] outline a negotiation protocol that views negotiation as a distributed proof tree, but use a centralized coordinator, which is fundamentally different from our approach.

Interest-based negotiation (IBN) [Rahwan2005; Pasquier2007] enables goal achievement through collaboration, and is both a more general alternative to game-theoretic and heuristic negotiation. Each negotiator generates a set of alternative plans and poses queries and sub-goals in turn. Every proposal is accepted or rejected on the basis of a negotiator's interest (utility) in getting its goals satisfied. This protocol is more similar to ours than grid negotiation protocols. Yet there are salient differences. The procedure is not modeled as a distributed policy resolution, and its theoretical correctness and completeness properties have not been studied as ours have. Also, this model assumes negotiation purely for collaboration, and trust and access control issues are not considered; our protocol maintains more anonymity and privacy of policies. We have studied the role of policies more comprehensively, and also reported results from optimality testing, which the IBN researchers have not.

### **10.1.3 Speech Act-Inspired Negotiation Protocols**

The use of speech acts as a way of expressing the language of a negotiation is not unique to our system [Jiang2005a; Xu2005]. Chang and Woo designed and implemented a speech-act based negotiation protocol (SANP) [Chang1994] based on Ballmer's and Brennenstuhl's speech act classification [Ballmer1981], whereas we use Searle's classification [Searle1981]; this enables users of our protocol to query information from each other, something that SANP does not do. Also, their protocol is suitable only for single-issue negotiations, whereas ours generates and offers negotiators multiple alternatives. Their protocol also does not leverage user policy in any way, nor does it emphasize security.

## **10.2. Policy Languages and Models**

Before going into comparisons with existing policy models and languages, we will briefly describe the Semantic Web [SemWeb]. The Semantic Web is work in progress toward a common framework that allows data to be shared and reused across application, enterprise, and community boundaries (see Chapter 1: Figure 1). The Resource Discovery Format [RDF], which is the semantic data description model that uses XML syntax and URI naming procedures in order to allow applications to interoperate, is one notable standard to have emerged from this effort. Though the complete scope of the semantic web is not very clear, some researchers consider the definition of ubiquitous computing ontology and security policies to be within its ambit [Kagal2003b]. Therefore, most policy models and languages that target agent communication on the Semantic Web are

equally applicable to ubiquitous computing. Local interactions are more common in the latter, but the issues related to policies can be formulated in similar ways. Therefore, in this section we will discuss related work in policy languages primarily targeted toward the Semantic Web.

### **10.2.1 Policy Languages**

Many research efforts have investigated the use of policies to specify goals, and control systems and security, because of their flexibility and adaptability [Sloman2002]. Such research has primarily focused on policy specification and expressivity as compared to the interoperation mechanism, but we have borrowed concepts from several policy languages. A number of languages have adopted a logic-based approach, like Rei [Kagal2003a], ASL [Jajodia1997], and others like KAoS [Uszok2004] are based on the OWL Semantic Web ontology. Among these, Rei [Kagal2003a; Kagal2003b] is targeted primarily toward pervasive computing and provided significant inspiration for our policy language. The designers of Rei correctly make the claim that a policy language for ubicomp must have well-defined logical semantics because the scope of facts and constraints that it must support is huge, and that it needs to be domain-independent. Rei treat rules as being part of a system, rather than independent policies. This enables inference of dependencies and conflicts among policies. The language is based on first-order semantics augmented by deontic concepts of obligations, permissions, prohibitions and dispensations. It supports specification of actions, action classes, speech acts like requests, offers, delegations and revocations, as well as meta-policies like modality (e.g.,

rule A overrules rule B) or priority for conflict resolution. In addition, we can describe common resources, entities and constraints, as can be done with most other languages. Our language goes further by providing support for inter-resource constraints, contextual description, and high- to low-level policy translation. These are some of the requirements for trust negotiation languages [Seamons2002] that are requirements for our language as well (see Chapter 4).

Portfolio and Service Protection Language (PSPL) [Bonatti2000] and DTPL [Herzberg2000] are languages designed with these requirements in mind; the former was designed expressly for trust negotiation. *Services* offered by servers are distinguished from information possessed by clients, like cryptographic credentials and plain-text data declarations, collectively called a *portfolio*. The language offers support for description of service classes, subset and domination relationships (which can be used to specify high- and low-level rules and their relation), state information (both persistent and per-negotiation). Evaluation of requests, release criteria for private resources, and policy filtering are some of the other essential functions that are provided. The syntax and rule evaluation procedure are Prolog-like, and so some constructs that aren't directly supported, such as more complex trust relationships, delegated credentials and trust chains, could be added without significant difficulty. Like other non-semantic languages, rules are associated with resources and so inter-resource relationships and complex security relationships cannot be handled. Specification of general policies that can be adjusted with context, deontic concepts and meta-policies are not supported.



Ponder [Damianou2001], a declarative policy language for specification of a distributed system security policy, deserves mention as an early piece of research in ubicomp policy. One can specify delegation policies, entity roles, application groups, constraints and obligations in Ponder. It supports conflict resolution using meta-policies and policy enforcement through the triggering of foreign functions. On the negative side, it suffers from the same drawbacks as Rei, and in addition is an object-oriented language rather than a semantic language. Keynote [Blaze1999] deserves mention too as one of the earliest policy languages used for trust management and access control in an open system. The language allows specification of credential types and instances, associated actions and state constraints, which could be used by a compliance checker when validating an object access request. It does not support context-awareness, meta-policies or deontic concepts and actions. It is too closely tied to entity names and credentials, and has been superseded by languages that are more suitable for ubicomp, though the concepts proposed in Keynote still remain relevant.

As a side note, we must mention Datalog, a declarative logic programming language, which has been used to manage database constraints. But even though its logical semantics and reasonable efficiency make it an attractive candidate for a policy language, its syntax is more restrictive than Prolog, posing obstacles to the kind of expressivity that the latter provides. None of the prominent languages like Rei or PSPL, and negotiation frameworks like Protune, are based on Datalog, and that influenced our decision to use Prolog as a basis for our language. The more widespread availability of tools and APIs for Prolog also make it a more attractive base compared to Datalog.

Various XML-policy languages have been designed for access control on the web, such as IBM's Trust Policy language [Herzberg2000], X-Sec [Bertino2001] and XACML [Lorch2003]. The former two support description of credentials and credential types, role types (in TPL), entities and attributes. Policies are used by servers to infer trust information through a more complex procedure than Keynote, though support for deontic concepts and different types of actions are missing. XACML supports more expressiveness in terms of groups of resources, protocols, actions and authentication mechanisms, and allows specification of conflict resolution rules. But it is not as suitable for our aims as the other two languages, because it lacks a logical reasoning back-end. Last, we will elaborate on the WS-Policy standard [WS\_Policy] mentioned in Section 10.1. WS-Policy is based on XML and specifies a general wrapper syntax, leaving details of policies to individual domains and applications. Though a widely accepted standard that provides support for specification of conjunctions and alternative constraints, WS-Policy is completely agnostic of enforcement mechanisms. This makes it less suitable for a language designed to support negotiation. We feel that policy-writing in Prolog syntax is more intuitive and less cumbersome than in WS-Policy. WS-Policy also does not specify ways to declare event-condition action rules.

P3P (Platform for Privacy Preferences) [P3P] is a proposed web user privacy standard rather than a language, but we reference it as an important contribution to web collaboration research. It provides an XML-based language that allows both websites and clients to describe their privacy policies, the goal being a mutual agreement. Feedback on policy conflicts is given to the user, who can then make an information release decision.

Much of the research in advancing P3P has focused on policy matching and providing feedback to users, rather than on automated negotiation. Currently, the standard suffers from many limitations, such as the lack of a model for website trust and verification of policies, and an expressive policy language, leading to low adoption [Kolari2005; Kagal2003b]. Though enhancements have been proposed, such as using the more expressive Rei language [Kolari2005], trust management using reputation frameworks, and allowing users to set per-website and data item preferences, P3P leaves too much to users and depends on rigid adherence to policies, requiring significant additional research to make it suitable for ubicomp negotiation.

As we can see, the above languages adopt a wide variety of approaches. In our research, we have taken the approach of building bottom-up from a language with loose semantics and a reasoning mechanism (like Prolog) rather than making syntactic choices (like XML) and adding new constructs and logical reasoning mechanisms. Though this may not be the only valid approach, we have shown that it can work.

### **10.2.2 Ontology**

As described in Chapter 4, ontology is an important part of a policy language [Leithead2004], which enables the language and negotiation methodology to be applicable across domains. We have studied semantic web ontologies like DAML+OIL (an extension to XML and RDF) [DAML], OWL [OWL], FOAF [Dumbill2002] and SOUPA [Chen2004]. DAML-Space and DAML-Time are ontologies for specification of

spatial and temporal characteristics, respectively. Among these, our research drew most inspiration from SOUPA, which was proposed by the designers of Rei.

### 10.2.3 Event-Triggered Condition-Action Policy Support

Most policy researchers classify security policy types in two ways: i) authorization policies, and ii) obligation (or event-triggered condition-action policies) of the form “*on event if condition do action*” [Sloman2002]. Neither our policy classification nor that of others [Kaminsky2005] is limited to security policies, and consists of goal and invariant policies that dictate resource needs and system-wide constraints that may not have security implications. Among these, only obligation (or ECA: Event-Condition-Action) policies explicitly mandate actions on the part of a policy manager. Consequently, obligation policy rules are written in a distinct way and occupy a distinct subset of the overall policy language. Not all policy languages support ECA rules; for example, Protune [Bonatti2005]. Some languages, like Lucent’s Policy Definition Language [Lobo1999], only support ECA rules, and others (like ours) establish a special syntactic construct for such rules. The framework underlying enforcement of such rules largely follows a standard template. The policy manager continuously monitors for events; if one occurs, matching policies are looked up and the specified actions executed [Lupu1999]. Most ECA policy language work has focused on detecting and resolving conflicts among the intended actions [Shankar2005a; Shankar2005b; Cholvy1997], which is not something we focus on in our research although we would like to add this in the future. Application-wise, ECA policy languages have been developed for ubicomp

[Shankar2005a; Shankar2005b], the Semantic Web [Papamarkos2003; Papamarkos2004], and IT infrastructures [Agrawal2007]. Syntactically, Shankar et al. [Shankar2005a] use an imperative-programming syntax (explicit *if-then* rules) to encode policies and Papamarkos et al. use RDF/XML. We differ from both approaches by specifying ECA (or *update* rules) in Prolog, though operating system calls and Java method invocations may result.

### **10.3. Middleware**

We have described the goal of our project as the achievement of spontaneous interoperation among ubicomp domains for the purpose of service and resource access. Our solution takes the form of a middleware that supports interoperation through negotiation and also manages and enforces per-domain policy. In this section, we cite relevant research in these areas, and show how their approaches differ from ours.

#### **10.3.1 Smart Spaces and Ubiquitous Interoperation**

We briefly survey the different approaches taken by ubiquitous computing “smart space” projects to handling interoperation in an automated manner, and how these and other open systems handle the core problems of service discovery, resource management and access control within and across domain boundaries.

The most prominent smart space projects typically look at different components of an active space (computing entities, resources, physical interfaces) as parts of a whole, rather than independent entities in their own right. Therefore, the typical mode of

management is centralized, and works well, but does not scale. Metaglué [Coen1999] is an extension of Java that provides the computational glue for interoperation of software agents in an Intelligent Room [Adjie-Winoto1999; Brooks1997], which is a product of MIT's Oxygen [MIT-Oxygen] project. Gaia [Román2002] and One.world [Grimm2004a; Grimm2004b] are the equivalent of operating systems for pervasive computing systems, which manage resources, devices and applications within a domain. What these systems do very well is to facilitate ad hoc interactions in a seamless manner, manage resource (or agent) discovery and allocation in a dynamic manner, and adapt to a limited range of context changes. The price that needs to be paid for such seamless operation is standardization of hardware and software components and application designs. Interoperation within a room is limited to familiar devices that know what to expect and have ways of obtaining those resources. They also do not consider the management of multiple active spaces. One.world, as an example, provides flexibility through an application-oriented approach that assumes the trustworthiness of devices, in contrast to our device-centric approach.

In these systems, security and privacy have been afterthoughts, and even the augmented frameworks do not provide good solutions to handle unknown and un-trusted devices. Metaglué has been superseded by Hyperglue, designed to enable interactions among multiple active spaces and avoid centralized management [Kottahachchi2004; Peters2003]. Hyperglue has the nice property of letting domains manage themselves independently and interact with others as a single virtual entity. A context-aware role-based access control scheme is used to grant permissions, but the assignment of roles is

limited to known entities or entities that can prove transitive relationships. With a limited trust model and the lack of a flexible negotiation scheme, the interoperation features provided fall short of ubicomp requirements. Gaia is one smart space project that has dealt more than most with security and privacy issues [Campbell2002]. Cerberus [Al-Muhtadi2003], a Gaia security extension, uses policies and *confidence levels* in authentication schemes to enforce access control during interoperation in a context-aware and non-intrusive manner. Like Hyperglue, Cerberus uses security policies for unidirectional access validation and does not support dynamic service discovery and negotiation. Mobile Gaia [Chetan2004] extends the basic Gaia design to manage ad hoc clusters, like Panoply spheres, but the negotiation is limited to the production of a familiar public key that can be authenticated by a cluster. Gaia Super Spaces [Al-Muhtadi2004] handles multiple active domains in a semi-centralized manner, interoperations occurring through *bridges*. Super Spaces provides service lookup and access functions across domains, though without considering the security aspects. Centaurus 2 [Undercoffer2003] and the Aware Home project [Kidd1999] are yet more examples of ubiquitous active spaces systems, though these place a high priority on security. The results are not very different from Hyperglue or Cerberus, however. Centaurus 2 enables secure interaction within a hierarchy of spaces through access capabilities, and an Aware Home uses the powerful GRBAC model for access control. Still, the security aspect is independent of the service lookup and management module, and security policies are enforced in a way so that only known entities with predeployed trust information may obtain access permissions for services.

### 10.3.2 Service Discovery and Access

As we described in the introductory chapter, discovery of services and obtaining access to them is the goal of interoperation. In ubiquitous computing environments, these goals face several challenges, such as device integration with environments, compatibility, naming issues, heterogeneity, security and privacy [Zhu2005b]. Over the past few years, many approaches have been proposed to handle one or more of these challenges. Though none of these approaches completely address service discovery in unfamiliar environments while ensuring some form of security, we survey the prominent ones below.

*Jini* [Waldo1999], a Java-based technology, enables autonomous service discovery and resource access over a network connection. Devices can register, discover services and access them through proxies, lookup tables and leasing mechanisms. Standard interfaces and mobile code enable spontaneous interoperation, since every device in a Jini-enabled space communicates using Java RMI. Jini is easier to use and maintain than similar frameworks, like CORBA or DCOM, where protocol changes must be synchronized among servers and clients offline. The primary goal of service discovery and access is handled well, but the use of open interfaces is unacceptable where there are even minimal security and privacy concerns. The model by itself, even with authentication and authorization mechanisms, works in a static domain that serves a set of known client devices and does not adapt to context change. Dynamic policy-guided negotiation expands the scope of the interoperation problem and solves complementary issues while retaining the useful features of Jini. *Universal Plug and Play* [UPnP] is



another communication architecture based on well-known technologies like TCP/IP, HTML, and XML. UPnP allows seamless, spontaneous networking among suitably configured devices irrespective of hardware or operating system characteristics. Devices can advertise their capabilities and learn about other devices' capabilities through SSDP, GENA and SOAP protocols. On the downside, security is handled by using ACLs and certificates which the users are expected to maintain in a static manner; this is non-scalable and intrusive, and therefore not suitable for ubicomp. Zhang et al. [Zhang2007] have proposed an impromptu service discovery and access protocol that is based on "Semantic Spaces" which provides Panoply sphere-like services and also acts as a service portal that interacts with mobile devices. Wide adoption is not a problem as it is based on OSGi [OSGI] standards (which include UPnP), but the complete lack of security and privacy is. Recently, other service discovery protocols have been proposed that are based on Semantic Web ontologies like OWL; these protocols attempt to match consumer and service provider preferences, focusing on context-sensitivity [Broens2004] or performance [Chakraborty2006]. But both these protocols assume the environment to be trustworthy with cooperative agents, and neither is concerned about security or privacy.

