Journal: Computer Networks

Title: *Securing Distributed Adaptation*

Keywords: open architecture, distributed adaptation, security

Abstract:

Open architecture networks provide applications with fine-grained control over network elements. With this control comes the risk of misuse and new challenges to security beyond those present in conventional networks. One particular security requirement is the ability of applications to protect the secrecy and integrity of transmitted data while still allowing trusted active elements within the network to operate on that data.

This paper describes mechanisms for identifying trusted nodes within a network and securely deploying adaptation instructions to those nodes while preventing unauthorized access and modification of application data. Promising experimental results of our implementation within the Conductor adaptation framework will also be presented, suggesting that such features can be incorporated into real networks.

Authors:

Jun Li, Mark Yarvis, Peter Reiher
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA

Point of Contact:

Jun Li
Email: lijun@cs.ucla.edu
Phone: 310-825-8524, 818-908-8940
Fax: 310-825-7578

Contact Mailing Address:

Jun Li
14419 Califa St, Apt 1
Van Nuys, CA 91401, USA

Securing Distributed Adaptation[†]

Jun Li Mark Yarvis Peter Reiher

Computer Science Department

University of California, Los Angeles

Los Angeles, CA 90095, USA

Abstract

Open architecture networks provide applications with fine-grained control over network elements. With this control comes the risk of misuse and new challenges to security beyond those present in conventional networks. One particular security requirement is the ability of applications to protect the secrecy and integrity of transmitted data while still allowing trusted active elements within the network to operate on that data.

This paper describes mechanisms for identifying trusted nodes within a network and securely deploying adaptation instructions to those nodes while preventing unauthorized access and modification of application data. Promising experimental results of our implementation within the *Conductor* adaptation framework will also be presented, suggesting that such features can be incorporated into real networks.

Keywords: **open architecture, distributed adaptation, security**

1. Introduction

As computer networks become more heterogeneous, applications must increasingly deal with suboptimal network conditions. Applications can use open network architectures to provide service tailored to network conditions, adapting the protocols used and perhaps altering the actual data sent. Services such as Protocol Boosters [8] and Panda [23] allow adaptation to occur at nodes within the network. Unfortunately, this added flexibility and control

[†] This work was partially supported by DARPA under contract number DABT63-94-C-0080. Authors can be contacted at {lijun, yarvis, reiher}@cs.ucla.edu.

could be used by attackers to damage or destroy communications, unless the open architecture is designed to prevent such misuse.

One security issue is protecting the open architecture elements from the user. However, protecting the secrecy and integrity of a user's data from network elements that might be untrustworthy is just as important. The existing solution to this problem is to encrypt the data end-to-end, but many useful adaptations, like removal of color from a video stream, require access to unencrypted data. Link-level encryption can protect the data stream while it is on the wire, but this approach allows any system on the end of the link unlimited access to the data, without any control by the user. A good solution should give the user the power to select which of those elements will be allowed to view or modify data in plaintext form.

Consider a home with many Internet-capable devices connected to a wireless LAN. A router connects that LAN to the Internet by way of a DSL link. A user on the wireless LAN wishes to obtain his bank balance over the web. Clearly this data should be encrypted, particularly for transmission over the Internet and the wireless LAN. At the same time, other users on the LAN are downloading software, also using web protocols. If short jobs were given priority, the interactive traffic would not be swamped by the bulk transmissions. Unfortunately, determining the expected length of the data stream requires access to the stream (since it's encoded in the Content-Length header field). Other possible adaptations, like reducing the resolution of images, would also require data access. While an active node provided by the ISP may be trusted to perform such adaptations, many of the other nodes on the path between the client and server need not be trusted.

One way to protect data from unauthorized modification within the network is through the use of a series of signatures [27]. By digitally signing a transmitted packet and re-signing subsequent versions of that packet, it is possible for the receiving application to determine the source of the data and any modifications to the data. While this approach detects unauthorized modifications to data packets, providing signatures on individual packets is expensive and does not provide secrecy. The common alternative of end-to-end encryption, mentioned earlier, provides the desired secrecy and data integrity. However, by ensuring access to only the endpoints of the connection, most useful adaptations are disallowed. Link-level encryption protects both integrity and secrecy across all network links, while allowing adaptation to occur on any node along the data path. However, every node in the

path is implicitly trusted. A node that is not trustworthy could easily siphon the data stream or alter it in an unauthorized manner. In addition, link-level encryption requires decryption and re-encryption at every node.

Virtual link encryption provides a compromise between end-to-end and link-level encryption. A trusted subset of network nodes is chosen, and encrypted data is transmitted between those nodes. The trusted nodes can arbitrarily adapt the unencrypted data. Decryption and re-encryption occur only where adaptation is desired, thereby reducing overhead.

Providing secure adaptation with the support of virtual link encryption requires that three activities be performed securely: selection of trusted nodes, selection and deployment of appropriate adaptive algorithms, and key distribution.

The endpoints of a connection can be implicitly trusted, since they already have full control over the data stream. Either users trust no other nodes in the network (in which case they should encrypt end-to-end), or they have some way to tell which nodes are trustworthy. In the latter case, authentication is required to prevent an untrustworthy node from masquerading as a trustworthy one. Since there is no ubiquitous infrastructure for authentication and because different applications may require different strengths of authentication, no single authentication mechanism will suffice. Instead, a pluggable authentication architecture is needed, allowing the user to determine an appropriate authentication mechanism for each stream. Some streams may require no authentication. Others may make use of an existing Kerberos or public key infrastructure. Because multiple authentication mechanisms are supported, the system must ensure that each node uses an acceptable mechanism to authenticate other nodes. The resulting chicken-and-egg problem of what mechanism to use to establish one or more acceptable authentication mechanisms must also be solved.

The decision of which adaptive algorithms to deploy and where to deploy them is based on information such as link characteristics, user preferences, and available node resources. Attackers could force unnecessary or even undesirable adaptations by falsifying information about conditions, or they could illicitly alter a good plan while it was being distributed to the trusted nodes. The process of gathering this information, analyzing it, and distributing the result must be protected. Thus, source information and resulting instructions must be authenticated, ensuring origination at a trusted node, and analysis must occur on a trusted node.

Finally, before any user data can flow, session keys must be securely distributed to those trusted nodes on which adaptation will be performed. These session keys provide a shared secret, allowing data to be encrypted for transmission between each pair of adjacent trusted nodes, the two endpoints of a virtual link. Untrusted nodes will see only encrypted data.

This paper will describe an implementation of virtual link encryption to protect the Conductor distributed adaptation service. The implementation includes an extensible authentication service with several sample authentication modules, a secured mechanism for selecting adaptations, and a facility for secure key distribution. We provide measurements of the overheads involved in connection setup, demonstrating the usability of this approach. Finally, we describe several extensions to the basic mechanism that allow broader applicability at the cost of somewhat higher overhead.

2. Conductor—A Distributed Adaptation Service

We built the Conductor adaptation service to demonstrate the value of distributed deployment of adaptive agents into a network. The portion of the Conductor design relevant to security is described below. Additional detail and performance results can be found in [28].