Zhu et al. [Zhu2005a] do consider security in their service discovery protocol for pervasive computing. In the absence of a trusted third party, a service provider and a client expose partial sensitive information in a progressive approach till both parties reach an agreement about exposure of the nature of the service and authentication information. Upon a mismatch or an unsatisfied request, the protocol can be terminated without loss of privacy. The key drawback, from my point of view, is that the entities are assumed to

share security information, and the protocol is essentially a way of preventing malicious devices from causing privacy violations. These constraints cannot be assumed in the more general negotiation problem that I am addressing.

### **10.3.3 Policy Management Frameworks**

Policy management modules have been used in a variety of operating systems, database management systems, and security frameworks. Recent pervasive computing research has inspired the development of policy management frameworks as distinct subsystems for reasons similar to ours: heterogeneity and context-changes. These systems typically enable self-management through the use of obligation (ECA) policies [Kumar2007]. UIUC's Gaia middleware provides such a policy manager that maintains ECA policies. Their research has contributed towards both static and dynamic detection of action conflicts by annotating policies with post-conditions [Shankar2005a] and with pre-conditions [Shankar2005b]. Jiang et al. have designed a CORBA-based security management middleware for ubiquitous computing [Jiang2005b] that supports authorization, delegation and obligation policies. These policy management frameworks make useful contributions but none of them provide support for negotiation or dynamic access control.

## **10.4. Access Control and Trust in Distributed Systems**

Access control in Panoply (and, potentially, other ubicomp frameworks) is achieved through event filtering followed by negotiation. Because this procedure is driven by the

current state of the policy database, access decisions are context-aware. And because the policies can express constraints in terms of classes of entities, objects and properties, performing access control rule updates is a scalable procedure. Access control through negotiation is also peer-to-peer and completely distributed; the burden of proving or verifying an access right (or a set of access rights) does not fall on one party or the other. Negotiation also achieves distributed trust in some ways, by enabling mutually unknown domains to reach an agreement from scratch. Below, we survey both traditional and recent approaches to access control and trust-building in distributed systems, show how these approaches fall short of the ideal, and differentiate our approach from them. Also, individual domains will frame access control and trust rules based on their unique concerns; guiding the framing of more effective policies is complementary research.

#### **10.4.1 Access Control Models**

ACLs (access control lists) and capabilities are basic mechanisms for enforcing access control when familiar entities access known objects, and still retain their use in limited domains where one can set per-entity and per-object policy. Policy expressivity is very limited and rules are rigidly enforced. These mechanisms suffer from scalability issues when applied directly to ubicomp scenarios, and also don't adjust to changing context. They can be used in individual domains in a ubiquitous computing environment, but each of these domains must have a security and trust management framework in addition. Frameworks like Kerberos enforce access control through secure protocols, though they also do not scale beyond a small self-contained domain.

Most open systems, and those that need to handle access control for a large number of entities, typically use RBAC (role-based access control) [Ferraiolo1995] models or its variations. Entities are assigned *roles*, which are associated with sets of access privileges, in the most basic model. Role-based access control frameworks have better scaling properties than ACLs. But as roles have to be defined in advance, it is difficult to provide optimal security in dynamic situations where policies need to be adjusted with context.

Generalized RBAC [Covington2000] enhances RBAC by adding roles for accessible objects and environment states (context), and has been used in the Aware Home ubiquitous computing project [Kidd1999]. Because it increases the expressivity and intuitiveness of policy writing, it is an excellent choice for modeling policy languages, but offers no suggestions for enforcement (such as automated negotiation) per se. The distributed RBAC [Freudenthal2002] model is an augmented variation of RBAC, which uses *delegation* and distributed proof building of permission rights using knowledge of possessed credentials. Delegation of permissions is an extremely useful concept, and has impressive scaling properties, though it has trust issues and must be augmented with something like our negotiation model before it gains widespread adoption. UbiComp middleware like Centaurus [Kagal2001a] and Vigil [Kagal2001b; Kagal2002] have used role-based access control with delegation and trust chains. We have leveraged concepts proposed by GRBAC (in our policy language design) and DRBAC (our voucher data structure/credential supports delegation) in our policy management framework.

Benefit and Risk Access Control [Zhang2006] shows how an individual domain could incorporate perceptions of benefits and risks of sharing information into a graph data structure. Access control decisions are made by examining the graph to determine whether total benefit outweighs total risk. The structure can be adjusted at runtime, albeit with manual intervention. This is complementary research, and our negotiation protocol could incorporate BARAC data structures to build better heuristics in the controller module. The model is still deficient in some ways: i) it does not easily lend itself to automation, and ii) benefit and risk values are subjective decisions made by users rather than being automatically extracted from user needs and security concerns.

Two key drawbacks of pure RBAC when used in open systems are lack of i) context-awareness, and ii) support for decentralized interaction among autonomous peers. Recently, attempts to address these have been made. *Law governed interaction* [Minsky2000] has the view, which we share, that role semantics should be dictated by context and policy, the latter being independent of particular role definitions. This helps to rectify drawbacks inherent in RBAC, such as difficulty in specification of exception conditions, and the potential security holes that could result. The key drawback with LGI is that it assumes a common *law* or policy governing all interacting domains, which is impractical, as I have argued earlier when discussing the need for negotiation. The access control model proposed by Bohrer et. al. [Bohrer2003] is a superset of the LGI model, as it incorporates individualized privacy policies in addition to common enterprise-wide policies. Another access control model that specifically targets pervasive computing environments [Hengartner2003; Hengartner2005] uses information relationships and per-

domain policy specifications to ensure that entities release minimum sensitive information. If multiple entities are involved in the transfer of such information, each entity gets information on a need-to-know basis. Kapadia et al. [Kapadia2007] use the metaphor of “virtual walls” to ensure a similar form of access control. They enforce three levels of access: *transparency*, *translucency* and *opacity*. We view event flow in Panoply the same way that Hengartner et al. view information flow, and enable access control through event filtering. However, our policy manager also explores alternatives upon failure, through negotiation.

Making access control more context-aware [Kulkarni2008] and distributed [Cautis2007] has been a research focus ever since the advent of pervasive computing. Toninelli et al. [Toninelli2006] propose a form of access control whereby policies are suitably adapted to context. Like our system, they use a semantic logical language to encode policies. Sampemane et al. [Sampemane2002] also demonstrate the use of policies and a role-based access control model in Gaia. Our policy management framework augments these systems by adding a negotiation protocol.

Minami and Kotz have designed and implemented a framework for secure context sensitive authorization [Minami2005; Minami2006] that models access control as a distributed proof tree spanning multiple entities. Their framework is similar to ours in many ways. Peers in their system formulate policies in a logical manner using Prolog, and frame meta-policies for integrity and confidentiality. In their scheme, when certain predicates in the body of a policy rule cannot be evaluated locally, such evaluation is delegated to a different entity that has policy rules or facts corresponding to that query.

Based on confidentiality policies, an entity could choose to either return an answer to a query or send a rejection message. The main difference with our framework is that their system provides no scope for negotiation, and also that they assume the integrity and confidentiality policies to be completely public, which facilitates selection of suitable hosts for building the distributed proof tree. We have gained valuable insights from Minami and Kotz's research, both conceptually and from the point of view of performance evaluation of our framework [Minami2006]. In reference to distributed proof trees, the PeerTrust researchers model automated trust negotiation in the form of a distributed proof tree [Alves2006], but this takes the form of a delegation tree more than a tree consisting of false starts and alternatives like ours; also, PeerTrust handles mainly credential requests and delegations, as opposed to our more general negotiation framework.

#### **10.4.2 Trust Models**

Trust has been studied in recent years, both in the context of theoretical models and for practical use in open systems. It has always been used in limited ways in computer security, based either on identity or trusted authorities. It would be fair to say that the research community generally accepts the notion of trust as a basis for secure interactions in ubicomp [English2002]. We have discussed how trust can be gained through automated negotiation earlier in this chapter. Here we discuss approaches that have been proposed over the years to build trust among entities in a distributed system.

PolicyMaker [Blaze1998] and KeyNote [Blaze1999] were seminal trust management projects where credentials (typically public keys) were tied to the permissions they represented rather than identity. An access required the production of a key which would be input to a compliance checker along with a request and a policy, the output being a yes/no answer. It was a simple and powerful model, but it has been overtaken by more sophisticated schemes that handle a much wider range of situations.

The Secure project [English2002; Cahill2003] argues for a dynamic notion of trust, with a history of past interactions being used as a basis for trust formation, and additional evidence leading to trust evolution. This trust information is then used for access control decisions using appropriately framed security policies. Secure also presents formal models that relate trust to events and results, and trust value changes based on evidence [English2004]. Trust building through reputation frameworks, or collection of evidence from known sources, has been well studied [Xiong2004]. Reputation models have been employed widely on the web, which provides a common forum for a large number of anonymous users to offer and obtain feedback about websites and products. Some of the most popular websites, including Amazon, Ebay and Bizrate act as reputation servers. An automated reputation-based augmentation has also been proposed for P3P [Kolari2005] using web crawling techniques. Though such reputation systems may offer economic benefits [Kennes2003], they have limited use in practical security, mainly because of the huge number of variables involved and the possibility of entities lying and colluding [Sen2002]. Formal reputation-based trust models generally use probabilistic reasoning, and do not clearly specify the decision-



making processes. Quantitative models have also been proposed to generate and make use of trust relationships. Shankar et al. have proposed a unified model that uses both identity and contextual properties and which expresses trust as a continuum [Shankar2002]. A different model attempts to model trust using probabilities, and in addition proposes ways to interpret the information during the actual process of performing a security-sensitive action [Jøsang1999].

These trust models are complementary to our research in that they could be used to frame policies and be used as heuristics in our protocol; the negotiation then enables a gradual evolution of trust based on history (sequence of messages exchanged up to that point). In the future, we envision a fusion of trust models and negotiation protocols as being the core features of any ubiquitous computing security framework.

# **Chapter 11**

## **Future Work: Extensions and Enhancements**

There are a number of ways in which our negotiation protocol can be enhanced and improved as a tool for enabling interoperation between mutually unknown computers connected by a network. Our research could be extended in many directions with the goal of designing negotiation-guided policy management modules for secure ubiquitous computing. Interaction with human users is another under-explored area. We discuss some of these extensions and enhancements in this chapter. First, we will discuss enhancements to the negotiation protocol. Then we will consider policy management framework performance and analysis issues. We will also propose other useful and fun ubicomp applications that would benefit from negotiation. Lastly, we will discuss the human computer interaction issues that have risen out of our research and how we could go about addressing them.

### **11.1. Negotiation Protocol**

The negotiation protocol was designed to be as general as possible (based on illocutionary speech acts) so that a wide range of applications could be supported. Our proof-of-concept consists of three parts: i) providing a basic building block for ubicomp middleware security modules, ii) demonstrating a wide range of dynamic application scenarios that require minimal effort to achieve, and iii) demonstrating the feasibility of

negotiation as a valid and practical procedure for policy resolution by comparing it against a centralized system. Enabling user access to ubiquitous services requires smarter devices and interoperation frameworks, and this protocol is the first step towards building a semi-expert system that will achieve those tasks. Though further advances in dynamic networking, context-aware computing and AI are awaited, we can explore and refine the properties of negotiations in certain ways that are described below.

### **11.1.1 Heuristics and Strategies**

In our measurements of negotiation performance (see Chapter 9), we used a simple heuristic for selection of alternatives: the cardinality of the alternative set. We order all the alternative sets in ascending order of cardinality and store them in a priority queue. Though this heuristic works fairly well (though nowhere close to optimal) on average simply because the probability of satisfying a set of  $k$  requests decreases with an increase in  $k$ . But all requests are not equal; we are not using any information extracted from the content of the requests themselves to make better selections. There is ample scope for research in the area of devising better heuristics, which in turn would manifest itself in a modified strategy. Architecturally, these heuristics could be designed as plugin functions selected at runtime by the policy manager controller module. We suggest a few candidate heuristics and strategic choices below.

*Categorization of Request Types*—Our informal classification of request types yields the set  $\{possessions, actions, state requests, queries/information\}$ . Based on the degree of

irrevocability and the risk involved in granting these requests, we propose an ordering of these classes as follows:

$$\textit{queries/information} < \textit{state requests} < \textit{actions} < \textit{possessions}$$

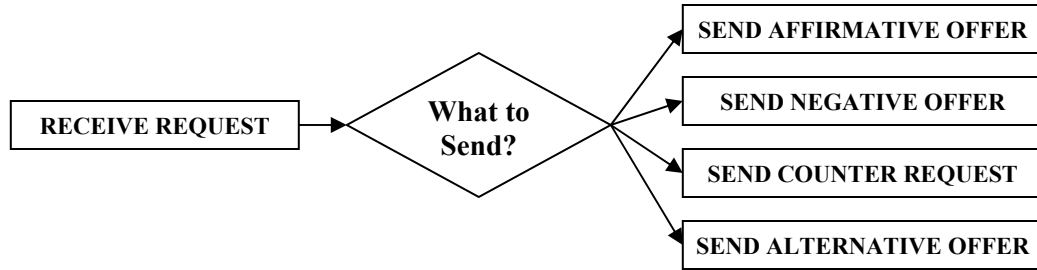
The ‘<’ symbol signifies a ‘less risky/less irrevocable’ relation. The ordering implies that a query for information is more likely to be satisfied than a request to change state, and so on. This is a total ordering, based on our understanding that giving up an object/possession is an irrevocable act, whereas some actions (such as making firewall settings) can be revoked based on future observations, and that most queries for information can be safely answered without risk of harm, though some privacy is lost. This is not a universal assertion; specific requests for information may turn out to be more risky than certain action requests. Using this ordering, an alternative set that contains more queries than action/possession requests will be selected first because the requests in this set are more likely to be satisfied by the opposite party in a realistic negotiation scenario.

*Expected time to finish*—If a negotiator has soft real-time constraints that are more important to it than its privacy concerns, it could make a strategic choice to send multiple alternative sets at once, giving the other party the option of trying to satisfy one of them. This would definitely reduce the total protocol termination time, and the amount of optimization depends on the stage at which this option is exercised. A key challenge here is to estimate the expected termination time with sufficient accuracy, since the processing times associated with different requests could vary widely. We could also experiment

with lazy and eager strategies in the same way as Winsborough et al. have in the area of automated trust negotiation [Winsborough2000].

### **11.1.2 Re-Modeling Negotiation**

The negotiation protocol allows a negotiator to compute alternative offers, should it not be able to send an affirmative reply to the primary one. As the protocol is based on logical resolution, it currently does not try to compute alternative offers when local queries based on received requests succeed. Yet, the protocol could be re-modeled and re-engineered to include non-logical criteria, including policy compromises, at choice points. The choice is indicated by the flowchart segment in Figure 74. The choice of how to reply when a request is received is not a purely policy-based decision here. (*Note:* heuristics for alternative selection lie within the bounds of logic, since they are simply employed to select one of a set of equally valid logical paths). Sometimes, instead of sending an affirmative offer, a negotiator could try to compute an alternative offer. At certain points, a negotiator could choose to send a negative offer instead of a counter-request. Likewise, instead of sending a counter-request because of the failure of a query formulated from a received request, a satisfiable alternative offer could be sent. We cannot envision a situation under which a negative offer would be sent even though an affirmative offer is consistent with local policy, but this is a topic that deserves further investigation. Below, we briefly suggest criteria upon which such choices can be made.



**Figure 74.** Negotiation Protocol Choice Point

*Balancing risks and benefits*—In our discussion of related work (see Chapter 10), we mentioned certain research projects that had tried to use perceived benefit and risk metrics to make trust and access control [Zhang2006] decisions. The SECURE project in particular [English2004] uses an event model to monitor state changes, to dynamically update trust levels in foreign entities and compute the costs of allowing a security- or privacy-sensitive action. Our negotiation protocol could adopt such a model and frame a dynamically computed *utility* ( $= \textit{benefit} - \textit{cost}$ ) metric; the choice with the highest utility would be selected. The question of how to assign suitable benefit and cost values to particular actions/concessions now arises. Let us just consider costs to be security risks or privacy loss. We can then come up with the following classification of risky actions: *{running applications/services upon request, offering access to resources, offering information or credential objects}*. The risks of performing these actions during negotiations are obvious. The benefits include satisfaction (or even quicker satisfaction) or one's goals by virtue of making risky concessions. What quantitative benefit/risk values one could attach to these risky actions is an open problem that we leave for future researchers. Possibly a partial ordering (based on relative risk or benefit levels) of actions

could be mined from the policy database in a similar way as access control roles have been [Molloy2008]. The expected time to finish could be incorporated into the benefit metric. Once values are assigned, policy statements in the database could be suitably annotated. Perhaps game-theoretic techniques could be leveraged, but we suspect those would not be suitable for this kind of negotiation, where both participants play by different rules (policies in their respective databases). The re-modeled protocol must also be analyzed for completeness, correctness, and optimality properties.

### **11.1.3 Collective Negotiation Among Multiple Parties**

Our framework enables multi-party negotiation as a set of bilateral negotiations. Yet this is not the full multi-party negotiation, the scenario of which was laid out in Chapter 3. Here, we propose a candidate protocol for true multi-party negotiation, where each party has goals that could be satisfied by a combination of other parties.

The negotiation proceeds in lock-step. In each round, every negotiator broadcasts a set of messages, each message intended for some of the other negotiators (we assume that the communication infrastructure provides multiple unicast as a core operation). The protocol starts by all negotiators sending their initial requests to everyone else. In response to a request received, offers could be returned or counter-requests could be generated. Offers are unicast only to the original requester, but counter-requests may be communicated to multiple negotiators, and even be distributed among them. Multiple offers could be received for the same request; one will be selected by the requester based

on some criterion that is not part of this model. Eventually the protocol terminates in the same way as in the case of the 2-party negotiation.

There are a number of challenges that we face before such a protocol can be fully designed and implemented. The procedures offered by the policy engine would have to be modified to compute counter-requests (unsatisfied constraints) for multiple entities rather than just one. Whether or not such a protocol can be correct or complete is also a matter for investigation. Implementation of reliability and fault tolerance mechanisms becomes a lot more complex in multi-party negotiation. Issues of collusion and Byzantine failure must also be addressed.

#### **11.1.4 Framing Binding Legal Contracts**

An interesting and fruitful target of future research could be the enforcement of legally binding agreements reached through negotiation. This would be particularly valuable in e-commerce applications, in which monetary transactions are conducted and goods are promised in return for money. Legality may have to be enforced by a trusted third party, akin to an e-notary, who is not involved in the negotiation procedure but is only involved in “attesting” and “recording” the agreement once negotiation terminates. Cryptographic protocols for transactions have been developed that maintain privacy while using trusted third parties [Micali2003], but enforcing the legality of a more arbitrary negotiation instance requires further work.



### **11.1.5 Security**

A Panoply policy manager monitors sphere events and reflects changes in the policy database. It also triggers renegotiations when the results of later negotiations cause change in state. For more comprehensive security, we should augment the manager with a reputation-based trust framework that monitors all possible violations of negotiated agreements and not only triggers renegotiations, but also propagates this information into a sphere trust network. These spheres' policies that are dependent on actions similar to the ones that have been violated could be suitably modified to reflect the lower level of trust in the violator sphere.

Given that neither negotiator knows anything about the other's policies, one side could execute a denial-of-service attack on the other simply by sending an endless sequence of requests. But, as we mentioned at the end of Chapter 8, this is not a very serious problem as negotiation messages cannot be sent out of order or flooded. A possible approach that could prevent this kind of attack is to ensure that negotiation steps result in some progress towards termination. Goal utility could be dynamically updated, whereby the utility of a request decreases with time or number of steps. Using a utility-based model as described in Section 11.1.2, we might be able to mitigate the effects of attempted denial-of-service attacks.

## **11.2. Policy Management Framework**

There is also room for research in either improving or enhancing the policy management middleware. We discuss some of these focus areas briefly in this section.

### 11.2.1 Performance

As mentioned earlier in this dissertation, the counter-request and alternative offer generation procedures (primarily the former, as it is employed more often) are negotiation performance bottlenecks, since neither negotiator can do any other useful work (like message communication) while these procedures are running. But as long as the policy rules don't change very often, these procedures perform repetitive tasks and generate identical results even when executed at different times and for different negotiators. Therefore, we can optimize performance by caching the result of these procedures in a table indexed by the request name. When two different negotiators make the same request, a *fast path* can be used in the second instance, and the overhead of counter-request generation can be avoided. This performance optimization comes at the price of guaranteed correctness and policy integrity. If the state of the policy database changes between the times when the identical requests were sent, negotiation may deliver a result inconsistent with policy. But this is definitely an optimization worth investigating. We have used a cache for a different purpose in the policy manager; it is used in the context of event-filtered access control (see Chapter 6). The resulting performance gains were found to be significant (see Chapter 9) and we would have no reason to expect otherwise in the case of the optimization proposed here.

### 11.2.2 Running the Policy Manager on Resource-Constrained Devices

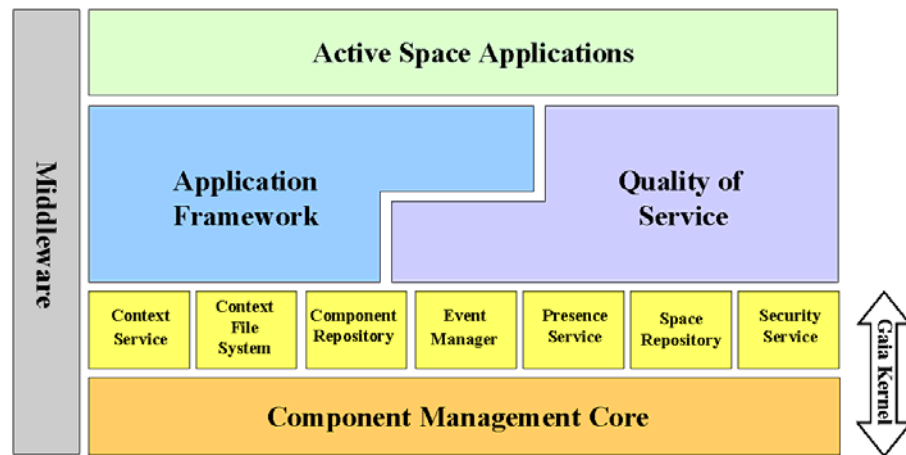
We have successfully run the policy manager (as part of the Panoply middleware) on devices running full featured Linux distributions. Panoply was also ported (without the Prolog subsystem) to smaller devices with fewer resources like the Nokia 770 Internet Tablet, though performance was less than optimal because the code base was built on Java. In the future, we would like to port the policy manager to devices like the Nokia 770 and even less capable devices like mobile phones and sensors. Porting the entire functionality to native C or embedded code (for non-JVM-enabled devices) will be a challenge, but doable. Prolog distributions are also available for smaller devices and mobile phones; e.g., *m-Prolog* [Koch2005]. In our experiments, we found that the policy manager running negotiation leaves a fairly small memory footprint as well. Successfully running computation-intensive helper functions will also be a challenge, though lightweight cryptographic schemes have been developed for small devices.

The biggest challenge in running a negotiation protocol would be the large amount of network communication potentially draining valuable battery charge. Reducing the number of negotiation steps will increase the power efficiency of the protocol as well. A pragmatic option for very low-capability devices like sensors may simply be to *outsource* negotiation altogether to more powerful devices that run Panoply sphere managers. If we really wanted such devices to run the negotiation protocol, the estimated battery time left could be incorporated in the utility heuristic; also, multiple alternatives could be bunched together as mentioned in Section 11.1.1.

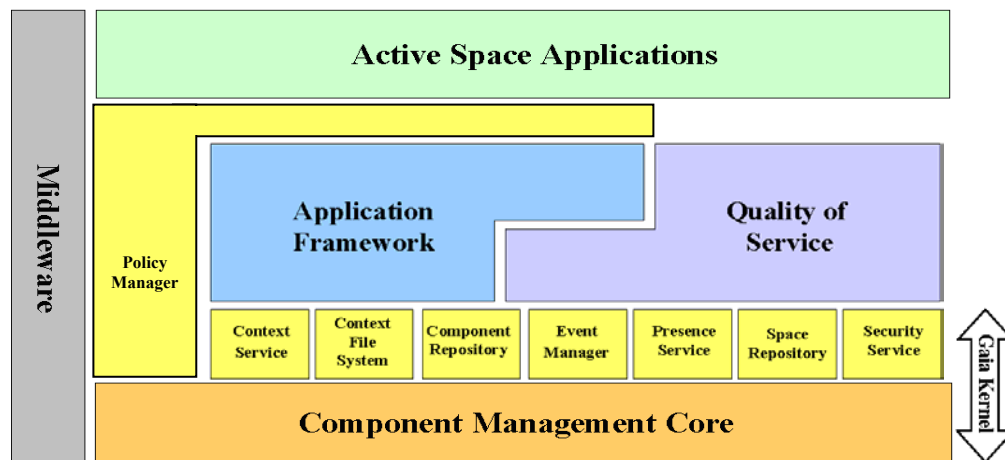
### 11.2.3 Porting to Other Ubicomp Platforms

We designed the negotiation protocol and the policy manager to be independent of the middleware platform they are a part of. Based on that philosophy, our policy manager is completely modular and is very loosely coupled with the other core Panoply components, communicating only through events. Porting the policy manager to a different ubicomp middleware is very feasible. Here we discuss how this could be done, using the Gaia framework [Román2002] as a case study.

Gaia acts as an operating system for active spaces, which consist of multiple computers and resources, similar to a Panoply sphere in some ways. The high-level Gaia architecture is illustrated in Figure 75. Our policy manager could be added to this architecture fairly easily, as illustrated in Figure 76. Gaia supports an event manager, like Panoply, and components and services could subscribe for events, including those that indicate context changes, presence of people, etc. The Gaia Space Repository supports a directory for active devices and services in a Gaia active space; this can be queried based on entity properties, returning a suitable XML description of the entity. The policy database must also contain this information in its policy database, so we would have to add a module that would query the Space Repository and convert XML descriptions into our Prolog-based policy language. This is not very straightforward, but can be done with some effort. The Gaia Security Service contains both access control rules for authorization purposes and pluggable authentication modules that can be dynamically invoked in different configurations. These modules could be registered with the policy manager as helper functions, and invoked when required as well.



**Figure 75.** GAIA Architecture (Borrowed from GAIA homepage: <http://gaia.cs.uiuc.edu/>)



**Figure 76.** GAIA Architecture Modified to Incorporate the Policy Manager

The policy manager listens for Gaia events and evaluates ECA rules as it does in Panoply. Negotiation would be typically invoked whenever a new device joined a Gaia active space and tried to register itself. Since Gaia-enabled devices do not incorporate the notion of negotiation, the registry procedure would have to be modified to fit in negotiation, similar to the way in which the Panoply Join protocol was modified. Also, as the Gaia policy manager mediates between the Application Framework and applications,

it could trap events destined for the applications and run them through a policy filter. Even though Gaia is a centralized model (Panoply is decentralized, with loosely coupled spheres), events destined for applications from active devices could be mediated through negotiation, resulting in a similar form of dynamic access control.

The actual implementation is rather straightforward. Gaia services are implemented as CORBA services, and are specified using the CORBA IDL (Interface Definition Language). Therefore, the policy manager will also be implemented as a CORBA service. Even though it is written in Java and the other Gaia services are written in C++, the source code for the policy manager need not be converted to C++. Just writing a policy manager interface using the IDL would suffice. CORBA provides standard support for both C++ and Java components, so the porting of our policy manager to Gaia would be, at best, a challenging programming assignment.

#### **11.2.4 Large-Scale User Studies**

Thus far we have experimented with writing and using policies that might look somewhat arcane to a non-technical user, though more intuitively understandable than XML-based policies. In the future, we would like to deploy the policy management framework more widely and have a large number of users set and modify policies themselves (through the GUI displayed in Figure 15). Based on user feedback (related to the difficulty in writing policies), we could refine the language by expanding the vocabulary and making harder distinctions between policies that ought to be left to system administrators and those that users can handle.

### **11.2.5 Stress Testing**

Our policy manager was designed to be resistant to failure and to dynamic changes in state. We have run multiple negotiations, both running in concurrent threads and serially on a long-term basis. The Java framework is generally resilient and efficient within its limits. The Prolog subsystem though is not as resilient to stress-testing as we would like. For example, running the Smart Party application for a long term (while also playing music) eventually causes the Prolog subsystem to crash. The reasons include memory leaks in the Prolog source code (which we fixed) and some 32/64 bit incompatibilities in the JNI (Java Native Interface) code written by SWI-Prolog designers. The designers admit that the JPL package is unstable when run in multi-threaded mode. Therefore, even though SWI-Prolog provides features and APIs that are perfect for our system, we may look for a different, and more stable, platform for our policy manager in the future.

### **11.2.6 Controlling Event Flow**

Kevin Eustice has already talked at length about event flow among Panoply spheres in his dissertation [Eustice2008b] and has defined a number of event-scoping criteria and mechanisms. Currently our policy management filters events only based on end points, i.e., when the events are known to be destined for a given destination sphere or an application. We can use policies to do more fine-grained and extensive event scoping. Other event-based information sharing [Singh2008] or access control [Hengartner2005] systems have also used policies for a similar purpose. We could address two main concerns: sending events to untrusted spheres, and flow control. The former case could

be handled through negotiation in the same way that application-destined events are. Failed negotiations would result in events not being broadcast along those paths. Such a negotiation would take place only for the first event in the flow, since the state changes would be reflected in the policy database. Alternatively, contents of events could be obfuscated or encrypted before being sent to an untrusted sphere. When the event eventually reaches a trusted sphere that possesses the capability to de-obfuscate its contents, normal event flow resumes. The rate at which events can be sent is also something that could be negotiated between neighboring spheres, thereby regulating event flow. We are confident that workable solutions to these goals can be obtained with more research.

#### **11.2.7 Integrating with Semantic Web Technologies**

To better conform to standards, Panoply (including the policy manager) should eventually be ported to, or augmented with, Semantic Web technologies. Our negotiation framework is equally applicable to agents negotiating for web services as it is to mobile hosts encountering ubiquitous networks. A more standard implementation (and one that could interoperate with existing web services) would use SOAP as a message exchange protocol and negotiation messages encoded in XML instead of serialized Java objects. The challenge here is to ensure that the policy engine and most of the controller can be retained as is while augmenting the front end with SOAP, HTTPS and XML-based negotiation messages.



### **11.3. Applications**

The negotiation protocol can be applied in a wide variety of scenarios; we have discussed only some of them in this dissertation (see Chapter 7). We describe a few candidate applications that demonstrate the use of negotiation and policy management in areas other than the ones we implemented.

#### **11.3.1 An Interactive Museum Tour**

A museum, divided as it is into multiple rooms and regions focused on particular pieces of art, offers an excellent use case scenario for policy management and negotiation. With Panoply support, each room could be configured as its own domain, as would every tour group visiting the museum. Every museum patron, from students in a school group to individual visitors, carries a personal mobile computer that would receive content relevant to the exhibit the user is close to, and one that is perceived as being his/her center of focus. These devices possess vouchers that indicate their owners' right to access such content. Content is obtained through negotiation with the individual rooms (which would be configured as Panoply spheres in our deployment). Individual guests obtain vouchers upon payment at the museum entrance. School group members do not have to pay, as their school has an agreement with the city authorities who run the museum. Their devices present valid school vouchers when requested during negotiation. The content offered to each user is different, and based on their age levels (inferred from their credential types), and the list of prior exhibits visited. User interests are ascertained during negotiation (some might be interested in a historical perspective whereas others

may be interested in artistic techniques), and they may also be offered hints about what room to visit next. Upon a selection, the user could be offered directions. We saw how such features were enabled by the policy manager in the Interactive Narrative (see Section 7.1). A manual option can also be added; a docent or teacher accompanying the group could dynamically change entire group preferences, e.g., from a description of art techniques to a historical perspective, thereby changing the nature of delivered content. During peak hours, the museum policy managers could collaborate in crowd control. This could be done in subtle ways (users not being provided content if they are in an already crowded room, thereby giving them an incentive to move away) or by providing explicit suggestions that they might have a more enjoyable time in a different gallery.

### **11.3.2 E-Commerce: A Shopping Mall**

E-commerce applications could benefit both from ubiquitous computing services and our negotiation framework. Consider a scenario where a customer walks into a shopping mall equipped with networking and localization facilities, and with each store configured as an autonomous domain. The customer's resource needs are reflected in his device's policy database, as are his policies governing payment, revealing credit card information, and priorities in terms of what is most urgently needed. This scenario offers room for multi-party negotiation, where the customer's device can simultaneously bargain with multiple stores for different items, and can obtain the best deals and discounts. More trust would have to be gained through negotiation before the customer's device is ready to provide a credit card number. To perform such transactions, the customer's device must also prove

that it is within the mall premises. During the customer's visit, stores (or the mall as a whole) could negotiate with his device for the right to send targeted advertisements in return for vouchers and discounts. Stores may choose to keep their final (negotiated) offers open for a limited period of time within which the customer is expected to physically appear at the store and presumably inspect the goods and complete the purchase; otherwise, any obtained discounts would be cancelled. Alternatively, the customer may have limited time and may choose to provide his shipping address, thus avoiding visiting the store. Some of these steps may require manual intervention, or a negotiation protocol augmented with heuristics that incorporate time (see Section 11.1) may be employed. Either way, a realistic and useful deployment of such a scenario requires changes to the negotiation protocol as it currently exists.