Conductor enables distributed adaptation by providing an adaptation framework at various nodes throughout the network. Conductor consists of essentially two parts: adaptors that operate on a data stream and a runtime environment that supports adaptors. Adaptors have the ability to view and modify the data stream in transit. Adaptors are frequently paired, allowing the data stream to be converted to an easily transmitted format and then back to the original format. For instance, a pair of adaptors might compress and then decompress a data stream for transmission across a low-bandwidth link, or encrypt and then decrypt a data stream for transmission across insecure links or nodes. Adaptations can be combined as needed to satisfy multiple user requirements.

The Conductor runtime environment is meant to be deployed on various nodes throughout the network to provide points of adaptation. A given data stream is intercepted by Conductor and routed through the Conductor-enabled nodes between the client and server. The framework is responsible for monitoring network and node conditions, routing the data stream, determining which adaptors to deploy for a particular data stream, inserting the selected adaptors into a data stream, and providing any resources required by an adaptor.

Each node that a data flow passes through may adapt the data based purely on local conditions, but such an ad hoc adaptation may not be appropriate. For instance, a pair of compression and decompression adaptors may be deployed around a low-bandwidth link, but if there is another low-bandwidth link upstream, end-to-end compression is better. Such compression, however, might impede other content-based adaptations. Adaptation planning is necessary to ensure a set of proper and compatible adaptations are applied at appropriate locations [24].

Conductor provides a planning infrastructure to determine which adaptors to deploy and where to deploy them (a planning process with four Conductor-enabled nodes is shown in Figure 1). When a new data connection is created, Conductor discovers a set of Conductor-enabled nodes along the path between the client and the server. These nodes are the potential adaptation points for this connection. Each of these nodes forwards its identity and planning-related information, such as local disk and CPU resources and network conditions, along the path toward one Conductor-enabled endpoint node. This endpoint, having received the planning information from every node, can now execute a planning algorithm and generate a plan. The plan, which describes a set of adaptors to deploy on each node, is then forwarded to each node along the path. Once the plan is delivered to all nodes, adaptors can be deployed, and data can begin to flow.

Of the Conductor-enabled nodes involved in a given connection, the endpoints have particular importance. The Conductor service on the client-side Conductor-enabled node is called the *initiator*, since the connection is initiated from this side. The final decision concerning which adaptations to employ at each node is made on the opposite server-side Conductor-enabled endpoint, which is known as the *planner*.

3. Design of Conductor Security

For a given connection, both the client and the server are initially assumed to be running on top of a Conductor-enabled node; thus the client is collocated with the initiator, and the server is collocated with the planner. In the discussions that follow, client and initiator are interchangeable, and so are server and planner. We will relax this assumption in Section 8.

Conductor provides an extensible architecture for securing both the planning process and the user's data. For a given connection, each Conductor-enabled node relies on a *security box* to authenticate itself to others or vice versa, protect planning messages, distribute keys for data stream secrecy, prevent replay attacks, etc. A variety of security

schemes are possible. Each security box implements a particular security scheme. Conductor provides a mechanism to ensure that the right security box is instantiated.

3.1 Security via a Security Box

3.1.1 Security box functionalities

A security box can be viewed as a security monitor that is responsible for node authentication, protection of the planning process, and session key distribution. A security box allows a node to authenticate other nodes or authenticate itself to another node. A security box protects planning by ensuring that only authentic planning information from authorized nodes can influence plan formulation, and only an authentic plan can be deployed. Finally, a security box can aid in data protection by enabling session key distribution. We will further discuss these functionalities in the following sections.

A security box can also be viewed as a message filter (Figure 2). All planning-related messages sent and received must pass through the security box. Incoming messages are accepted or rejected based on trust and authenticity. Outgoing messages are inspected, enhanced with additional authentication information, and perhaps encrypted.

Many security box implementations are possible, each providing a different level of node authentication, message verification, replay prevention, and possibly secrecy. The level of protection provided depends entirely on the particular security box implementation.

This architecture allows a user to choose a specific security scheme based on the desired level of protection. Flexibility is necessary because there is no ubiquitous authentication mechanism, nor is one level of trust appropriate for all situations.

We have constructed several security boxes based on public key cryptography. They will be discussed in detail in Section 4. Other cryptographic mechanisms can also be used to implement different security boxes.

3.1.2 Node authentication

Authentication is fundamental to Conductor security. Only trusted nodes can participate in planning and access the plaintext data stream.

To authenticate one Conductor-enabled node to another, a security box can include authentication information on behalf of the sender. When authentication information is received, the security box on the receiving node can invoke its authenticating functionality to determine whether a node is trusted.

While many security box implementations are possible, each security box may enforce a different authentication mechanism. Each authentication mechanism may have a different specification for what cryptographic algorithm to use and how it should be employed. Each node sends its own authentication information toward the planner. Nodes A, B, and C in Figure 3 will send their authentication information to planner D in the same way that planning information was sent. The planner can then authenticate the node. The planner sends its own authentication information in the reverse direction in the same manner as plan distribution, allowing every node to authenticate the planner.

3.1.3 Planning process protection

Each connection's planning process must be protected, including node selection at the planner node and plan deployment at other Conductor-enabled nodes. Each node provides authenticating information for planning information, typically a digital signature (Figure 3). The planner node selects those nodes it trusts, authenticates their incoming planning information, formulates a plan, and distributes the plan along the reverse path. The planner node also provides authenticating information for the plan. During plan distribution, each node verifies the authenticity of the incoming plan before it is instantiated. Planning messages can also be encrypted via the security box to provide secrecy.

Conductor supports a trust management mechanism. At the planner node, if a node is trusted to participate in the planning process according to the trust management mechanism, and its planning information is correctly authenticated, its planning information can be trusted and used in forming a plan. Similarly, if an intermediate node trusts the planner node according to the trust management mechanism, and can authenticate the plan, the plan can be accepted. This trust management system will be discussed in Section 3.4.

In the above discussion the planner node has full control of which nodes can be selected. The initiator can later reject a plan, but not otherwise influence node selection. This could be improved by assigning more control power to the initiator. For instance, after the planner node selects one or more nodes, it can negotiate with the initiator to

reach a final agreement on which nodes to finally select. However, the improvement would be achieved at the price of more coordination cost.

3.1.4 Data stream protection

If the data stream of a connection needs to be encrypted to protect the communication secrecy or integrity between the application client and the server (perhaps only when crossing a dangerous area), or to protect the data from unauthorized adaptation, the planner can select encryption and matching decryption adaptors to deploy at trusted nodes. The planner may have several encryption/decryption pairs to choose from based on the desired encryption strength. Each of these pairs of adaptors protects the data stream across one virtual link.

Session keys for data encryption and decryption can be generated on the planner node, which is implicitly trusted. Typically only one key is required per type of encryption/decryption adaptor deployed for each session. The planner needs to distribute the keys to those nodes where the keys are needed.

Node authentication is the fundamental basis for key distribution. Only trusted nodes should receive session keys, so the session key must be encrypted in a form that only the target can decrypt. Also, the receiver must be able to determine that the keys originated from a trusted distribution source, so the planner must provide authentication information for a session key in the same way it would for a plan (a digital signature).