#### **11.4. User Interaction**

Though negotiation was conceived as an *under-the-covers* protocol that abstracts away the complexities of resource usage and context from the human user, its workings are not completely transparent to the user in its current incarnation. In the absence of a learning (or any other AI) component, the decisions made by the policy manager are limited by the set of policy rules framed by the users. Within the boundaries of policy rules framed using logical semantics we have proved that the negotiation protocol is correct and complete. But in order to make ubiquitous services more usable, and interactions more intelligible to users, better user interfaces must be designed, and user feedback ought to

be incorporated (potentially departing from a purely logical approach). We discuss certain directions that can be followed below.

#### **11.4.1 Negotiation: User Control and Feedback**

There are many situations where the policy manager may not resolve a situation to the complete satisfaction of the user. A negotiation may fail because the policies of the negotiators, as framed, contain irreconcilable conflicts. Alternatively, a negotiation that may or may not succeed at the end could take up more time than a user can tolerate. In these cases, it may prove useful as a practical matter to allow users more control over the protocol. This control can take two forms:

- i) *Changing policy during a negotiation so that results that were impossible earlier would become possible:* Consider an example where P and Q are negotiating, and P receives a REQUEST from Q. P evaluates the request against its policy database and makes a decision to either send an affirmative offer, a negative offer, or counter-requests. Conversion of a negative offer to an affirmative offer would result either in quicker progress or be the key to converting a failed negotiation into a successful one, or one leading to a better result from P's point of view. Consider the peer-to-peer application (see Section 7.3: third case). P's policies do not dictate compliance with anyone who requests the prohibition of networked applications. When P receives such a request, it would ordinarily decline it, but a variation would involve the user controlling P to override this default and instead send an affirmative offer. As a result, P would gain access to more disk space than it otherwise would have.

Would changing policy in the middle of a resolution procedure preserve the theoretical correctness properties proved in Chapter 8? Seemingly it would, since we posit that the negotiation procedure does not evaluate a policy (and hence, a request) more than once. Therefore, changing this policy would not affect the correctness of nodes in the policy resolution tree that have already been evaluated. Still, further investigation is needed to provide a comprehensive proof.

- ii) *Making strategic decisions at intermediate choice points that override the default heuristics that the controller module currently uses:* Following up on the above example, P may be able to generate a set of alternative counter-requests in response to a request from Q. Instead of using its built-in heuristic for alternative selection, the negotiation framework could prompt the user to make an alternative selection. This would be useful in cases where the user can detect subjective differences among alternatives while the negotiation controller (purely guided by the information in the policy database) cannot.

Such user intervention comes at a cost. The user may be drawn into making decisions he/she may not be completely qualified to make—the negotiation protocol was created to handle this problem. Also, users will take additional time to make their decisions, thereby increasing the overall negotiating time. This may not provide a net benefit, and may even result in harm if the user makes a bad decision. The additional overhead may also break the fault tolerance measures currently in place (see Section 6.6), depending on the timeout values set by the negotiators. Still, we feel the option of user intervention is a feature worth adding to our negotiation framework.

### 11.4.2 Feedback and Analysis

For a user to be able to effectively control a protocol instance, the policy manager must be able to provide feedback in a suitable form that the user can understand. Currently, the user gets limited feedback in case of failure. All the policy manager does is to simply state something like “*You failed to satisfy <request>*”. These requests and offers are communicated in the form of logical predicates. A system administrator may understand these but the average user will find these completely unintelligible. Options must be presented to a user in a form he/she will understand. A simple first step would just be to associate a natural language string with each predicate type. For example, the request `possess(diskSpace, D)` would be presented to the user in the form “*Do you want to give entity  $Q$  access to  $D$  amount of disk space?*”. This mitigates the problem of language, but the user still may not understand the implications of performing the requested action. For example, a user will not be concerned or even be aware of what it means for a service running on port  $XX$  to be closed and why he should agree to do so. Here, it would be helpful if the policy manager could provide certain cues to the user. As we have described, the negotiation protocol constructs a policy resolution tree, where a counter-request sent is the child of a request received. Therefore, the question posed to the user would be of the form “*Do you want to .... in return for <negotiator> doing ...?*”. It may also be useful to trace back to a root request (some levels higher in the resolution tree) that would eventually be satisfied by the user agreeing to change policy. There are significant research possibilities in this area. In fact, Nam Nguyen, one of the researchers working on Panoply, is designing an analysis framework for event-based ubiquitous

systems; his research results may prove relevant to the kind of user interface we have described here. In addition, Bonatti et al. [Bonatti2006] show how explanations for failure of a Protune negotiation can be provided to users; both fine-grained and global information can be provided. Their work could provide useful insights in designing a better explanation system for our policy manager.

In addition to providing a user feedback on what could be gained from potentially breaking a policy rule, it would be useful to let the user know about the risks incurred by doing so. This is a much harder problem, since the risks cannot be inferred by an examination of the protocol history. The risks, if they can be inferred at all, would require a full examination of the policy database. A tool one could use here is simulated forward chaining. We could simulate the change in policy and record the changes that occur in the database through the forward chaining algorithm. But this algorithm runs in exponential time, and is extremely slow in practice. This is an area where significant AI research is required for performance optimization. Also, what results are presented to the user and in what form is also a matter of investigation.

### **11.4.3 Usable Policies**

A number of policies listed in our discussion of negotiation-supported applications (see Chapter 7) are rather hard to read and understand for a non-technical user. Consequently, such a user would find it even harder to write such policies. Even users with a technical background may find it cumbersome to write policies without sufficient SWI-Prolog training. In the future, we would like to make the policy language more usable. The key

modification is *abstraction*, or keeping the more complex and technical portions of the language under covers. Ordinary users never get to see them, yet lose no control over their system configurations.

We intend to leverage the work done by Herzog and Shahmehri in making the setup of security policies more usable [Herzog2007]. Their work does not completely apply to our framework, since they advocate the setting up of policies as much as possible at runtime rather than beforehand. On the other hand, negotiation needs pre-configured policies. Still, we can use some of their recommendations, such as the enforcement of least-privilege as an implicit default policy; policies are likely to be less cumbersome to write if the user does not have to explicitly add least-privilege constraints (which can be added automatically by a policy parser). Also, their experiences suggest that even the best designed policy set will be found wanting at runtime, so continuous tests and user feedback must be obtained, and the policy language easy to modify. Our logic-based language is easily modifiable, though we need to do real user tests before we can refine it. We can also leverage work done in building more usable GUIs [Shneiderman2004] and in establishing better criteria for successful HCI security [Johnston2003].

We add a couple of guidelines that should make policies more readable and writable: (i) the language must have few, simple, and intuitively understandable semantics, and (ii) the abstracted view should be as written as much as possible in natural language. Users could write policies using predicates written in natural language (and



whose meanings would be intelligible to them) and semantic operators. For example, consider the following policy rule (from Section 7.3):

```
update :-
    ((parentSphere(S), networkApp(App,P),
    not(runApp(prohibit,App)), not(running(App))) ->
    (((is_list(P),length(P,L),L>0) ->
    (jpl_datums_to_array(P,Parr0),
    jpl_datums_to_array([Parr0],Parr),
    jpl_call('panoply.utils.ApplicationLoader','launchApplication',[A
    pp,Parr],Res))) ;
    ((is_list(P),length(P,L),L=0) ->
    jpl_call('panoply.utils.ApplicationLoader','launchApplication',[A
    pp],Res))),
    assert(running(App))) .
```

This policy implies the following: *If I have a parent sphere and I have a network application ‘App’ that is currently not running, and is not prohibited from running, I must start it.* The predicates `parentSphere`, `networkApp`, `runApp(prohibit, App)`, and `running` are in a syntax that is close to a natural language, and can be understood by a user. The operator ‘`->`’ is the implication (if-then) operator, which can also be understood by the user. A simple natural language translator could be used to convert phrases like ‘*have a parent sphere*’ to the predicate ‘`parentSphere(S)`’ (the article ‘a’ is equivalent to the  $\forall$  operator and can be represented by a variable). The special (designated) predicates used by JPL for data processing (`jpl_datums_to_array`,

`is_list`, etc.), method invoking (`jpl_call`), and statement assertion (`assert`) could be hidden from the user through the following low-level policy rule:

```
startApplication(App) :-  
    (((is_list(P),length(P,L),L>0) ->  
    (jpl_datums_to_array(P,Parr0),  
    jpl_datums_to_array([Parr0],Parr),  
    jpl_call('panoply.utils.ApplicationLoader','launchApplication',[App,  
    Parr],Res))) ;  
    ((is_list(P),length(P,L),L=0) ->  
    jpl_call('panoply.utils.ApplicationLoader','launchApplication',[App]  
    ,Res))),  
    assert(running(App)).
```

This policy statement does not need to be changed unless the core JPL API changes; hiding this from the user does not result in any loss of control over system behavior. The user can thus specify a policy in natural language (using if-then and conjunction operators) and the translator could then translate that to a logical statement that the policy manager can understand and reason with.

## **Chapter 12**

### **Conclusion**

Automating interactions between computer systems gets more complex the higher we go up the network stack. This is because lower layers communicate fewer units of information (e.g., bits and frames at the MAC layer; packets at the network layer), whereas the higher layers (the application layers) must deal with a wider variety of entities, services, resources, data units, and context (this being especially important in a ubiquitous system). Planned interoperation does not scale, and one cannot anticipate every possible eventuality and put in place a suitable, efficient mechanism to deal with it. It is more manageable and scalable to determine the appropriate parameters of interactions at runtime, and make adjustments for context. In this dissertation we have demonstrated how spontaneous interactions can be enabled through negotiations at the application layer. The interacting end points need to have their state, possessions, implemented mechanisms, offered services, and available data encoded in the form of declarative logical policies. This allows negotiators to pose queries, and obtain logically consistent answers as well as unsatisfied constraints.

Our negotiation-based solution was predicated upon a particular environmental model that divides the world into autonomous domains. Each domain governs a certain group of entities within it and enforces a security perimeter around itself. Bounded domains will grow organically in the ubiquitous cyber-world, and map onto real-world

organizations of people. In the real world, people are individuals and are also affiliated with various groups, some temporary (such as being present in a museum with other patrons) and some permanent (such as being part of a family, a company, or a municipality). These domains have their own laws or constraints that their members are expected to abide by. Whenever two such domains interact in the real world, they come to an agreement through negotiations because their policies may conflict. When an entity or organization gets subsumed into a larger one, it is expected to abide by the policies of the latter, though it is free to do what it wants in all matters that are unrelated to the activities and priorities of the entire group. This model works well and preserves autonomy of domains in the real world, and we felt it was a natural model to apply in ubiquitous computing.

We modeled inter-domain interoperation as a negotiation. Policy constraints and resource possessions are framed as first-order logical statements, and goals are framed in logical predicates, both in Prolog syntax. The negotiation is designed to result in an agreement whereby the goals may be satisfied as framed, or equivalent alternatives may be provided, or the goals may be denied. Ensuring compliance with the negotiators' policies is the immutable criteria. The policy statements and goals have truth values in first-order logic, and we used theoretically verified logical properties to model such negotiation as a distributed policy resolution procedure. The goals are formulated as logical queries, and query processing involves building a search tree (as in the logical backward-chaining procedure) to find a solution that is logically consistent with the negotiators' policies. These policies may contradict, resulting in a failed negotiation. The

query resolution procedure performs an exhaustive search of the AND-OR tree (generated through policy dependencies), and backtracks upon failure. Since correctness and security are foremost in our priority list, this is unavoidable in our model. Also, privacy considerations prevent us from using a centralized entity to perform such policy resolution. We have shown how entities can make autonomous decisions about policy exposure, while distributing the policy resolution between the negotiators, all without sacrificing correctness properties.

We showed how we could translate this negotiation model into a working end-to-end protocol. This protocol consists of requests, counter-requests and offers. Alternatives are tried upon intermediate failure (equivalent to backtracking in the policy resolution tree) and the negotiation terminated when no more requests remain unresolved. We showed how this protocol is applicable to a wide variety of application scenarios by using concepts from linguistics (speech acts and illocutionary logic). Though speech acts have been considered as a basis for negotiation by other researchers, no one has gone so far as we have in the modeling of negotiation in logically correct terms, implementing it as part of a ubiquitous policy management framework, and wide-scale testing. Our high-level protocol is deterministic; the framework supports multiple concurrent negotiations and is tolerant to network and end-point failures. A layered architecture provides core query processing and constraint-extraction procedures in its back end, which are used to determine appropriate logical responses to messages. A control layer supports plugin capabilities for external heuristics and application-dependent helper functions as well.

We have demonstrated both the necessity and versatility of our negotiation protocol in mobile and ubiquitous computing scenarios through real-life applications. Using our negotiation protocol, conference room environments and attendees can come to suitable service access agreements that vary depending on what their goals are, what information and credentials they possess, and what they are willing to offer one another. Nomadic devices and semi-open networks could safely interact without the former becoming vulnerable to malicious or infected computers and the latter not losing its privacy and autonomy. A Panoply Smart Party could regulate access to its environment, its resources (like a smart door), and dynamically control access to its playlists based on who is requesting access and in what context. We showed how quantitative negotiations for disk access could be enabled through our protocol whenever one entity needed it from another (to run a peer-to-peer file sharing application, for example).

We proved and listed various theoretical properties of the negotiation protocol. It is free from deadlock, and suffers from livelocks only when there are cycles in the collective policy set of the negotiators; termination can be guaranteed with simple modifications and extra checks. Theoretically, the protocol is trivially correct; i.e., the result is consistent with policy and a negotiator never concedes more than what is requested of it. Our analysis does indicate that intermediate effects of non-logical helper functions may prevent results that would otherwise be possible. We showed that our protocol would be provably correct and complete in situations where such intermediate effects were reversible.

Our protocol is not guaranteed to be optimal. Therefore, we compared negotiation with a centralized policy resolution using an oracle, which always selects the correct alternative at every negotiation step. In our comparisons, we used a simple heuristic: the alternative sets were ordered in ascending order of cardinality, and the available set of least-size is selected. We designed and implemented a test case generation procedure, through which we could generate random pairs of policy databases that are characterized by the maximum branching factor and depths of policy resolution trees that can be generated from them. We tested negotiation performance for a range of branching factors and depth values. The number of actual negotiation steps was found to increase linearly with the number of actual negotiation steps, which indicates that the protocol and our simple heuristic do scale. Also, even though total processing times for negotiations dominate total oracular processing times, the corresponding average processing time per step is lower for negotiations. We also found that an increase in the maximum branching factor affected negotiation performance much more significantly than an increase in the maximum depth. Since database branching factors can be bounded more effectively in practice than depth, these results indicate that negotiation performance will not significantly degrade when database sizes increase.

### **Summary of Thesis Contributions**

- Design and implementation of a generic negotiation protocol based on illocutional speech acts, and modeling this in the form of a distributed policy resolution.
- Support for dynamic access control through negotiation.

- Design and implementation of a policy management framework for ubiquitous environments that supports multi-threaded fault tolerant negotiations as well as renegotiations.
- Building a test case generator and performing large-scale statistical comparison of centralized and decentralized policy resolutions.

## **Lessons**

Our research has made a significant contribution towards enabling spontaneous interoperation. We have shown that it is possible to design an automated negotiation framework based on declarative logical policies. We have demonstrated its worth in a real ubiquitous computing middleware and in many practical applications. Our system also allows dynamic access control and is sensitive to runtime policy changes. We have also implemented a test case generator that could be useful to other researchers in the future. Our performance results indicate that distributed policy resolution is feasible.

Yet, over the course of the project we discovered that there are still many hard problems that remain to be solved. Though the performance results indicated feasibility of negotiation, a large number of test cases resulted in sub-optimal trees, implying that there is ample room for the development of better heuristics. Automated negotiations can be achieved within a tight theoretical model. If we veer off from the model, such as by introducing non-logical external function and policy compromises, we lose the theoretical properties of non-trivial correctness and completeness because the resulting modification of state may introduce many complications. Also, a large number of situations require



real AI, which our system was not designed to handle. As we have mentioned before, trust and risk factors play a large role in determining the appropriate course of action in security-sensitive systems. So a negotiator should try to break policy and make compromises as the situation requires, being aware of the risks. Mistakes may occur; the framework must learn from these and use that knowledge to make better decisions in the future. Also, a truly intelligent system would have to discern user intention and possibly frame a large portion of the policy itself. Our policy language is also not readily usable by a wider audience, as its use requires a high level of technical sophistication. These drawbacks facing our system require solving extremely challenging problems, and though some work has been done in these areas, a lot more needs to be done.