The planner may trust some Conductor-enabled nodes to adapt plaintext data while others nodes may only be given access to encrypted text. In the first case, the planner will distribute a session key to the node along with an adaptation plan. In the second case, the planner may still distribute an adaptation plan but no session key.

3.2 Dynamic Selection of Security Schemes

Conductor allows multiple pluggable security schemes. Since there is no ubiquitous security scheme, and each connection may require a different level of protection, Conductor allows many security box implementations. This flexibility makes it easy to add a new security scheme with a new security box implementation. For one connection between an application client and a server, all involved Conductor-enabled nodes use one particular security scheme. For another connection, a different security scheme may be employed. Each Conductor-enabled node may get involved in more than one connection, and for each connection it can employ a different scheme.

Conductor ensures that all Conductor-enabled nodes involved in a connection use the same security scheme. At the beginning of a planning process, the user selects an appropriate security scheme (or one is selected on his behalf) at the initiator. A *security scheme selector message* is then forwarded toward the opposite end point, the planner node. This message tells which security scheme should be employed for this data connection. Each selector message can also include parameters specific to a particular security scheme, such as the names of the desired public key encryption algorithm, message digest algorithm, signature algorithm, and so forth. After receiving the selector message, each intermediate Conductor-enabled node will load the appropriate security box and forward the message to the next Conductor-enabled node on the path toward the planner. As a result, each Conductor-enabled node on the path, including the planner, will enforce the corresponding security scheme for this connection.

Furthermore, Conductor provides a mechanism to ensure that every node of a connection has indeed used the same security scheme throughout the planning process. Protection of scheme selection is done via the security box itself. When a security scheme selector message is forwarded toward the planner, it is unprotected. However, the planner node, as the last node to receive the selector message, sends back an indication of the security scheme that it has used. This time the information is protected (typically signed) by the security box (Figure 3). Each Conductor-enabled node, including the initiator, can securely determine whether the planner has used the expected security scheme. If the planner has used a different scheme (perhaps through subversion of the scheme selector during transmission), this will be caught by the initiator, if not earlier. If other nodes have used a different scheme, they will not be authenticated by the planner and will therefore not be selected in the plan.

When a connection crosses multiple domains, each of which supports different security mechanisms, it may not always be possible to select one common security scheme. We address this issue in Section 7.

3.3 Security Roles of the Initiator and the Planner

Conductor is careful in dividing tasks between the initiator and the planner. Because of their full access to the data stream, both the initiator and the planner of a connection are trusted. In principle, either of them can be responsible for the security scheme selection, session key generation, or a variety of other tasks. Or these two endpoints could negotiate for these tasks. However, since Conductor is frequently deployed where network conditions are poor, it attempts to minimize data transfer. Conductor also assumes as little prior coordination between nodes as possible.

Since the planning process starts at the initiator of a connection, it is most economical if the initiator is responsible for selecting security schemes. A security scheme selector message can be delivered to the planner node along the same route as the planning information. On the other hand, since the planner has authentication information for all nodes, it is in the best position to generate and distribute session keys.

3.4 Establishing Trust In Intermediate Nodes

After a planner authenticates an intermediate node, it must decide whether to authorize that node to adapt the data stream. Similarly, an intermediate node also needs to determine whether to trust the plan from a particular planner node. Here we focus on the former: that is, establishing a planner's trust of intermediate nodes.

Conductor can support a variety of authorization mechanisms. In a simple form, each Conductor-enabled node can keep a static list of nodes that it trusts. At the planner node, this list specifies those nodes that the planner trusts to adapt the data stream arbitrarily. Nodes not on the list are not trusted for any adaptation.

More flexible and dynamic models of trust can be enforced, typically by leveraging an automated trust management system. KeyNote is one such system [3]. It provides a mechanism to determine whether an action (described by an action attribute set) by a principal (typically expressed as the holder of a particular cryptography key) complies to a security policy (expressed by policy and credential assertions) by querying a general-purpose compliance checker.

KeyNote can be used to specify various trust relationships. For instance, some nodes are authorized by a planner to provide input to the planning process and to have full access to the data (and thus adapt plaintext data arbitrarily). Other nodes may be authorized to participate in the planning process but may not be allowed to see the plaintext data stream, instead adapting only encrypted data. Still other nodes may not be authorized even to provide input to the planning process. Various degrees of trust may also be possible. For instance, selective encryption of a layered encoding may be employed to allow partial access to (and adaptation of) plaintext data.

In order for Conductor to employ KeyNote, each Conductor-enabled node will have to enforce its specific security policy, describe those actions to check, and interact with the compliance checker. The security policy itself must be specified by the user. Several plausible mechanisms for designing policies are given below:

- The user specifies certain companies (ISPs, ASPs, content providers, etc.) that are trusted. Since any IP address is associated with a domain that is associated with a company, such a policy can be applied.

- The user assumes that bonded companies will have a set of Conductor nodes throughout the network, and the user will have a list of those companies.
- The user shares with his friends, lists of nodes that can be trusted.

3.5 Security Issues Not Addressed

We do not intend to address issues of denial-of-service in this work. If a Conductor-enabled node attempts to thwart the planning process by refusing to forward control information to the planner, the system will fail. However, this result is the same as a router refusing to forward data in any stream. This issue is, therefore, beyond the scope of this research. The safety of adaptor code is also not addressed. We intend to leverage existing research results on mobile code safety [2] [18] [26].

4. Authentication Schemes

Authentication is the basis of Conductor security. Our design allows security boxes with different authentication schemes to be plugged in. We have constructed three security boxes, *null*, *tree* and *chain*, each with a different authentication scheme. Different schemes provide different levels of protection, require different amounts of infrastructure (which may or may not be available), and have different levels of overhead. The *null* scheme does not provide any authentication. The other two schemes adopt authentication mechanisms based on public key cryptography, but with different assumptions on the structure of certificate authorities (*CA*s) and different methods for the collection and verification of public key certificates. We will further illustrate the ability to plug in symmetric cryptosystem-based authentication scheme using Kerberos.

4.1 Null Scheme

The *null* scheme provides no real authentication enforcement. It cannot be used when stream protection (and hence key distribution) is required. The *null* scheme is most useful for the case in which the user does not require security. In addition, having such a scheme can help demonstrate the added cost of the security architecture.

4.2 Authentication Using Public Key Cryptography

We have designed and implemented two authentication schemes, *tree* and *chain*, based on public key cryptography. Here, the authentication of a node is, in fact, the authentication of the public key of that node. The *tree* scheme

assumes that a certificate hierarchy infrastructure is available. The *chain* scheme assumes there is no certificate hierarchy; instead, CAs are distributed in a flat topology. In both the *tree* and *chain* schemes, each Conductor-enabled node has one associated CA (both schemes can be easily extended to allow each node to have multiple associated CAs, but in this paper we only discuss the single-CA case).

In a security box with either authentication scheme, planning information is authenticated using a digital signature based on public key cryptography. When a Conductor-enabled node provides its own planning information, it is signed with its own private key. When the planner node receives the planning information, it can check the authenticity of the planning information based on the signature, which in turn necessitates the authentication of the public key of that Conductor-enabled node. The authentication information for the public key of each Conductor-enabled node is included in an *authenticator* message.

Similarly, the authenticity of a plan is assured with the signature of the planner. When a node wants to install a distributed plan, it needs to ensure that the plan is authentic. The node checks the signature of the plan with the public key of the planner node. This operation requires the authentic public key of the planner node. The authentication information for the public key of the planner node is transmitted in a *reverse authenticator* message, which is similar to the authenticator.

When the public key of a node can be authenticated, a session key can be securely distributed to support data secrecy. Before a planner delivers a session key to a Conductor-enabled node, it can sign the session key with its own private key and encrypt with the authenticated public key of the node. Only the target recipient can decrypt the session key with its private key. The node can also verify that the session key is indeed from the planner after authenticating the public key of the planner.

Each different authentication scheme has its own protocol to generate authenticator and reverse authenticator messages and use them to do authentication and select trusted nodes.

4.2.1 Authentication scheme: tree

The *tree* scheme assumes a certificate hierarchy infrastructure is available. In this hierarchy, all CAs are organized in a tree structure, each at a particular level. The CA at the top (the parent) produces certificates for the next level

down (the child). This repeats recursively. The public key for the CA at the root of the tree (level 0) is universally known.

With such a structure, multiple certificates from the hierarchy may be required to authenticate a public key. The authenticator message (or the reverse authenticator message) sent by a Conductor-enabled node includes a list of all necessary certificates to verify the public key of that node. To build such a message, a node contacts its associated CA, *CA(n)*, for a certificate of the node's public key signed by *CA(n)*, *cert(node, CA(n))*. The certificate shows that *CA(n-1)* is the parent of *CA(n)*. The node then contacts *CA(n-1)* for a certificate of *CA(n)*'s public key signed by *CA(n-1)*. This repeats until a certificate signed by the root is returned.

Note that the set of certificates needed to certify a node's public key is static in this scheme. A node can therefore cache all of the certificates it will need to authenticate itself to any other node.

After the planner receives the authenticator message of a Conductor-enabled node, or a Conductor-enabled node receives the planner's reverse authenticator message, the list of certificates is retrieved from the message. Starting at the root, for which all nodes have a valid public key, lower-level CA certificates are authenticated recursively. Eventually, the certificate of the node in question is authenticated.

4.2.2 Authentication scheme: *chain*

4.2.2.1 Chain of trust

The deployment of a CA hierarchy is not required by the *chain* scheme. Instead, CAs are flatly distributed, as shown in Figure 4, possibly deployed independently by a variety of administrative authorities.

A CA typically provides certificates for the nodes in its "neighborhood," but may also contain a small number of "distant" nodes whose public keys are frequently queried.

We assume a certain degree of overlap between "neighboring" CAs. A CA may store the public keys for some "nearby" nodes and CAs.

This certification overlap can allow one node to authenticate to another by forming a chain of trust. As in other systems, a chain of trust is a chain of certificates, in which one end is the certificate for the public key of the node in

question, the other end is the certificate signed by the CA associated with the node running the authentication, and each certificate involved is verified.

4.2.2.2 Certificate collection

In the chain scheme, each node may add certificates useful in authenticating other nodes. When forwarding authentication information, each Conductor-enabled node asks its associated CA for every potentially useful certificate, and includes them in authenticator or reverse authenticator messages.

During the information-gathering portion of planning, each Conductor-enabled node along the path must authenticate itself to the planner. As demonstrated in Figure 5, the data stream from an application client to an application server is intercepted by four Conductor-enabled nodes, A, B, C and D. D is the planner for this connection. Each Conductor-enabled node (for example nodes A, B and C in Figure 5) initially generates a single-certificate authenticator. This authenticator contains a certificate for that node from its associated CA, the identity of the node, and the identity of the CA. This authenticator is then forwarded to the next node toward the planner. When an authenticator is received, each downstream Conductor-enabled node contacts its own associated CA to add two more certificates signed by this CA (if available): one certificate for the node specified in the authenticator and one for the CA specified in the authenticator. This node further forwards the authenticator message toward the planner node. Each authenticator, therefore, can be enhanced as it is forwarded toward the planner node.

While a Conductor-enabled node can ask for a certificate by contacting its associated CA, certificate caches can be deployed at Conductor-enabled nodes to improve performance. A negative certificate cache might also be employed; if a certificate is already known not to be contained in its associated CA, a node does not need to contact that CA.

The planner may receive multiple certificates in each authenticator message. In Figure 6, each square represents a certificate that may be finally available at the planner D of Figure 5. For instance, the authenticator for node A could include the certificates in the rows labeled "Node A" and "CA-A."

The same certificate collection principle is applied in the reverse direction. However, only a single reverse authenticator message flows along the reverse path toward the initiator (node A in Figure 7). So, in addition to

asking for certificates for the planner and the planner's associated CA, each Conductor-enabled node also asks for a certificate for every CA listed in the reverse authenticator; for example, *cert(CA-C, CA-B)* as shown in Figure 7.

4.2.2.3 Authentication

Authentication in the *chain* scheme requires a search for a valid chain of trust. Multiple chains are possible for a given node. Any valid chain to a node that includes only trusted CAs leads to a trusted public key. So, each possible chain must be checked until a trusted chain is discovered.

If the planner node (node D in Figure 5) receives a certificate for A signed by CA-D, since D knows the public key of CA-D, D can authenticate and obtain A's public key. This is a chain of trust composed of only one certificate, *cert(A, CA-D)*. However, if *cert(A, CA-D)* is not available, D will still try to verify A's public key by searching other chains of trust. For instance, if node D can get *cert(A, CA-A)* and *cert(CA-A, CA-D)*, a chain of trust (*cert(A,CA-A)*, *cert(CA-A, CA-D)*) is formed. D then can authenticate A's public key: CA-A's public key can be verified using *cert(CA-A, CA-D)* and CA-D's public key; CA-A's public key can then be used to verify *cert(A, CA-A)*.

The chain can be longer. The longest valid chain here would be *cert*(A, CA-A), *cert*(CA-A, CA-B), *cert*(CA-B, CA-C), *cert*(CA-C, CA-D). As long as there is a chain of trust in which CA-D is the last element, the public key certified by the first certificate of the chain can be verified; otherwise, the authentication fails.

Along the reverse direction, each Conductor-enabled node authenticates the planner in the same way. For instance, in Figure 7 at node B, planner D's public key can be verified if a chain of trust can be formed as (*cert*(D, CA-D), *cert*(CA-D, CA-C), *cert*(CA-C, CA-B)).

4.3 Authentication Using Kerberos

Integrating Kerberos [21] into Conductor as an authentication scheme is straightforward.

4.3.1 The Kerberos model

To authenticate to a service using Kerberos, an application obtains a ticket and then presents that ticket to the service for authentication. At an extremely high level, a client sends a request to Kerberos to authenticate to a particular service. The client receives (in the end) a session key for talking with the requested service, encrypted with a key it

shares with Kerberos, along with a ticket that it can send to the service. The ticket contains (among other things) the identity of the nodes involved and a session key for talking with the client encrypted with a secret that the service shares with Kerberos.

To authenticate to the service, the client sends an Kerberos authenticator (a time-stamp, a checksum, etc.) encrypted using the session key to the service along with the ticket. The service can obtain the session key using the key it shares with Kerberos and use it to decrypt the authenticator and therefore verify the authenticity of the client. The server can (optionally) send an authenticator back to the client, again encrypted with the session key, allowing the client to authenticate the server.