At a high level, our framework could be placed in the category of semi-expert systems, as it runs logical queries and search algorithms to find answers so that users don't have to, yet is not *intelligent* enough to learn, evaluate, and correct itself.

The realization of true ubiquitous computing requires advances in systems, networks, and AI research. But even though significant advances have been made in the two former areas, as well as in the area of building better processors and smaller mobile computers, AI research has yet to catch up. Research in ambient intelligence is ongoing. As the field advances, we will determine ways to improve interoperation and negotiation frameworks so that the systems of the future can retain their autonomy while being more secure and usable.

## Appendix A: Policy Language Reference

Here we provide a reference to the common predicates and constants that comprise our policy language. We describe the global vocabulary that is understood by negotiating spheres and common low-level policy rules that enable mechanisms used by negotiators in our demonstrated applications. We also indicate which predicates users are expected to see and understand, and distinguish them from designated predicates that implement a particular operating system, networking, or string processing mechanism. We only list the SWI-Prolog predicates used; a comprehensive reference can be obtained from the SWI-Prolog home page [SWIProlog].

(Note: The ‘%’ symbol denotes a comment in Prolog syntax)

### Commonly Used SWI-Prolog Constructs

#### *Modification Predicates*

```
dynamic(Statement) .
consult(Filename) .
assert(Statement) .
retract(Statement) .
abolish(Statement) .
erase(StatementReference) .
```

#### *Examination Predicates*

```
clause(Head, Body) .
clause(Head, Body, Reference) .
nth_clause(Pred, Index, Reference) .
functor(Term, Functor, Arity) .
predicate_property(Head, Property) .
term_variables(Term, List) .
findall(Template, Goal, Bag) .
bagoff(Template, Goal, Bag) .
unify_with_occurs_check(Term1, Term2) .
arg(ArgNum, Term, Value) .
```

#### *Logical Terms*

```
true.
false.
not(Statement) .
```

#### *Data Processing Predicates*

```
atom_concat(Atom1, Atom2, Output) .
get_char(Char) .
get_char(Stream, Char) .
string_length(String, Length) .
string_concat(Str1, Str2, Output) .
string_to_atom(String, Atom) .
term_to_atom(Term, Atom) .
```

#### *JPL Predicates*

```
jpl_new(Class, Params, Reference) .
jpl_call(Ref, Method, Params, Result) .
jpl_get(Class_or_Object, Field, Data) .
```

#### *Operating System Operations*

```
shell(Command) .
shell(Command, Status) .
open(Filename, Mode, Stream) .
close(Stream) .
delete_file(Filename)
```

## Global Vocabulary

```
% Declares which predicates can be requested in a negotiation.
    requestable(PredicateName).
% We declare the following as being 'requestable' predicates.
    requestable(possess).
    requestable(action).
    requestable(memberIn).
    requestable(location).
    requestable(Pred) :-
        stateToRequest(Pred, OtherPred),
        requestable(OtherPred).

% Declares which predicates are understandable globally.
% In a negotiation, these predicates would be communicated as support
% strings, or extra predicates.
    global(PredicateName).
% We declare the following as being 'global' predicates.
    global(locationVoucher).
    global(socialVoucher).
    global(voucher).
    global(dispatch).
    global(location).
    global(printer).
    global(file).
    global(mediaFile).
    global(F) :-
        file(F, P), not(publicKey(K, F, P)).
    global(S) :-
        sphere(S).
    global(S) :-
        number(S).

% Declares mapping from a Request predicate to a State predicate.
    stateToRequest(RequestPredicate, StatePredicate).
    stateToRequest(memberIn, member).      % Globally asserted mapping
% Declares which predicates denote state information.
    statePred(StatePredicate).
    statePred(member).      % Globally asserted state predicate

% Indicates possession.
    possess(VAR).      % I possess VAR
    possess(S, VAR).   % Sphere S possesses VAR

% Indicates that S has access to VAR.
    access(S, VAR).

% Mechanism to perform action 'ActionType' on 'Argument'
    action(ActionType, Argument).
% Denotes that sphere S is required to perform action 'ActionType'
% on 'Argument'
    action(S, order, ActionType, Argument).
```

```

% Denotes that permission has been obtained from sphere S to perform
% action 'ActionType' on 'Argument'
    action(S,permission,ActionType,Argument) .

% Indicates that I must obey S's request to perform action ActionType.
    obey(S,ActionType) .
% Indicates that I must obey S's request to perform action ActionType
% on argument Argument.
    obey(S,ActionType,Argument) .

% Denotes that S has access to the information P.
% P is a predicate of the form 'pred(arg1,arg2,.....)' .
    accessInfo(S,P) .

% Denotes that S is a member of my sphere.
    member(S) .
% Denotes that I am a member in sphere S.
    memberIn(S) .

% Denotes that 'Request' must be requested; 'Type' may be:
% 'q' (query for data/info) or 'r' (any other kind of request).
    request(Request,Type) .

% Denotes that 'Answer' must be offered in response to 'Request'.
    offer(Request,Answer) .

% Sphere relationships.
    sphere(SphereID) .           % SphereID denotes a valid sphere
    childSphere(SphereID) .      % SphereID is my child sphere
    parentSphere(SphereID) .     % SphereID is my parent sphere
    relation(SphereID) .         % sphere SphereID is related to me
    relation(SID1,SID2) .        % spheres SID1 and SID2 are related
    negotiator(SphereID) .       % SphereID is my current negotiator
    localSphereID(SphereID) .    % SphereID is my ID
    candidateSphere(SphereID) .   % SphereID is a candidate for my
                                % sphere
    candidateInSphere(SphereID) . % I am a candidate in sphere SphereID

% Predicate used for event filtering:
% So = event source sphere ID; T = event type;
% ST = event subtype; UT = event user type.
    condition(So,T,ST,UT) .

% Denotes that predicate S2 is an acceptable offer (request) compared
% to S1.
    acceptable(S1,S2) .
% Denotes that predicates S1 is a preferable offer (request) compared
% to S2.
    preferable(S1,S2) .

% Indicates the resources or requests have been obtained from sphere
% SID. The counter-request generation algorithm is run on this
% clause at the beginning of every negotiation.
    needForResources(SID) :- <Body_Pred1>,<Body_Pred2>,<.....> .

```

## Application-Specific Vocabulary

```
% Indicates specific actions
    action(closePort,Po).           % Closing of port Po.
    action(runApp,App).             % Running Panoply application App.
    action(changeCharacter,C).      % Changing Interactive Narrative
                                    % Character to C.

% Indicates possession of a certain amount of disk space
% Used in the peer-to-peer file sharing application.
    possess(S,diskSpace,D). % Sphere S possesses D amount of disk space.
    possess(diskSpace,D).    % I possess D amount of disk space.

% Cert denotes an X.509 certificate.
    certificate(Cert).
    certificateIssuer(Cert,SID). % Sphere SID issued Cert.

% Denotes known voucher marked by the name 'V'.
    voucher(V).
    locationVoucher(V).
    socialVoucher(V).

% Denotes properties of location voucher 'V'.
    locationVoucher(V,Ver). % V has voucher ID 'Ver'.
    locationVoucher(V,Ver,Vee). % V has voucher ID 'Ver' and vouchee
                                % 'Vee'.
    locationVoucher(V,Ver,T). % V has voucher ID 'Ver' and
                                % expiration time T.
    locationVoucher(V,Ver,Vee,T). % V has voucher ID 'Ver', vouchee ID
                                % 'Vee', and expiration time T.

% Denotes properties of social voucher 'V'.
% Indicates affiliation with social group G.
    socialVoucher(V,G). % V has voucher ID 'Ver'.
    socialVoucher(V,G,Ver). % V has voucher ID 'Ver'.
    socialVoucher(V,G,Ver,Vee). % V has voucher ID 'Ver' and vouchee
                                % 'Vee'.
    socialVoucher(V,G,Ver,T). % V has voucher ID 'Ver' and
                                % expiration time T.
    socialVoucher(V,G,Ver,Vee,T). % V has voucher ID 'Ver', vouchee ID
                                % 'Vee', and expiration time T.

% Policies indicating dependencies among predicates that describe
% voucher properties.
    voucher(V) :- locationVoucher(V).
    voucher(V) :- socialVoucher(V).
    voucher(V,G) :- locationVoucher(V,G).
    voucher(V,G) :- socialVoucher(V,G).
    voucher(V,G,Ver) :- locationVoucher(V,G,Ver).
    voucher(V,G,Ver) :- socialVoucher(V,G,Ver).
    voucher(V,G,Ver,Vee) :- locationVoucher(V,G,Ver,Vee).
    voucher(V,G,Ver,Vee) :- socialVoucher(V,G,Ver,Vee).
    voucher(V,G,Ver,Vee,Dur) :- locationVoucher(V,G,Ver,Vee,Dur).
```

```

voucher(V,G,Ver,Vee,Dur) :- socialVoucher(V,G,Ver,Vee,Dur) .

locationVoucher(V) :- locationVoucher(V,C) .
locationVoucher(V,C) :- locationVoucher(V,C,C1),not(number(C)) .
locationVoucher(V,C,C1) :- locationVoucher(V,C,C2,C1),number(C1) .
locationVoucher(V,C,C1) :-
    locationVoucher(V,C,C1,C2),not(number(C1)),number(C2) .
locationVoucher(V,C,C1) :-
    locationVoucher(V,C1,C,C2),not(number(C1)),number(C2) .
locationVoucher(V,Vee) :-
    localSphereID(Ver),locationVoucher(V,Ver,Vee) .

socialVoucher(V) :- socialVoucher(V,G) .
socialVoucher(V,G) :- socialVoucher(V,G,C) .
socialVoucher(V,G,C) :- socialVoucher(V,G,C,C1),not(number(C)) .
socialVoucher(V,G,C,C1) :- socialVoucher(V,G,C,C2,C1),number(C1) .
socialVoucher(V,G,C,C1) :-
    socialVoucher(V,G,C,C1,C2),not(number(C1)),number(C2) .
socialVoucher(V,G,C,C1) :-
    socialVoucher(V,G,C1,C,C2),not(number(C1)),number(C2) .
socialVoucher(V,G,Vee) :-
    localSphereID(Ver),socialVoucher(V,G,Ver,Vee) .

% Public Key 'Key' is stored in file 'File' in folder 'Dir'.
    publicKey(Key,File,Dir) .
% Voucher 'Voucher' is signed using Public Key 'Key'.
    voucherKey(Voucher,Key)

% Sphere 'SphereID' is mapped to IP Address 'IPAddress'
    ipAddress(SphereID,IPAddress) .

% Indicates location 'L' of a sphere.
% Used in Interactive Narrative Application.
    location(L) .                % I am in location L.
    location(SphereID,L) .       % Sphere 'SphereID' is in location 'L'.

% Denotes that 'P' is a printer.
    printer(P) .

% Denotes that 'D' is a display.
    disp(D) .

% Denotes that 'D' is a door.
    door(D) .

% Governs sound playing from time T1 to T2.
% Used in Conference Room application.
    sound(S,prohibit,T1,T2) .    % Sphere S is prohibited from playing
                                % sound from time T1 to T2.
    sound(S,play,T1,T2) .        % Sphere S may play sound from time
                                % T1 to T2.
    sound(prohibit,T1,T2) .      % I am prohibited from playing sound
                                % from time T1 to T2.
    sound(play,T1,T2) .          % I may play sound from time T1 to T2.

```

```

% These predicates govern the state of a door.
% Used in Smart Party application.
    doorOpen(D) .           % Door D is currently open
    openD(D) .             % Door-Open Event has been scheduled
    closedD(D) .           % Door-Close Event has been scheduled

% These predicates govern the running of a Panoply application App.
    runApp(S,prohibit,App) . % S is prohibited from running App.
    running(App) .          % App is currently running within my sphere.
    netWorkApp(App,Params) . % App is a networked application launched
                           % with parameters Params.

```

## Low-Level Policy Rules (Global)

```

% Policy manager initialization requires running an 'initialize' query.
% This results in the querying of all predicates in the body of every
% rules similar to the following.
    initialize :- <BodyPredicates>.
% For example: the following results in the 'request' predicate being
% set as a 'dynamic', or modifiable predicate.
% A static predicate, on the other hand, cannot be modified.
    initialize :-
        not(predicate_property(request(V1),number_of_clauses(N)))->
        (not(predicate_property(request(V1),dynamic))->
        dynamic(request/1)).

% Negative predicate (denotes opposite effect)
    negative(Pred,N) :- functor(Pred,F,A),string_concat('not_',F,N).

% Number of children spheres
    numChildren(N) :-
        jpl_call('panoply.policy.PolicyEngine','countFacts',
        ['childSphere(C)'],N).

% Number of parent spheres
    numParents(N) :-
        jpl_call('panoply.policy.PolicyEngine','countFacts',
        ['parentSphere(C)'],N).

% Number of relative spheres
    numRelatives(D,N) :-
        numChildren(C), numParents(P), N is (C+P).

% Extract sphere name from sphere ID
    sphereName(SID,N) :-
        jpl_call('panoply.policy.Helper','sphereName',[SID],N).

```

```

% Check whether a sphere is a relation
relation(R) :-
    childSphere(R) ; parentSphere(R) .

% Get event type from object reference
eventType(E,T) :-
    jpl_call(E,'getTypeVal',[],V),
    jpl_call('panoply.event.EventType','getName',[V],T) .

% Get event subtype from object reference
eventSubType(E,ST) :-
    jpl_call(E,'getSubType',[],EST),
    jpl_call(EST,'toString',[],ST) .

% Get event user type from object reference
eventUserType(E,UT) :-
    jpl_call(E,'getUserType',[],UT) .

% Get event source from object reference
eventSource(E,So) :-
    jpl_call(E,'getSourceSphere',[],SID),
    jpl_call(SID,'toString',[],So) .

% Event filter: Check whether event with ID = EID can be passed
action(pass,event,EID) :-
    jpl_call('panoply.policy.EventPolicyMediator','currentEvent',[
    EID],E),
    eventType(E,T),
    eventSubType(E,ST),
    eventUserType(E,UT),
    eventSource(E,So),
    condition(So,T,ST,UT) .

% Instantiate and send an event with given attributes immediately.
send_event(EventType,EventSubType,UserType) :-
    string_concat(EventType,':',S1),
    string_concat(S1,EventSubType,S2),
    string_concat(S2,':',S3),
    string_concat(S3,UserType,S),
    string_to_atom(S,E),
    jpl_call('panoply.policy.Helper','execAction',['Event',E,'-
1'],@true) .

% Instantiate and send an event with given attributes after 'Time'
% seconds.
send_event(EventType,EventSubType,UserType,Time) :-
    string_concat(EventType,':',S1),
    string_concat(S1,EventSubType,S2),
    string_concat(S2,':',S3),
    string_concat(S3,UserType,S),
    string_to_atom(S,E),
    jpl_call('panoply.policy.Helper','execAction',['Event',E,Time],@true) .

```



```

% Read next character from a file stream
readFile(S, '') :-
    at_end_of_stream(S).
readFile(S, R) :-
    not(at_end_of_stream(S)),
    get_char(S, C),
    readFile(S, T),
    atom_concat(C, T, R).

% Run an operating system command 'Prog' with optional result 'Result'
action(run, Prog) :-
    shell(Prog, 0).
action(run, Prog) :-
    file(Prog, Dir),
    atom_concat(Dir, '/', COM1),
    atom_concat(COM1, Prog, COM),
    shell(COM, 0).

% Run an operating system command 'Prog' with result 'Result'
action(run, Prog, Result) :-
    jpl_call('panoply.policy.Helper', 'generateTemporaryFileName', [
], TF),
    atom_concat(Prog, ' 1> ', C1),
    atom_concat(C1, TF, C),
    shell(C),
    open(TF, read, Stream),
    (readFile(Stream, Result1) ->
        (close(Stream),
         delete_file(TF))),
    string_length(Result1, L),
    ((L>0 ->
        (L1 is L-1,
         sub_string(Result1, 0, L1, 1, Result))) ;
     (L==0 ->
        (Result=''))).

action(run, Prog, Result) :-
    file(Prog, Dir),
    atom_concat(Dir, '/', Prog1),
    atom_concat(Prog1, Prog, Prog2),
    jpl_call('panoply.policy.Helper', 'generateTemporaryFileName', [
], TF),
    atom_concat(Prog2, ' 1> ', C1),
    atom_concat(C1, TF, C),
    shell(C),
    open(TF, read, Stream),
    (readFile(Stream, Result1) ->
        (close(Stream),
         delete_file(TF))),
    string_length(Result1, L),
    ((L>0 ->
        (L1 is L-1,
         sub_string(Result1, 0, L1, 1, Result))) ;
     (L==0 ->
        (Result=''))).