The session key used for authentication of the session can now be used by the client and server for whatever they like (typically encryption).

4.3.2 Integration with Conductor

In Conductor, each node along the path needs to authenticate itself to the planner node by sending an authentication message. Thus, each node needs to share a secret with Kerberos. At connection setup time, a given node will send a request to Kerberos to allow the node to authenticate with the planner node. Kerberos will provide a session key and a ticket for authenticating to that service.

The authentication message sent from a Conductor-enabled node to the planner will thus contain the Kerberos authenticator (encrypted with the session key) and ticket. The planner node will then be able to obtain the session key from the ticket and verify the authenticity of the sender's identity from the authenticator.

The planner' s reverse authentication message will consist of a Kerberos authenticator (encrypted using the session key). Again, the client will be able to use the session key it already has to verify the identity of the planner node.

Once established, the Kerberos session keys can also be used to digitally sign both the planning information and plan distribution messages. Key distribution can be accomplished by encrypting the Conductor session keys using the Kerberos session key. A signature is not required for key distribution since the Kerberos session key is known only to a given node and the planner.

Note that Kerberos requires a Conductor-enabled node to know the identity of the planner, and vice versa. This is possible because the planner is assumed to be on the same machine as the server of the connection in question. (In Section 8 we will relax this assumption.)

4.3.3 Cross-realm authentication

If a client and planner that wish to communicate are in different domains, they will likely have different Kerberos servers. The Kerberos infrastructure is already designed to handle this case. While the client may have to communicate with several Kerberos servers, it will eventually end up with a session key and ticket that have been generated by the remote endpoint's Kerberos server.

This is not particularly desirable, because a given Conductor node may have to go through several rounds with remote services in order to obtain the required ticket. Once obtained, however, no further communication with Kerberos is required.

4.4 Other Authentication Schemes

The *chain* scheme has similarities to PGP/X.509 where the chain of trust principle is also applied [11]; the *tree* scheme is similar to the PEM [14] authentication model, in which a CA hierarchy is also assumed.

Our design is open to other authentication models as well, and a new scheme can be easily plugged in. For instance, researchers at the University of California, Davis, proposed a solar trust model [5]. With this model, with respect to each specific CA (the sun), other CAs are ordered based on the trust degree (planets in orbit around the sun). Each CA has a rule set determining the trustworthiness of information signed by other CAs. Applying this model to our system, each authenticator would be formed in the same way as the *chain* scheme, but each certificate inside the authenticator would also have a rule set attached. To authenticate a public key, a node would need to apply the corresponding rule set for each involved certificate.

5. Attacks and Countermeasures

In this section we describe possible attacks and the countermeasures employed by Conductor. These attacks are independent of the security scheme selected, but the countermeasures and the effectiveness depend on specific mechanisms adopted by security boxes. We will show that the *tree* and *chain* schemes we developed are effective.

5.1 Node Impersonation

A node may attempt to impersonate another Conductor-enabled node in order to send a planner node fake planning information. A node may also impersonate the planner to distribute a fake plan or fake session keys. Recall that planning messages must pass through the security box at each Conductor-enabled node. The security box is responsible for preventing node impersonation.

The protection strength of the security box depends on the power of the adopted security scheme in the security box. The *null* scheme does not attempt to protect against node impersonation. In the *tree* or *chain* scheme, assuming the public key cryptography is not broken and CAs are not subverted, impersonation is not possible without knowing the private key of the node being impersonated. Upon receipt of a message, such an attack can be detected by obtaining the authentic public key of the sender and using the key to verify the signature of the planning messages from that node.

5.2 Key Stealing

The security box at each Conductor-enabled node aids session key distribution. In Conductor a session key is generated and distributed from the planner. The session key must be encrypted to ensure it is readable only by the intended recipient.

In the *tree* or *chain* scheme, when a session key is distributed to selected nodes, it is encrypted with each selected node's public key, which is already authenticated by the planner. Since the session key can only be decrypted with the node's private key, it cannot be stolen unless the private key of the node is stolen or unless node authentication is subverted and the planner uses the wrong public key to encrypt the session key.

5.3 Replay Attack

A Conductor-enabled node that has been selected in the past may execute a replay attack if it is not selected in the current planning process. Consider Figure 8 where both node B and C are selected, and the same session key **K1** is to be delivered to B and C. C receives encrypted session key **K1** that only C can decrypt. It also receives a second encrypted **K1** in a form such that only B can decrypt. Node C cannot decrypt the latter one and forwards it to node B.

Now consider a second connection as shown in Figure 9. This time, node C is not selected. It intercepts a new session key **K2** destined for B that only B can decrypt. Instead of forwarding **K2** to B, node C forwards the previous session key **K1** destined for B to B. C knows **K1**, and will be able to decrypt anything that B sends it. B will not be able to detect the problem when B receives **K1**.

This attack is prevented by associating a random number with each round of the planning process (Figure 10). The initiator injects a random number to each Conductor-enabled node. When a session key is distributed, the session key and the random number are encrypted together. Since in each planning process the random number is different, it is hard for C to provide B with an encrypted session key for the current round of planning.

Other replay attacks may also happen. The replay of a previous plan, for instance, occurs in a similar manner to the replay of session keys. We solve this problem using the method discussed above.

5.4 Security Scheme Replacement

After a security scheme is specified, a security scheme selector message is forwarded toward the planner in plain text. A corrupted node along the path could forge a different scheme and trick every node downstream into using that scheme. For instance, a *null* scheme may be substituted for the original *tree* or *chain* scheme.

The general principle of counteracting such an attack has been addressed in Section 3.2. Here we take a further look at how this is done in the *tree* or *chain* scheme. In the *chain* or *tree* scheme, the planner signs the scheme selector message, together with the ID of the current connection, and sends back the signature. Each Conductor-enabled node will verify the signature. If it is inconsistent with the original scheme, this will be detected at the initiator, if not sooner. In addition, a replay attack of the scheme selector signature cannot be successful since the ID of the current connection is unique, and it is signed together with the selector message.

6. Implementation and Experiments

The Conductor security architecture is fully implemented. We have also measured and analyzed the cost of using Conductor with different security schemes in terms of plan setup latency and bandwidth consumption.

6.1 Implementation

The implementation of Conductor security follows the design discussed above. We implemented the security box mechanism, and we also implemented the three pluggable security schemes, *null*, *tree* and *chain*. We used a static list to manage the trust relationships between Conductor-enabled nodes. The tools we used include the Java Cryptography Architecture [13] and the *cryptix* public domain encryption library 3.0.3 [6].

Additionally, we implemented a public key certificate authority (CA). A certificate client can send a request to a CA for the certificate of a node's public key. The CA in turn can return a certificate if one is available. We do not address certificate revocation.

6.2 Experiments

We measured the cost of providing Conductor security and the cost of different security schemes.

6.2.1 Experiment design

The security costs we consider include the latency to set up a plan and the bandwidth consumed during the plan setup procedure. Each time an application establishes a connection, a path of a certain number of Conductor-enabled nodes will be discovered. Our experiments measured how the security cost varies with the number of Conductor-enabled nodes (including the two endpoints).

Neither latency nor bandwidth consumption by the data stream was measured. The stream starts after the plan is deployed, and its cost is irrelevant to setting up the security scheme.

Four different scenarios were measured: *none, null, tree,* and *chain.* In the *none* scenario, each Conductor-enabled node along the path has no security implementation at all. None of the security mechanisms discussed in this paper are in place for the *none* scenario. In the *null* scenario, the entire generic security mechanism is in place, but no authentication is actually invoked for the connection. In the *null*, *tree*, and *chain* scenarios, each Conductor-enabled node along the path will enforce the selected scheme.

In the *tree* scenario, a three-level CA hierarchy was composed. The CA associated with each Conductor-enabled node was at the bottom level. Costs would vary with certificate hierarchies of different depths.

In the *chain* scenario, each Conductor-enabled node was associated with a different CA (Figure 11). Each CA could only certify 1) the node it was associated with and 2) the CAs associated with the Conductor-enabled nodes within the immediate neighborhood (Table I). Thus, for any pair of claimant and verifier, the only feasible chain of trust that we provided is the one containing all the CAs from the claimant to the verifier, which is the longest possible chain of trust.

To decrease the cost of obtaining a certificate, Conductor-enabled nodes can use a certificate cache, reducing the number of times they must consult a CA. We compared the *chain* scenario, with exactly the same environment setup, in two different cases. In one case there was no cache at all. In the other case optimal caching was deployed at each Conductor-enabled node, so the node never needed to contact its associated CA.

In both the *chain* scenario with optimal caching and the *tree* scenario, certificate retrieval from CAs is avoided, and the location of CAs has no impact on measurement results. But this is not true with the *chain* scenario without caching—certificate retrieval cost varies with the location of CAs. However, certificate retrieval cost can also significantly vary with many other factors. Therefore, we simply chose to collocate the associated CA of each Conductor-enabled node on the same machine.

Only successful cases were measured. Authentication never fails in the *chain* and *tree* scenarios. The RSA algorithm was used for public key encryption [25]. The signature algorithm was RSA-based with a SHA-1 message digest algorithm [22].

6.2.2 Resources

All Conductor-enabled nodes in these experiments were the same. Each was a Dell Inspiron 3500 machine running Linux Redhat 6.0 and IBM JDK 1.1.8 [10], with Intel Mobile Pentium II 333Mhz, 256KB cache, 64MB RAM, 4GB harddrive, and 100Mb/s Ethernet connection.

Each CA associated with a Conductor-enabled node shared the same resources as the Conductor-enabled node, collocated on the same machine. For the *tree* scenario, each non-leaf CA was running under Linux Redhat 6.0 on an Intel Celeron 300Mhz with 128KB cache, 128MB SDRAM, and a 100 Mb/s Ethernet connection.

Each Conductor-enabled node was also homogeneous in the sense that each machine was kept under the same workload with the same set of processes running. Only processes related to the experiment and normal system processes were running.

6.3 Results and Analysis

6.3.1 Plan setup latency

For each of the four scenarios, Figure 12 shows plan setup latency versus the number of Conductor-enabled nodes between two endpoints. Here, in the *chain* scenario, optimal caching is deployed.

The *null* scenario differs from the *none* scenario by including the entire security framework, but with no actual authentication. The difference between the performance of the *null* and *none* scenarios is thus the cost of the security framework devoid of cryptographic operations or other authentication mechanisms. That difference is statistically indistinguishable at the 99% confidence level (Figure 12).

Use of a security scheme such as the *tree* or *chain* (with optimal caching) introduces greater latency in plan setup than the *null* or *none* scenario (Figure 12). The increased costs include cryptographic operations and the transmission and handling of cryptographic messages. Recall that Conductor uses these cryptographic operations both to protect planning messages and do node authentication.

To protect message integrity, every node in a connection needs to sign its planning information and have it verified by the planner of the connection. Also the planner needs to sign both the plan and security scheme selector, which are verified afterwards by each node. With n Conductor-enabled nodes, this leads to (n+1) signing operations and 3(n-1) verification operations. This is same for both the *chain* and the *tree* scenarios.

Certificate verification distinguishes the *chain* scenario from the *tree* scenario in terms of plan setup latency. In the *chain* scenario, with n total Conductor-enabled nodes (Figure 11), the planner needs to verify (n-i+1) certificates to verify the public key of Conductor-enabled node i (i could be $1, 2, ... , n-1$). Notice that a planner only needs to verify each certificate once; it verifies 2(n-1) certificates in total. Conductor-enabled node i also must verify (n-i+1) certificates to authenticate the planner. So, the total number of certificates to verify before the plan is set up is

$$2(n-1)+\sum_{i=1}^{n-1}(n-i+1)=\frac{1}{2}(n^2+5n-6).$$

In the *tree* scenario, to authenticate the public key of each Conductor-enabled node, the planner needs to verify k certificates, where k is the depth of the certificate hierarchy. Since we assume the k certificates of one node do not overlap with those k certificates of another, the planner needs to do certificate verification $k*(n-1)$ times. Also, each Conductor-enabled node needs to verify k certificates to authenticate the public key of the planner. So, the total number of certificates to verify before the plan is set up is $2*k*(n-1)$. In our experiment, $k=3$, so the value is $6(n-1)$.

The above analysis shows that as more nodes are involved, the increased cost due to cryptographic operations varies linearly in the *tree* scenario and quadratically in the *chain* scenario (with optimal caching). This cost is paid once at setup time, and primarily represents cryptographic operations performed in Java. (In our experimental setup and using the *cryptix* library version 3.0.3 [6], the time taken to compute and verify cryptographic signatures of various Conductor messages varied between a few milliseconds and a few tens of milliseconds.)

The *chain* scenario without certificate caching incurs higher plan setup latency (Figure 13). In our experiment, each Conductor-enabled node and its associated CA are collocated on the same machine; otherwise, the latency could be even higher. But certificate retrieval latency is independent of the security implementation of Conductor.

Although the *chain* scenario leads to higher plan setup latency in many cases, it is easier to deploy than the *tree* scenario. The *tree* scenario requires a certificate hierarchy, and the root of the hierarchy must be trusted. This is not feasible in many circumstances. The *chain* scenario only requires each Conductor-enabled node to have an associated CA and some level of coverage overlap between CAs.

6.3.2 Bandwidth consumption

We analyzed bandwidth consumption during the plan setup procedure for four different scenarios. In the *chain* scenario, when optimal caching is used, no bandwidth is consumed for certificate retrieval.

To provide a fair comparison, we distributed the same plan in all four scenarios. We chose a plan in which every Conductor-enabled node is selected but no adaptors are deployed.

Figure 14 shows the bandwidth consumption per link in the four scenarios. In the *null* scenario, each Conductor-enabled node needs to forward the security scheme selector message to the next Conductor-enabled node, in addition to transmitting all the same messages as those in the *none* scenario. In the *chain* and *tree* scenarios, there are also other extra security-related messages consuming more bandwidth, such as the authenticator messages, the signatures of planning messages, and the signature of the scheme being used.