```

```

% Obtain the k'th token 'Tok' of string 'Str' with <TAB> delimiter
kthToken(Str,Tok,K) :-
    jpl_new('java.util.StringTokenizer',[Str,' \t'],ST),
    kthTokenStrTok(ST,Tok,K),
    term_to_atom(Tok,Token).
kthTokenStrTok(ST,Tok,0) :-
    jpl_call(ST,'hasMoreTokens',[],@true),
    jpl_call(ST,'nextToken',[],Tok).
kthTokenStrTok(ST,Tok,K) :-
    K > 0,
    jpl_call(ST,'hasMoreTokens',[],@true),
    jpl_call(ST,'nextToken',[],NextTok),
    K1 is K-1,
    kthTokenStrTok(ST,Tok,K1).

% Get mapping from atom A to number N
atom_to_num(A,N) :-
    atom_chars(A,C), number_chars(N,C).

% Find the sum of numbers in a list.
sumList(L,0) :- length(L,0).
sumList(L,S) :- length(L,1),L=[S|L1].
sumList(L,S) :- length(L,K),K>1, L=[F|L1], sumList(L1,S2),S is F+S2.

```

## Low-Level Policies (Local) and Implemented Mechanisms

```

% Mechanism to play file 'File' using 'mplayer' (Linux)
action(play,File) :-
    atom_concat('mplayer ',File,C),
    shell(C,0).

% Mechanism to allow sound to be played (Linux)
action(play,sound) :-
    shell('amixer set Master 100% unmute',0).

% Mechanism to mute sound (Linux)
action(prohibit,sound) :-
    shell('amixer set Master mute',0).

% Mechanism to allow sound to be played (Linux)
action(permit,sound) :-
    shell('amixer set Master 100% unmute',0).

% Mechanism to set firewall rule using IPTables to shut off incoming
% connections to port Po (Linux).
action(closePort,Po) :-
    atom_concat('iptables -A INPUT -j DROP -p tcp --dport',Po,C1),
    atom_concat(C1,' -i lo',C), shell(C,0).

```

```

    action(print,F) :-
        file(F), printer(P), printerCommand(P,C), atom_concat(C,' <
        ',C1),
        atom_concat(C1,F,COM), shell(COM,0).

% Check whether port Po is blocked using IPTables (Linux).
% If not closed, it must be asked to close Po.
closedPort(S,Po) :-
    (negotiator(Thr,G);childSphere(G)),
    ipAddress(G,IP),
    atom_concat('nmap -sS ',IP,C0),
    atom_concat(C0,' -p ',CP),
    atom_concat(CP,Po,C1),
    atom_concat(C1,' | grep ',C2),
    atom_concat(C2,Po,C3),
    atom_concat(C3,'/tcp',C),
    action(run,C,Result),
    string_to_atom(Result,Res),
    kthToken(Res,'open',1),
    action(S,order,closePort,Po).
closedPort(S,Po) :-
    (negotiator(Thr,G);childSphere(G)),
    ipAddress(G,IP),
    atom_concat('nmap -sS ',IP,C0),
    atom_concat(C0,' -p ',CP),
    atom_concat(CP,Po,C1),
    atom_concat(C1,' | grep ',C2),
    atom_concat(C2,Po,C3),
    atom_concat(C3,'/tcp',C),
    action(run,C,Result),
    string_to_atom(Result,Res),
    (kthToken(Res,'filtered',1) ; kthToken(Res,'closed',1)).

% Check how much disk space is free and available to allocate.
allocatableFreeDiskSpace(FS) :-
    freeDiskSpace(FDS), allocatedDiskSpace(ADS), FS is FDS-ADS.

% Check how much disk space has been allocated.
allocatedDiskSpace(FS) :-
    findall(D,(sphere(S),possess(S,diskSpace,D,Path)),LS),
    sumList(LS,FS).

% Check how much disk space is free (Linux).
freeDiskSpace(FS) :-
    action(run,'df /dev/hda5 | grep hda5',Result),
    string_to_atom(Result,Res),
    kthToken(Res,T,3),
    atom_to_num(T,FS).

% I possess 'D' KiloBytes allocatable, mounted on 'Path'.
possess(diskSpace,D,Path) :-
    freeDiskSpace(F), allocatedDiskSpace(A),
    ((var(D) -> D is F-A); ((integer(D);float(D)) -> D =< F-A)).

```

## Appendix B: Sample Database Generated for Performance Testing

In Chapter 9, we described the procedure for generating test cases for comparing the performance of the negotiation protocol to a centralized oracle. Each test case consists of a pair of policy databases and a pair of request sets; one request and one policy set belongs to each negotiator. Given below is an example test case that was generated by our procedure based on the parameter values  $\langle b_{max}, d_{max}, numRules \rangle = \langle 6, 8, 28 \rangle$ .

### Database 1:

#### *Facts*

<pre> storage(px6) . voucher(ht) . type(zp,bw) . voucher(vk) . printerName(pjff) . disp(l) . group(py,acm) . tim(gh) . brand(o, hp7100) . parentName(htm) . possess(diskSpace,0) . displayName(ck) . directory(ec,v) . file(fil) . type(xz1,color) . voucher(c) . </pre>	<pre> printerName(jy) . directory(py,v) . childName(mwd) . displayName(t3) . brand(yv, hp4150) . groupSize(g2) . door(zp) . possess(ht) . action(open,w) . tim(mu) . printer(o) . brand(e1, hp4150) . possess(x) . brand(ht, hp4150) . voucher(ec) . directory(l, a2p) . </pre>	<pre> groupSize(d) . displayName(q) . childName(xe) . possess(o) . type(uf,color) . brand(yv, hp7100) . printerName(f) . printer(x) . type(ec,color) . type(ht,bw) . printer(w) . action(run,xz1) . voucher(xz1) . door(ol2) . memberIn(asphere) . door(uf) . directory(c, kg) . </pre>
--	---	---

## *Rules*

```
obey(X,open) :- printer(VAR),displayName(X, VAR001723).
accessInfo(X,(tim(VAR01213))) :- possess(X,
VAR0683),voucher(VAR0683),type(VAR0683, VAR1684).
access(X,diskSpace,VAR) :- childName(X, VAR023638).
obey(X,run) :- voucher(VAR),type(VAR,VAR1),possess(X, diskSpace, VAR1610).
obey(X,open) :- printer(VAR),memberIn(X).
obey(X,open) :- printer(VAR),childName(X, VAR023592).
accessInfo(X,(tim(VAR01213))) :- possess(X, diskSpace, VAR1573).
accessInfo(X,(tim(VAR01011))) :- action(X, order, open,
VAR0554),printer(VAR0554).
accessInfo(X,(childName(VAR023))) :- memberIn(X).
obey(X,open) :- printer(VAR),possess(X, diskSpace, VAR1481).
accessInfo(X,(childName(VAR023))) :- displayName(X, VAR001382).
access(X,VAR) :- voucher(VAR),type(VAR,VAR1),childName(X, VAR023354).
obey(X,run) :- voucher(VAR),type(VAR,VAR1),action(X, order, open,
VAR0144),printer(VAR0144).
accessInfo(X,(tim(VAR01011))) :- displayName(X, VAR00120).
access(X,diskSpace,VAR) :- memberIn(X),printerName(X, VAR089771).
access(X,diskSpace,VAR) :- memberIn(X),childName(VAR023637).
access(X,VAR) :- voucher(VAR),type(VAR,VAR1),action(X, order, prohibit,
VAR074),printer(VAR074),brand(VAR074, VAR175),childName(VAR023353).
access(X,diskSpace,VAR) :- true.
obey(X,run) :- voucher(VAR),type(VAR,VAR1),action(open,
VAR0143),printer(VAR0143),action(X, order, closePort,
VAR0287),door(VAR0287),type(VAR0287, VAR1288),possess(diskSpace, VAR1609).
obey(X,open) :- printer(VAR),displayName(VAR001130),possess(diskSpace,
VAR1480),member(X),printerName(X, VAR089732).
obey(X,run) :- voucher(VAR),type(VAR,VAR1),action(X, order, closePort,
VAR040),door(VAR040),type(VAR040, VAR141).
obey(X,open) :- printer(VAR),possess(diskSpace,
VAR131),memberIn(X),childName(VAR023591),displayName(VAR001722).
accessInfo(X,(tim(VAR01213))) :- action(X, order, prohibit,
VAR0105),printer(VAR0105),brand(VAR0105, VAR1106),action(open,
VAR0250),printer(VAR0250),possess(X, diskSpace, VAR1432),possess(diskSpace,
VAR1572),possess(VAR0681),voucher(VAR0681),type(VAR0681, VAR1682).
accessInfo(X,(tim(VAR01011))) :- displayName(VAR00119),action(X, order,
prohibit, VAR0233),disp(VAR0233),directory(VAR0233,
VAR1234),possess(diskSpace, VAR1413),action(open, VAR0553),printer(VAR0553).
accessInfo(X,(childName(VAR023))) :-
memberIn(X),displayName(VAR001381),member(X).
accessInfo(X,(displayName(VAR001))) :- memberIn(X).
```

## Database 2:

### *Facts*

```
groupSize(zu).
door(uz).
type(l,color).
type(uf,color).
brand(yv, hp7100).
printerName(r).
bandwidth(x).
possess(uz).
groupName(v).
type(ec,color).
location(d9p).
door(ol2).
printer(e1).
memberIn(csphere).
door(uf).
```

```
action(prohibit,o).
door(py).
group(l,ieee).
groupSize(kh8).
door(zp).
printer(o).
action(closePort,zp).
door(yv).
brand(ht, hp4150).
voucher(ec).
directory(l,a2p).
location(d11).
groupSize(v4).
possess(diskSpace,37).
parentName(jn).
```

```
voucher(ht).
type(zp,bw).
childName(ey5).
voucher(vk).
disp(l).
group(py,acm).
action(prohibit,l).
brand(o, hp7100).
action(play,ol2).
memberIn(bsphere).
directory(ec,v).
type(xz1,color).
voucher(c).
displayName(rlc).
directory(ol2,yj).
```

## Rules

```
obey(X,prohibit) :- printer(VAR),brand(VAR,VAR1),possess(X, diskSpace,
VAR1742).
obey(X,closePort) :- door(VAR),type(VAR,VAR1),memberIn(X).
accessInfo(X,(bandwidth(VAR067))) :- printerName(X, VAR089654).
member(X) :- printerName(X, VAR089651).
access(X,diskSpace,VAR) :- printerName(X, VAR089643).
accessInfo(X,(tim(VAR045))) :- memberIn(X).
obey(X,prohibit) :- disp(VAR),directory(VAR,VAR1),possess(X, diskSpace,
VAR1491).
obey(X,closePort) :- door(VAR),type(VAR,VAR1),possess(X, diskSpace, VAR1451)
accessInfo(X,(bandwidth(VAR067))) :- tim(X, VAR045386).
access(X,VAR) :- door(VAR),printerName(X, VAR089329).
access(X,diskSpace,VAR) :- memberIn(X).
obey(X,prohibit) :- printer(VAR),brand(VAR,VAR1),memberIn(X).
obey(X,prohibit) :- disp(VAR),directory(VAR,VAR1),memberIn(X).
accessInfo(X,(bandwidth(VAR067))) :- memberIn(X).
accessInfo(X,(tim(VAR045))) :- action(X, order, prohibit,
VAR0192),printer(VAR0192),brand(VAR0192, VAR1193).
member(X) :- true.
member(X) :- printerName(VAR089187).
access(X,diskSpace,VAR) :- childName(X, VAR023177),printerName(VAR089642).
access(X,VAR) :- door(VAR),possess(X, diskSpace,
VAR161),printerName(VAR089328),action(X, order, open,
VAR0753),printer(VAR0753).
access(X,diskSpace,VAR) :- member(X),childName(X, VAR023633).
obey(X,prohibit) :- printer(VAR),brand(VAR,VAR1),displayName(X,
VAR00150),possess(X, diskSpace, VAR1167),member(X),possess(diskSpace,
VAR1741).
obey(X,prohibit) :- disp(VAR),directory(VAR,VAR1),action(X, order, open,
VAR0160),printer(VAR0160),member(X),possess(diskSpace, VAR1490).
obey(X,closePort) :- door(VAR),type(VAR,VAR1),tim(X, VAR0101125),action(X,
order, open, VAR0117),printer(VAR0117),possess(diskSpace,
VAR1267),possess(diskSpace, VAR1450),member(X).
accessInfo(X,(printerName(VAR089))) :- true.
accessInfo(X,(bandwidth(VAR067))) :- action(X, order, run,
VAR088),voucher(VAR088),type(VAR088, VAR189),member(X),tim(VAR045385),tim(X,
VAR01011526),printerName(VAR089653).
accessInfo(X,(tim(VAR045))) :- action(prohibit,
VAR0190),printer(VAR0190),brand(VAR0190, VAR1191),member(X).
```

### ***Requests from Negotiator1 to Negotiator2***

```
possess (VAR) , door (VAR)
action (order, play, VAR) , door (VAR) , directory (VAR, VAR1)
action (order, closePort, VAR) , door (VAR) , type (VAR, VAR1)
action (order, prohibit, VAR) , printer (VAR) , brand (VAR, VAR1)
action (order, prohibit, VAR) , disp (VAR) , type (VAR, VAR1)
member
location (VAR)
displayName (VAR)
printerName (VAR)
groupName (VAR)
groupSize (VAR)
bandwidth (VAR)
parentName (VAR)
childName (VAR)
```

### ***Requests from Negotiator2 to Negotiator1***

```
possess (VAR) , printer (VAR) , brand (VAR, VAR1)
possess (VAR) , voucher (VAR) , brand (VAR, VAR1)
action (order, open, VAR) , printer (VAR)
possess (VAR) , printer (VAR)
action (order, run, VAR) , voucher (VAR) , type (VAR, VAR1)
member
tim (VAR)
displayName (VAR)
printerName (VAR)
groupSize (VAR)
storage (VAR)
parentName (VAR)
childName (VAR)
```



## References

- [**Aarts1995a**] E. Aarts, "Complexity of Pure Prolog Programs," *Proceedings of Accolade 1995*, Editors: M. Trautwein, S. Fischer, Publisher: Dutch Graduate School in Logic (OZSL), Amsterdam, pp. 13-33, 1995.
- [**Aarts1995b**] E. Aarts, "Complexity of Horn Programs," *Proceedings of the 5<sup>th</sup> International Workshop on Logic Programming Synthesis and Transformation*, LNCS Vol. 1048, pp. 76-90, 1995.
- [**Abdoessalam2004**] A.M. Abdoessalam and N. Mehandjiev, "Collaborative Negotiation in Web Service Procurement," *Proceedings of the International Workshop on Agent-Based Computing for Enterprise Collaboration (WETICE 2004)*, IEEE CS Press, Modena, Italy. June 14-16, 2004.
- [**Adjie-Winoto1999**] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," *Proceedings of the 17<sup>th</sup> ACM Symposium on Operating Systems Principles*, pp. 186-201, Charleston, SC, December 1999.
- [**Agrawal2005**] D. Agrawal, S. Calo, J. Giles, K.-W. Lee, and D. Verma, "Policy Management for Networked Systems and Applications," *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*, Nice, France, May 2005.
- [**Agrawal2007**] D. Agrawal, S. B. Calo, K.-W. Lee, and J. Lobo, "Issues in Designing a Policy Language for Distributed Management of IT Infrastructures," *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM 2007)*, pp. 30-39, Munich, Germany, May 2007.
- [**Aït-Kaci1991**] H. Aït-Kaci, "Warren's Abstract Machine: A Tutorial Reconstruction," *MIT-Press 1991*, ISBN 0-262-51058-8, <http://www.vanx.org/archive/wam/wam.html>.
- [**Al-Muhtadi2003**] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and M. Mickunas, "Cerberus: A Context-Aware Security Scheme for Smart Spaces," *Proceedings of the 1<sup>st</sup> IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, March 23-26, 2003.
- [**Al-Muhtadi2004**] J. Al-Muhtadi, S. Chetan, A. Ranganathan, and R. Campbell, "SuperSpaces: A Middleware for Large-Scale Pervasive Computing Environments," *Proceedings of the IEEE International Workshop on Pervasive Computing and Communications (Perware 2004)*, Orlando, Florida, March 2004.