The difference between the bandwidth consumed in the *chain* and *tree* scenarios is caused by the authenticator messages. An authenticator is mainly composed of several certificates. With our experiment setup, every authenticator message in the *tree* scenario includes three certificates. In the *chain* scenario, however, every authenticator initially includes one certificate, and increases to two certificates after the first hop; in particular, the reverse authenticator of the planner will have one more certificate after every hop before reaching the initiator. With n Conductor-enabled nodes in the connection, the average number of certificates passing a link is $(1.5n+3)$ in the *tree* scenario, and $(1.5n-1)$ in the *chain* scenario. There are four certificates less per link on average in the *chain* scenario than in the *tree* scenario. No attempt was made to compact the data in the authenticator messages. Therefore, we believe the bandwidth consumption could be further optimized. Note that even without optimization with 9 nodes, the average bandwidth usage for any scenario is at most slightly more than 5000 bytes per link, which is acceptable for most situations.

7. Multiple Security Schemes

The discussion thus far has assumed a single security scheme for each connection selected by the client node. However, the ability to use multiple security schemes is desirable to allow authentication of as many intermediate Conductor-enabled nodes as possible, considering each node may support a different set of security schemes. In this section, we revisit the planning procedure and introduce an extended version of the secure planning procedure that supports multiple security schemes for a connection, with the cost of one additional round trip.

7.1 Secure Planning Algorithm For Multiple Security Schemes

If multiple security schemes are allowed, each node must select which scheme it will use to authenticate itself to the planner node, protect its planning information, and secure the deployment of a plan. Such scheme negotiation and selection is done in the first round trip. Each node may support several schemes. The client and the planner nodes

also have a list of schemes that would be acceptable for a given type of connection. Note that a client or planner may choose to accept a scheme that it does not support. In the following, we describe the algorithm step by step.

(1) The client node sends a list of acceptable schemes to the planner node. Each node, including the client, also sends a list of supported schemes to the planner node, ordered by its preference. The intermediate nodes do not need to see this message (but they may).

(2) The planner node computes the intersection of (a) the schemes acceptable to the client, (b) the schemes acceptable to the planner, and (c) the schemes supported by the planner. The result is the list of schemes selected for use in this connection.

Because each node specified its level of preference for each supported scheme, the planner will also know which scheme each node will use.

(3) The planner generates a message containing the list of selected schemes and the client's acceptable schemes (literally, the first message from the client node in step 1). Each selected scheme is used to generate a signature for this message. The message and all of the signatures are sent toward the client, visiting each node in turn.

(4) The planner generates an authenticator using each of the selected schemes, and sends it toward the client, visiting each node in turn.

(5) Given the messages from steps 3 and 4, each node can verify the selected schemes. If a node fails to verify the scheme, it will not participate. If the client fails to verify the scheme, or if the copy of the message returned from step 1 does not match, the connection will fail (or be reattempted).

If the above verification succeeds, each node sends its signed planning information and its reverse authenticator toward the planner, visiting other nodes along the way. It will also send a signature for the list of supported schemes that it sent in step 1. The scheme used to generate the signature and the authenticator should be the first one in the list of supported schemes sent from this node to the planner that was also selected by the planner for this connection. Note that the authenticator for a node may be augmented by other nodes along the way, no matter what scheme the intermediate node used to authenticate itself.

(6) Upon receipt of the messages from step 5, the planner will verify the identity of the nodes, select a set of authorized (trusted) nodes, and formulate a plan. Any node that cannot be authenticated or trusted and any planning information that cannot be validated will not be considered in planning.

The signature for supported schemes will also be verified. The connection will fail if the verification does not succeed.

(7) The planner will send a message containing the plan toward the client, visiting each node in turn. The planner will generate a set of signatures using each of the selected schemes and attach them to the message.

(8) Each node will verify the plan using the attached signature and the authentication information received in step 4. If a node cannot validate the plan, it will not participate. From an adaptation point of view, we can treat this as a node failure. ♦

Figure 15 shows an example that involves four Conductor-enabled nodes. Each node supports a different set of security schemes (Table II). Employing the above steps, both scheme **x** and scheme **y** are used for the connection.

This algorithm is justified as follows: For a given connection, the client must ensure that its list of acceptable security schemes was used by the planner to select schemes (otherwise the client can be tricked into using an unacceptable scheme, such as a scheme not enforcing a desired level of security); this is satisfied by steps 3 and 4 above. In addition, the planner must ensure that the list of supported schemes from each node is authentic (otherwise a node that supports an acceptable scheme may be regarded as supporting an unacceptable scheme, and will be thus unable to participate the planning process); this is guaranteed through steps 5 and 6. The planner also must ensure that the planning information from each node is authentic in order to form an effective plan; this is also enforced through steps 5 and 6. Finally, to avoid falsified adaptation, every node must be able to ensure that a plan to deploy is authentic; this is guaranteed through steps 4, 7, and 8. Every node also must ensure that a list of selected schemes is authentic (to avoid using a wrong scheme to authenticate itself to the planner and thus not be selected); this is implemented in steps 3 and 4.

7.2 Bridging Different Security Schemes

One drawback of the algorithm presented in the previous section is that it cannot select any scheme that is not supported by the planner node. This restriction is enforced in step 2 and ensures that the planner will be able to form a direct relationship with each participating node. If a node does not support any scheme supported (and hence selected) by the planner, it cannot participate. This is particularly troublesome since a group of neighboring nodes are likely to support a common security scheme, forming an island. If the client and the planner belong to different islands, secure communication may not be possible. However, this restriction can be removed if we allow nodes at the edges of islands that support more than one security scheme to act as bridges between security schemes.

We revise the previous algorithm to support bridge nodes. Below, we describe additions and changes to each step in the algorithm from Section 7.1.

(1) This step is unchanged.

(2) We add bridge selection to this step. Using the information in step 1, nodes are identified that can act as a bridge. A node can be a bridge if it supports an acceptable scheme not supported by the planner and also supports either (1) an acceptable scheme that is supported by the planner, or (2) an acceptable scheme that is supported by a bridge between itself and the planner. The list of selected schemes is then augmented by a list of bridge nodes and the translations they can perform. Note that bridges should only be specified if they will be useful in translating for another node between the bridge and the client. Some schemes that would not have been selected by the previous algorithm may be selected now because of the introduction of bridge nodes.

A node that is acting as a bridge node has a greater ability to affect the resulting adaptation than other nodes in the stream. A bridge node has capabilities similar to the planner. It is therefore important that the planner have strong trust in a node before selecting it to act as a bridge.

(3) The message (and its signature) sent by the planner in this step will receive special handling at bridge nodes. When the message reaches a bridge node, the bridge verifies the message signature and adds additional signatures for security schemes it is charged with translating. Note that since this signature is generated by the bridge node, authentication information for the bridge node will be required by any node wishing to verify the signature (see revised step 4).

(4) The authenticator from the planner in this step will also be handled by a bridge node in a special manner. When the authenticator reaches a bridge node, the bridge verifies the identity of the sender. If the sender is deemed authentic and trusted by the bridge, the bridge generates a new authentication message (to authenticate itself) in the scheme it is charged with translating. Note that the bridge knows to which nodes it is authenticating, because, like the planner, it can deduce what scheme each node will use.