- [**Alves2006**] M. Alves, C. V. Damásio, W. Nejdl, and D. Olmedilla, “A Distributed Tabling Algorithm for Rule Based Policy Systems,” *Proceedings of the 7<sup>th</sup> IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, pp. 123-132, 2006.
- [**Andreoli2001**] J. M. Andreoli and S. Castellani, “Towards a Flexible Middleware Negotiation Facility for Distributed Components,” *Proceedings of the DEXA Workshop on E-Negotiation*, Munich, Germany, 2001.
- [**Andrieux2007**] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, “Web Services Agreement Specification (WS-Agreement),” *GRAAP Working Group, Open Grid Forum, (GFD-R-P.107)*, <http://www.ogf.org/documents/GFD.107.pdf>, March 14<sup>th</sup> May 2007.
- [**Antonioniou1999**] G. Antoniou, “A Tutorial on Default Logics,” *ACM Computing Surveys*, Vol. 31, Issue 4, pp. 337-359, December 1999.
- [**Ardagna2004**] D. Ardagna, C. Batini, M. Comerio, Marco Comuzzi, F. De Paoli, S. Grega, and B. Pernici, “Negotiation Protocols Definition,” *MultiChannel Adaptive Information Systems: Tech Report 2.2.2*, [http://www.mais-project.it/documenti\\_pubblico/IIIIssemester/r2.2.2.pdf](http://www.mais-project.it/documenti_pubblico/IIIIssemester/r2.2.2.pdf), November 4, 2004.
- [**Ballmer1981**] T. Ballmer and W. Brennenstuhl, “Speech Act Classification: A Study in the Lexical Analysis of English Speech Activity Verbs,” *Springer Series in Language and Communication*, Vol.8. Springer-Verlag, Berlin-Heidelberg-New York, 1981.
- [**Bagrodia2003**] R. Bagrodia, S. Bhattacharyya, F. Cheng, S. Gerding, G. Glazer, R. Guy, Z. Ji, J. Lin, T. Phan, E. Skow, M. Varshney, and G. Zorpas, “iMASH: Interactive Mobile Application Session Handoff,” *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, May 2003.
- [**Bertino2001**] E. Bertino, S. Castano, and E. Ferrari, “On Specifying Security Policies for Web Documents with an XML-Based Language,” *Proceedings of the 6<sup>th</sup> ACM Symposium on Access Control Models and Technologies (SACMAT 2001)*, pp. 57-65, Chantilly, VA, May 2001.
- [**Bhagwat1996**] P. Bhagwat, C. Perkins, and S. Tripathi, “Network Layer Mobility: An Architecture and Survey,” *IEEE Personal Communications*, Vol. 3, Issue 3, pp. 54-64, June 1996.

- [**Blaze1998**] M. Blaze, J. Feigenbaum, and M. Strauss, "Compliance Checking in the PolicyMaker TrustManagement System," *Proceedings of the Financial Cryptography Conference*, Lecture Notes in Computer Science (Springer), Vol. 1465, pp. 254-274, 1998.
- [**Blaze1999**] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The KeyNote Trust Management System Version 2," *RFC 2704*, September 1999.
- [**Bohrer2003**] K. Bohrer, S. Levy, X. Liu, and E. Schonberg, "Individual Privacy Policy Based Access Control," *Proceedings of the 6th International Conference on Electronic Commerce Research (ICECR 2003)*, 2003.
- [**Bonatti2000**] P. Bonatti and P. Samarati, "Regulating Service Access and Information Release on the Web," *Proceedings of the 7<sup>th</sup> ACM Conference on Computer and Communications Security*, Athens, November 2000.
- [**Bonatti2005**] P. A. Bonatti and D. Olmedilla, "Driving and Monitoring Provisional Trust Negotiation with Metapolicies," *Proceedings of the 6<sup>th</sup> IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pp. 14-23, Stockholm, Sweden, June 2005.
- [**Bonatti2006**] P. A. Bonatti, D. Olmedilla, and J. Peer, "Advanced Policy Explanations on the Web," *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, pp. 200-204, Riva del Garda, Italy, August-September 2006.
- [**Broens2004**] T. Broens, S. Pokraev M. V. Sinderen, J. Koolwaaij, and P. D. Costa, "Context-Aware, Ontology-Based, Service Discovery," *Proceedings of the 2<sup>nd</sup> European Symposium on Ambient Intelligence (EUSAI 2004)*, Springer, pp. 72-83, 2004.
- [**Brooks1997**] R. Brooks, "The Intelligent Room Project," *Proceedings of the 2<sup>nd</sup> International Cognitive Technology Conference*, Aizu, Japan, 1997.
- [**Cahill2003**] V. Cahill, E. Gray, J.-M. Seigneur, C. D. Jensen, Y. Chen, B. Shand, N. Dimmock, A. Twigg, J. Bacon, C. English, W. Wagealla, S. Terzis, P. Nixon, G. di Marzo Serugendo, C. Bryce, M. Carbone, K. Krukow, and M. Nielsen, "Using Trust for Secure Collaboration in Uncertain Environments," *IEEE Pervasive Computing*, Vol. 02, No. 3, pp. 52-61, July-September 2003.
- [**Campbell2002**] R. Campbell, J. Al-Muhtadi, P. Naldurg, G. Sampemane, and M. D. Mickunas, "Towards Security and Privacy for Pervasive Computing," *Proceedings of the International Symposium on Software Security*, Tokyo, Japan, 2002.

- [**Cautis2007**] B. Cautis, “Distributed Access Control: A Privacy-Conscious Approach,” *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*, Sophia Antipolis, France, June 20-22, 2007.
- [**Chakraborty2006**] D. Chakraborty, A. Joshi, Y. Yesha, and T. Finin, “Toward Distributed Service Discovery in Pervasive Computing Environments,” *IEEE Transactions on Mobile Computing*, Vol. 5, Issue 2, pp. 97-112, February 2006.
- [**Chang1994**] M. K. Chang and C. C. Woo, “A Speech-Act-Based Negotiation Protocol: Design, Implementation, and Test Use,” *Proceedings of the ACM Transactions on Information Systems (TOIS)*, Vol. 12, Issue 4, pp. 360-382, October 1994.
- [**Chao2002**] K.-M. Chao, R. Anane, J.-H. Chen, and R. Gatward, “Negotiating Agents in a Market-Oriented Grid,” *Proceedings of the 2<sup>nd</sup> IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, p. 436, 2002.
- [**Chen2004**] H. Chen, F. Perich, T. W. Finin, and A. Joshi, “SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications,” *Proceedings of the 1<sup>st</sup> Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2004)*, pp. 258-267, Boston, MA, August 2004.
- [**Chetan2004**] S. Chetan, J. Al-Muhtadi, R. Campbell and M. D. Mickunas, “A Middleware for Enabling Personal Ubiquitous Spaces,” *Proceedings of the System Support for Ubiquitous Computing Workshop (UbiSys 2004) at the 6<sup>th</sup> Annual Conference on Ubiquitous Computing (UBICOMP 2004)*, Nottingham, England, September 2004.
- [**Cholvy1997**] L. Cholvy and F. Cuppens, “Analyzing consistency of security policies,” *Proceedings of the 18<sup>th</sup> IEEE Computer Society Symposium on Research in Security and Privacy (RSP 1997)*, pp. 103-112, Oakland, CA, 1997.
- [**CiaoProlog**] “The Ciao Prolog Development System WWW Site,” <http://www.ciaohome.org/>.
- [**Cisco2003**] White Paper—“Cisco NAC: The Development of the Self-Defending Network,” [http://www.cisco.com/en/US/netsol/ns340/ns394/ns171/ns413/networking\\_solutions\\_white\\_paper09186a00801e0032.shtml](http://www.cisco.com/en/US/netsol/ns340/ns394/ns171/ns413/networking_solutions_white_paper09186a00801e0032.shtml).
- [**Coen1999**] M. Coen, B. Phillips, N. Warshawsky, L. Weisman, S. Peters, and P. Finin, “Meeting the Computational Needs of Intelligent Environments: The Metagluue System,” *Proceedings of the 1<sup>st</sup> International Workshop on Managing Interactions in Smart Environments (MANSE 1999)*, pp. 201-212. Dublin, Ireland, December 1999.

- [Coi2007] J. L. De Coi and D. Olmedilla, "A flexible policy-driven trust negotiation model," *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pp. 450-453, Silicon Valley, CA, November 2007.
- [Covington2000] M. J. Covington, M. J. Moyer, and M. Ahamad, "Generalized Role-Based Access Control for Securing Future Applications," *Technical Report GIT-CC-00-02*, Georgia Institute of Technology, College of Computing, February 1, 2000.
- [Czajkowski2002] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," *Proceedings of the 8<sup>th</sup> Workshop on Job Scheduling Strategies for Parallel Processing*, July 2002.
- [Damianou2001] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," *Proceedings of the 2<sup>nd</sup> IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, Bristol, U.K., January 2001.
- [DAML] "The Darpa Agent Markup Language Homepage," <http://www.daml.org>.
- [Dang2006] J. Dang and M. N. Huhns, "Concurrent Multiple-Issue Negotiation for Internet-Based Services," *IEEE Internet Computing*, Vol. 10, No. 6, pp. 42-49, November/December 2006.
- [Doyle1979] J. Doyle, "A Truth Maintenance System," *AI*, Vol. 12, No 3, pp. 251-272, 1979.
- [Dumbill2002] E. Dumbill, "XML Watch: Finding friends with XML and RDF," *IBM Developer Works*, <http://www-106.ibm.com/developerworks/xml/library/x-foaf.html>, June 2002.
- [English2002] C. English, P. Nixon, S. Terzis, A. McGettrick, and H. Lowe, "Dynamic Trust Models for Ubiquitous Computing Environments," *Proceedings of the Workshop on Security in Ubiquitous Computing (in conjunction with Ubicomp 2002)*, 2002.
- [English2003] C. English, W. Wagealla, S. Terzis, H. Lowe, A. McGettrick, and P. Nixon, "Trust Dynamics for Collaborative Global Computing," *Proceedings of the IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)*, June 2003.
- [English2004] C. English, S. Terzis, and P. Nixon, "Towards Self-Protecting Ubiquitous Systems: Monitoring Trust-based Interactions," *Journal of Personal and Ubiquitous Computing*, Vol. 10, Issue 1, pp. 50-54, December 2005.

- [**Ferraiolo1995**] D. Ferraiolo, J. Cugini, and D. R. Kuhn, "Role Based Access Control (RBAC): Features and Motivations," *Proceedings of the 1995 Computer Security Applications Conference*, pp. 241-248, December 1995.
- [**Eustice2003a**] K. Eustice, L. Kleinrock, S. Markstrum, G. Popek, V. Ramakrishna, and P. Reiher, "Enabling Secure Ubiquitous Interactions," *Proceedings of the 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing (in conjunction with Middleware 2003)*, Rio de Janeiro, Brazil, 17 June 2003.
- [**Eustice2003b**] K. Eustice, L. Kleinrock, S. Markstrum, G. Popek, V. Ramakrishna, and P. Reiher, "Securing WiFi Nomads: The Case for Quarantine, Examination, and Decontamination," *Proceedings of the New Security Paradigms Workshop (NSPW 2003)*, Ascona, Switzerland, August 2003.
- [**Eustice2007**] K. Eustice, V. Ramakrishna, A. Walker, M. Schnaider, N. Nguyen, and P. Reiher, "nanosphere: Location-Driven Fiction for Groups of Users," *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI 2007)*, Beijing, P.R.China, 22-27 July 2007.
- [**Eustice2008a**] K. Eustice, V. Ramakrishna, N. Nguyen, and P. Reiher, "The Smart Party: A Personalized Location-aware Multimedia Experience," *Proceedings of the Fifth IEEE Consumer Communications and Networking Conference (CCNC 2008)*, Las Vegas, NV, January 10-12, 2008.
- [**Eustice2008b**] K. F. Eustice, "Panoply: Active Middleware for Managing Ubiquitous Computing Interactions," *PhD Thesis*, Computer Science Department, UCLA, April 2008.
- [**Freudenthal2002**] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti, "dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments," *Proceedings of the 22<sup>nd</sup> International Conference on Distributed Computing Systems (ICDCS 2002)*, IEEE Computer Society, July 2002.
- [**Gavriloaie2004**] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett, "No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web," *Proceedings of the 1<sup>st</sup> First European Semantic Web Symposium*, Heraklion, Greece, May 2004.
- [**Google2005**] "Google proposes free Wi-Fi for San Francisco," [http://news.yahoo.com/s/nm/20051001/wr\\_nm/google\\_wifi\\_dc](http://news.yahoo.com/s/nm/20051001/wr_nm/google_wifi_dc).
- [**GRID**] "Grid Computing Info Centre (GRID Infoware)," <http://www.gridcomputing.com/>.

- [**Grimm2004a**] R. Grimm, "One.world: Experiences with a Pervasive Computing Architecture," *IEEE Pervasive Computing*, Vol. 3, Issue 3, pp. 22-30, July-September 2004.
- [**Grimm2004b**] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, "System Support for Pervasive Applications," *ACM Transactions on Computer Systems*, Vol. 22, Issue 4, pp. 421-486, November 2004.
- [**Guttman2001**] E. Guttman, "Autoconfiguration for IP Networking: Enabling Local Communication," *IEEE Internet Computing*, Vol. 5, No. 3, pp. 81-86, May 2001.
- [**Han1991**] J. L. Han, "Preventing infinite looping in Prolog," *Proceedings of IEEE International Conference on Tools for Artificial Intelligence*, pp. 524-525, San Jose, CA, November 1991.
- [**Hengartner2003**] U. Hengartner and P. Steenkiste, "Access Control to Information in Pervasive Computing Environments," *Proceedings of the 9<sup>th</sup> Workshop on Hot Topics in Operating Systems (HotOS 2003)*, pp. 157-162, Lihue, HI, May 2003.
- [**Hengartner2005**] U. Hengartner and P. Steenkiste, "Exploiting Information Relationships for Access Control," *Proceedings of the 3<sup>rd</sup> IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pp. 269-278, Kauai Island, HI, March 2005.
- [**Herzberg2000**] A. Herzberg, Y. Mass, L. Mihaeli, D. Naor, and Y. Ravid, "Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers," *Proceedings of the Symposium on Security and Privacy*, pp. 2-14, 2000.
- [**Herzog2007**] A. Herzog and N. Shahmehri, "Usable Set-up of Runtime Security Policies," *Proceedings of the International Symposium on Human Aspects of Information Security and Assurance*, pp. 99-113, July 2007.
- [**Hudert2006**] S. Hudert, H. Ludwig, and G. Wirtz, "A Negotiation Protocol Framework for WS-Agreement," *IBM Research Report RC24094 (W0610-165)*, October 31, 2006.
- [**Huhns2005**] M. N. Huhns and M. P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, Vol. 9, No. 1, pp. 75-81, January/February, 2005.
- [**IEEE802.16**] "IEEE 802.16 Backgrounder",  
<http://grouper.ieee.org/groups/802/16/pub/backgrounder.html>.

- [**Jajodia1997**] S. Jajodia, P. Samarati, V. S. Subrahmanian, "A Logical Language for Expressing Authorizations," *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 4-7, 1997.
- [**Jiang2005a**] H. Jiang and M. N. Huhns, "An Approach to Broaden the Semantic Coverage of ACL Speech Acts," *J. Akoka et al. (Editors) ER Workshops 2005*, pp. 162-171, 2005.
- [**Jiang2005b**] Z. Jiang, K. Lee, S. Kim, H. Bae, S. Kim, and S. Kang, "Design of a Security Management Middleware in Ubiquitous Computing Environments," *Proceedings of the 6<sup>th</sup> International Conference on Parallel and Distributed Computing Applications and Technologies*, pp. 306-308, 2005.
- [**Jøsang1999**] A. Jøsang, "Trust-Based Decision Making for Electronic Transactions," *Proceedings of the 4<sup>th</sup> Nordic Workshop on Secure IT Systems (NORDSEC 1999)*, Stockholm University Report, pp. 99-105, Stockholm, Sweden, 1999.
- [**Johnston2003**] J. Johnston, J. H. P. Eloff, and L. Labuschagne, "Security and Human Computer Interfaces," *Computers & Security*, Vol. 22, No. 8, pp. 675-684, December 2003.
- [**Kagal2001a**] L. Kagal, V. Korolev, H. Chen, A. Joshi, and T. Finin, "Centaurus: A Framework for Intelligent Services in a Mobile Environment," *Proceedings of the 21<sup>st</sup> International Conference on Distributed Computing Systems Workshops (ICDCS 2001)*, Mesa, Arizona, April 16-19, 2001.
- [**Kagal2001b**] L. Kagal, T. Finin, and A. Joshi, "Moving from Security to Distributed Trust in Ubiquitous Computing Environments," *IEEE Computer*, December 2001.
- [**Kagal2002**] L. Kagal, J. Undercoffer, F. Perich, A. Joshi, and T. Finin, "A Security Architecture Based on Trust Management for Pervasive Computing Systems," *Proceedings of the Grace Hopper Celebration of Women in Computing*, 2002.
- [**Kagal2003a**] L. Kagal, T. Finin, and A. Joshi, "A Policy Language for a Pervasive Computing Environment," *Proceedings of the IEEE 4<sup>th</sup> International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003.
- [**Kagal2003b**] L. Kagal, T. Finin, and A. Joshi, "A Policy Based Approach to Security for the Semantic Web," *Proceedings of the 2<sup>nd</sup> International Semantic Web Conference (ISWC 2003)*, September 2003.



- [**Kaminsky2005**] D. Kaminsky, "An Introduction to Policy for Autonomic Computing," <http://www.ibm.com/developerworks/autonomic/library/ac-policy.html>, March 22, 2005.
- [**Kapadia2007**] A. Kapadia, T. Henderson, J. Fielding, and D. Kotz, "Virtual Walls: Protecting Digital Privacy in Pervasive Environments," *Proceedings of the 5<sup>th</sup> International Conference on Pervasive Computing (Pervasive 2007)*, Toronto, Ontario, Canada, May 13-16, 2007.
- [**Kennes2003**] J. Kennes and A. Schiff, "The Value of a Reputation System," *Industrial Organization 0301011, EconWPA*, 2003.
- [**Kidd1999**] C. D. Kidd, R. J. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. Mynatt, T. E. Starner, and W. Newstetter, "The Aware Home: A Living Laboratory for Ubiquitous Computing Research," *Proceedings of the 2<sup>nd</sup> International Workshop on Cooperative Buildings (CoBuild 1999)*, October 1999.
- [**Kindberg2002**] T. Kindberg and A. Fox, "System Software for Ubiquitous Computing," *IEEE Pervasive Computing*, Vol. 1, No. 1, pp. 70-81, January-March 2002.
- [**Koch2005**] F. Koch, J.-J. C. Meyer, F. Dignum, and I. Rahwan, "Programming Deliberative Agents for Mobile Services: the 3APL-M Platform," *Proceedings of the 3<sup>rd</sup> International Workshop on Programming Multi-Agent Systems (ProMAS 2005)*, LNCS Vol. 3862, pp. 222-235, Utrecht, The Netherlands, July 26, 2005.
- [**Kolari2005**] P. Kolari, L. Ding, S. Ganjugunte, A. Joshi, T. W. Finin, and L. Kagal, "Enhancing Web Privacy Protection through Declarative Policies," *Proceedings of the 6<sup>th</sup> IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*, pp. 57-66, Stockholm, Sweden, June 2005.
- [**Kottahachchi2004**] B. Kottahachchi and R. Laddaga, "Building Access Controls for Intelligent Environments," *Proceedings of the 4<sup>th</sup> Annual International Conference on Intelligent Systems Design and Applications (ISDA 2004)*, Budapest, Hungary, August 2004.
- [**Kulkarni2008**] D. Kulkarni and A. Tripathi, "Context-Aware Role-Based Access Control in Pervasive Computing Systems," *Proceedings of the 13<sup>th</sup> ACM Symposium on Access Control Models and Technologies (SACMAT 2008)*, pp. 113-122, Estes Park, CO, 2008.
- [**Kumar2007**] V. Kumar, B. F. Cooper, G. Eisenhauer, and K. Schwan, "iManage: Policy-Driven Self-management for Enterprise-Scale Systems," *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, Newport Beach, CA, November 2007.

- [**Lawley2003**] R. Lawley, K. Decker, M. Luck, T. R. Payne, and L. Moreau, “Automated Negotiation for Grid Notification Services,” *Euro-Par 2003*, pp. 384-393, 2003.
- [**Leithead2004**] T. Leithead, W. Nejdl, D. Olmedilla, K. E. Seamons, M. Winslett, T. Yu, and C. C. Zhang, “How to Exploit Ontologies for Trust Negotiation,” *Proceedings of the ISWC Workshop on Trust, Security, and Reputation on the Semantic Web*, Vol. 127, Hiroshima, Japan, November 2004.
- [**Lobo1999**] J. Lobo, R. Bhatia, and S. Naqvi, “A Policy Description Language,” *Proceedings of the AAAI*, pp. 291-298, , Orlando, FL, July 18-22, 1999.
- [**Lorch2003**] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, “First Experiences Using XACML for Access Control in Distributed Systems,” *Proceedings of the 2003 ACM Workshop on XML Security*, Fairfax, VA, October 2003.
- [**Lupu1999**] C. Lupu and M. Sloman, “Conflicts in Policy-Based Distributed Systems Management,” *IEEE Transactions on Software Engineering*, Vol. 25, Issue 6, pp. 852-869, 1999.
- [**Marriott1989**] K. Marriott and H. Sondergaard, “On Prolog and the Occur Check Problem,” *ACM SIGPLAN Notices*, Vol. 25, Issue 5, pp. 76-82, May 1989.
- [**Micali2003**] S. Micali, “Simple and fast optimistic protocols for fair electronic exchange,” *Proceedings of the 22<sup>nd</sup> Annual Symposium on Principles of Distributed Computing (PODC 2003)*, pp.12-19, Boston, Massachusetts, July 13-16, 2003.
- [**Minami2005**] K. Minami and D. Kotz, “Secure Context-sensitive Authorization,” *Journal of Pervasive and Mobile Computing (PMC)*, Vol. 1, Issue 1, March 2005.
- [**Minami2006**] K. Minami and D. Kotz, “Scalability in a Secure Distributed Proof System,” *Proceedings of the 4<sup>th</sup> International Conference on Pervasive Computing (Pervasive 2006)*, Dublin, Ireland, May 2006.
- [**MinervaProlog**] “Minerva,” <http://www.ifcomputer.com/MINERVA/>.
- [**Minsky2000**] N. H. Minsky and V. Ungureanu, “Law-governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 9, No. 3, pp.273-305, July 2000.
- [**MIT-Oxygen**] “MIT Project Oxygen: Overview”, <http://www.oxygen.lcs.mit.edu/Overview.html>.

- [**Molloy2008**] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, "Mining Roles with Semantic Meanings," *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT 2008)*, pp. 21-30, June 2008.
- [**Necula1997**] G. Necula, "Proof-Carrying Code," *Proceedings of the 24<sup>th</sup> Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, January 1997.
- [**Nejdl2004**] W. Nejdl, D. Olmedilla, and M. Winslett, "PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web," *Secure Data Management*, pp. 118-132, 2004.
- [**OSGI**] Open Service Gateway Initiative (OSGi), <http://www.osgi.org>.
- [**OWL**] "OWL Web Ontology Language Overview," <http://www.w3.org/TR/owl-features/>.
- [**P3P**] "P3P Public Overview," <http://www.w3.org/P3P/>.
- [**Papamarkos2003**] G. Papamarkos, A. Poulouvasilis, and P. T. Wood, "Event-Condition-Action Rule Languages for the Semantic Web," *Proceedings of the Workshop on Semantic Web and Databases, at VLDB 2003*, Berlin, Germany, September 2003.
- [**Papamarkos2004**] G. Papamarkos, A. Poulouvasilis, and P. T. Wood, "Event-Condition-Action Rules on RDF Metadata in P2P Environments," *Proceedings of the 2<sup>nd</sup> Workshop on Metadata Management in Grid and P2P Systems (MMGPS): Models, Services and Architectures*, Senate House, University of London, December 17, 2004.
- [**Parkin2006**] M. Parkin, D. Kuo, and J. Brooke, "A Framework & Negotiation Protocol for Service Contracts," *Proceedings of the IEEE International Conference on Services Computing (SCC 2006)*, pp. 253-256, 2006.
- [**Pasquier2007**] P. Pasquier, L. Sonenberg, I. Rahwan, F. Dignum, and R. Hollands, "An Empirical Study of Interest-based Negotiation," *Proceedings of 9<sup>th</sup> International Conference on Electronic Commerce (ICEC)*, Minneapolis, MN, pp. 339-348, August 2007.
- [**Paurobally2007**] S. Paurobally, V. Tamma and M. Wooldridge, "A Framework for Web Service Negotiation," *Proceedings of the ACM Transactions on Autonomous and Adaptive Systems (TAAS 2007)*, Vol. 2, Issue 4, Article No. 14, November 2007.

- [**Peters2003**] S. Peters, G. Look, K. Quigley, H. Shrobe and K. Gajos, “Hyperglue: Designing High-Level Agent Communication for Distributed Applications,” *Originally submitted to AAMAS 2003*.
- [**RDF**] “Resource Description Framework (RDF) / Semantic Web Activity,” <http://www.w3.org/RDF/>.
- [**Rahwan2005**] I. Rahwan, F. Koch, C. Graham, A. Kattan, and L. Sonenberg, “Goal-directed Automated Negotiation for Supporting Mobile User Coordination,” *Proceedings of the 5<sup>th</sup> International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT 2005)*, pp. 382-395, Paris, France, July 2005.
- [**Ramakrishna2007**] V. Ramakrishna, Kevin Eustice, and Peter Reiher, “Negotiating Agreements Using Policies in Ubiquitous Computing Scenarios,” *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA’07)*, Newport Beach, California, June 19-20, 2007.
- [**RFC2693**] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, “SPKI Certificate Theory,” *RFC 2693*, September 1999.
- [**Román2002**] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt, “Gaia: A Middleware Infrastructure to Enable Active Spaces,” *IEEE Pervasive Computing*, pp. 74-83, October-December 2002.
- [**Roy1990**] P. Van Roy, “Can Logic Programming Execute as Fast as Imperative Programming?,” *PhD thesis*, University of California, Berkeley, November 1990.
- [**Sampemane2002**] G. Sampemane, P. Naldurg, R. H. Campbell, “Access Control for Active Spaces,” *Proceedings of ACSAC 2002*, pp. 343-352, 2002.
- [**Seamons2002**] K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu, “Requirements for Policy Languages for Trust Negotiation,” *Proceedings of the 3<sup>rd</sup> International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, Monterey, CA, June 2002.
- [**Searle1981**] J. R. Searle and D. Vanderveken, “Foundations of Illocutionary Logic,” *Cambridge University Press*, Cambridge, UK, 1984.
- [**SemWeb**] “W3C Semantic Web home page,” <http://www.w3.org/2001/sw/>.
- [**Sen2002**] S. Sen and N. Sajja, “Robustness of Reputation-Based Trust: Boolean Case,” *Proceedings of the 1<sup>st</sup> International Joint Conference on Autonomous Agents and Multiagent Systems*, Bologna, Italy, July 15-19, 2002.

- [**Shankar2002**] N. Shankar and W. A. Arbaugh, “On Trust for Ubiquitous Computing,” *Invited paper in the Workshop on Security for Ubiquitous Computing (in conjunction with UBICOMP 2002)*, October 2002.
- [**Shankar2005a**] C. Shankar, A. Ranganathan, and R. Campbell, “An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments,” *Proceedings of the 2<sup>nd</sup> Annual International Conference on Mobile and Ubiquitous Systems (Mobiquitous 2005)*, San Diego, California, July 2005.
- [**Shankar2005b**] C. Shankar and R. Campbell, “A Policy-based Management Framework for Pervasive Systems using Axiomatized Rule Actions,” *Proceedings of the 4<sup>th</sup> IEEE International Symposium on Network Computing and Applications (IEEE NCA 2005)*, Cambridge, MA, July 2005.
- [**Shneiderman2004**] B. Shneiderman and C. Plaisant, “Designing the User Interface,” *Addison Wesley*, 4<sup>th</sup> edition, 2004.
- [**Singh2008**] J. Singh, L. Vargas, J. Bacon, and K. Moody, “Policy-Based Information Sharing in Publish/Subscribe Middleware,” *Proceedings of the IEEE Workshop on Policies for Distributed Systems and Networks*, pp. 137-144, 2008.
- [**Sloman2002**] M. Sloman and E. Lupu, “Security and Management Policy Specification,” *IEEE Network*, March/April 2002.
- [**SWIProlog**] “SWI-Prolog’s Home,” <http://www.swi-prolog.org>.
- [**Toninelli2006**] A. Toninelli, R. Montanari, L. Kagal, and O. Lassila, “A Semantic Context-Aware Access Control Framework for Secure Collaborations in Pervasive Computing Environments,” *Proceedings of the 5<sup>th</sup> International Semantic Web Conference*, Athens, GA, November 5-9, 2006.
- [**Tsai2007**] W. T. Tsai, Q. Huang, J. Xu, Y. Chen, and R. Paul, “Ontology-Based Dynamic Process Collaboration in Service-Oriented Architecture,” *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA’07)*, Newport Beach, California, June 19-20, 2007.
- [**TuProlog**] “Home – tuProlog,” <http://alice.unibo.it/xwiki/bin/view/Tuprolog/>.
- [**Undercoffer2003**] J. Undercoffer, F. Perich, A. Cedilnik, L. Kagal, and A. Joshi, “A Secure Infrastructure for Service Discovery and Access in Pervasive Computing,” *Mobile Networks and Applications, Special Issue on Security in Mobile Computing Environments*, Vol. 8, Issue 2, pp. 113-125, April 2003.
- [**UPnP**] “UPnP Forum,” <http://www.upnp.org>.

- [**Uszok2004**] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "KAoS Policy Management for Semantic Web Services," *IEEE Intelligent Systems*, Vol. 19, No. 4, pp. 32-41, July 2004.
- [**Waldo1999**] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM*, Vol. 42, No. 7, pp. 76-82, 1999.
- [**Weiser1991**] M. Weiser, "The Computer for the 21<sup>st</sup> Century," *Scientific American*, Vol. 265, Issue 30, pp. 94-104, 1991.
- [**Winsborough2000**] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated Trust Negotiation," *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, Hilton Head, SC, January 2000.
- [**Winslett2002**] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu, "Negotiating Trust on the Web," *IEEE Internet Computing*, November/December 2002.
- [**Winslett2003**] M. Winslett, "An Introduction to Trust Negotiation," *Proceedings of the 1<sup>st</sup> International Conference on Trust Management*, Crete, Greece, May 2003.
- [**WS\_Policy**] S. Bajaj et al., "Web Services Policy Framework (WS-Policy)," *Specification* *Version* 1.2, <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf>, March 2006.
- [**Xiong2004**] L. Xiong and L. Liu, "PeerTrust: Supporting Reputation-Based Trust in Peer-to-Peer Electronic Communities," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, *Special Issue on Peer-to-Peer Based Data Management*, 2004.
- [**Xu2005**] W. Jiang, Y. Xu, D. Hao, and S. Zhen, "Research on Multi-agent System Automated Negotiation Theory and Model," *Proceedings of Network and Parallel Computing, IFIP International Conference (NPC 2005)*, Beijing, China, November 30-December 3, 2005.
- [**Zartman1993**] I. W. Zartman, "Pre-Negotiation Phases and Functions," In *Janice Stein, Getting to the Table*, The Johns Hopkins United Press, 1993.
- [**Zartman1988**] I. W. Zartman, "Common Elements in the Analysis of the Negotiation Process," *Negotiation Journal*, Vol. 4, No. 1, pp. 31-43, January 1988.

- [**Zhu2005a**] F. Zhu, W. Zhu, M. W. Mutka, and L. M. Ni, "Expose or Not? A Progressive Exposure Approach for Service Discovery in Pervasive Computing Environments," *Proceedings of the 3<sup>rd</sup> IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pp. 225-234, Kauai Island, HI, March 2005.
- [**Zhu2005b**] F. Zhu, M. W. Mutka, and L. M. Ni, "Service Discovery in Pervasive Computing Environments," *IEEE Pervasive Computing*, Vol. 4, Issue 4, pp. 81-90, October 2005.
- [**Zhang2006**] L. Zhang, A. Brodsky, and S. Jajodia, "Toward Information Sharing: Benefit And Risk Access Control (BARAC)," *Proceedings of the 7<sup>th</sup> IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006)*, pp. 45-53, London, Ontario, Canada, June 2006.
- [**Zhang2007**] D. Zhang, B. Lim, M. Zhu, and S. Zheng, "Supporting Impromptu Service Discovery and Access in Heterogeneous Assistive Environments," *Proceedings of ICOST 2007*.