(5) Verification of selected schemes is the same except that the message of selected schemes may be signed by a bridge node instead of the planner. Such messages can be verified using the bridge node's authenticator in the revised step 4.

Messages sent in this step, including each node's signed planning information message, a signature message of its supported schemes, and its authenticator, will receive special handling by bridge nodes. When a planning information message that has been signed in a manner that this bridge has been charged with translating is received at a bridge node, the bridge verifies the signature and replaces it with its own signature in a scheme supported by the planner or a downstream bridge. The signature provided for the schemes supported by a node can be similarly verified and replaced by a bridge when required. Finally, when an authenticator is received at a bridge node in a scheme that the bridge is charged with translating, the bridge verifies the identity of the source and replaces the message with a new authenticator for itself. Note that this means that the node providing authentication information has to authenticate to the bridge, not to the planner.

(6) This step is unchanged, except that a node's planning information and its supported schemes may be signed by a bridge node, thus requiring verification based on the bridge's authenticator from step 5.

(7) Bridge nodes may attach additional signatures to the plan produced in this step.

(8) This step is unchanged, except that the plan may be signed by a bridge node, thus requiring verification based on the bridge's authenticator from step 4. ♦

Figure 16 shows the usage of a bridge node (node C) for the same connection as in Figure 15. Node B can now participate using scheme **z**, because of the introduction of bridge node C. Node B can verify the signature of the scheme selection message (signed by the bridge node C), authenticate itself (to bridge node C), have its planning information signed (verifiable by bridge node C), and finally verify the plan (signed by bridge node C).

Justification of this revised algorithm follows the same argument as the earlier algorithm in Section 7.1. In addition, with the introduction of bride nodes, each message's authenticity is now verified in two ways: either the message must come from a source as indicated, or the message must come from a *trusted* bridge that has already verified the message's authenticity. For the former, the previous justification can be applied. The latter is complicated by the fact that several bridges may be used and may form a chain of trust. If each node, including bridge nodes, verifies that a received message was sent either from the original node or an authorized bridge, then an appropriate chain can be formed. Note that every node knows which nodes are authentic bridges from steps 3 and 4. So long as each bridge is trusted and verifies the identity of the previous bridge, the chain will be verified.

8. Non-Conductor-Enabled Client and Server

In the preceding sections we have assumed that both the client and the server of a connection are Conductor-enabled. We relax this assumption here. The initiator and the planner will not necessarily be the same node as the client and the server, respectively.

If the client is not Conductor-enabled, either because the client doesn' t care about security or because the client has specifically set up a Conductor node nearby that can be responsible for security decisions, then the initiator can act on behalf of the client with respect to security.

If the planner is not on the same node as the server, in our original algorithm (Sections 3 and 4) the participating nodes will not know which node is the planner. While public key-based security schemes do not require the nodes to know to whom they are authenticating, some authentication algorithms (notably Kerberos) do require such knowledge. In our revised secure planning algorithm (Section 7), however, the planner authenticates itself to other Conductor-enabled nodes first (including the initiator), correcting this problem. The assumption that the planner collocates with the server can be relaxed if either Kerberos-like authentication is not employed or the revised secure planning algorithm is used.

9. Other Open Architectures

Our design for securing Conductor is applicable to many other open architectures. In this section, we briefly address how to port the design to several adaptation systems and active networks.

9.1 Adaptation Systems

The Dynamic Proxy architecture developed at Columbia University [29] relies on a single proxy node to adapt a data stream. In this system the client dynamically selects and controls the adaptation performed at the proxy. The server plays no role in proxy management. To provide security for such a system in which the client also assumes the role of the planner in Conductor, we propose the following steps:

(1) The client specifies to the proxy node what security scheme must be followed.

(2) The proxy node, following the specified scheme in (1), sends to the client its authenticator and a signed planning information message.

(3) The client authenticates the proxy, verifies the planning information, and forms a plan.

(4) The client sends its reverse authenticator and the signed plan to the proxy node.

(5) The proxy node authenticates the client, and verifies and deploys the plan. ♦

A session key can also be distributed from the client to the proxy. Using the authentication information of the proxy node from step 2, the client can generate a session key, encrypt it, and send it to the proxy. For instance, if public key-based authentication is used, the proxy's public key can be used to encrypt the session key.

Protocol Boosters [8] adapts protocols by inserting boosters into a network. While no planning procedure has been specified, a policy is needed in order to deploy protocol boosters. If Conductor's planning architecture were ported to Protocol Boosters, we believe that Conductor's security design could also be utilized.

9.2 Active Networks

The security design for Conductor can be used to secure both an active network application and active packets in general.

9.2.1 Securing an active network application

Consider "active traceroute" as a sample application: a message traveling from end1 to end2 is stamped with the identity of each active node en route and returns with a record of the path from end1 to end2. The security task is to

ensure the identities are authentic. Assuming a single security scheme is enforced, as in Conductor (Section 3), we can let each active node add its own authentication information and signature. End2 can verify the authentication information, mark which recorded identity is authentic, and send the path information back to end1; this path information is signed by end2, requiring reverse authentication of end2 as well. To support multiple security schemes, similar handling can be enforced as shown in Section 7.

9.2.2 Securing active packets

An active packet, carrying both data and instructions, faces a danger that a malicious node on the way may corrupt the packet. Again, Conductor's security mechanism can be adopted to solve this problem.

With the goal to only allow trusted nodes to read/modify an active packet, a basic design follows, where a single scheme is enforced:

(1) The client that is going to run an active application sends out its security scheme and an authenticator towards the server.

(2) Each active node en route also authenticates itself to the server, using the selected scheme by the client.

(3) The server authenticates each node and selects trusted active nodes.

(4) For each selected trusted node, the server sends a reverse authenticator for itself, the signature of the client's security scheme specification message, and an encrypted session key.

(5) Each node authenticates the server, verifies that the specified security scheme is indeed enforced, and receives the distributed session key.

(6) The client starts sending active packets, encrypted with the session key. Only trusted active nodes will be able to decrypt the packets. ♦

10. Related Work

Security has been identified by many researchers as a key issue in open architectures. Much research has focused on protecting network elements from malicious code [1] [19], while comparatively less attention has been paid to

protecting data streams from misbehaving network elements. Murphy noted that in active networks, end-to-end security strategies do not always work because of the participation of intermediate active nodes [20]. Jackson proposed a possible packet format in active networks to support data integrity via signature [12], but data confidentiality is not addressed, and the approach is expensive on a per-packet basis. Researchers at the University of W. Sydney, Australia, identified the need for data confidentiality in active networks, and analyzed the difficulties with both end-to-end encryption and link encryption in supporting data confidentiality [27]. Researchers from NAI Labs proposed a hop-by-hop integrity model between active nodes that are "adjacent" in the active network topology, where a secret key returned from a trusted third-party is associated with each pair of active nodes; but this work assumes every active node is already trusted [7].

Research on data secrecy in open architecture has not typically included the notion that some nodes are trusted and some are not. Secure planning (together with encryption) can be used to control modifications of the data stream. Virtual link encryption, as proposed in this paper, provides data security in open architecture networks while still allowing intermediate nodes to adapt the data with reduced performance overhead. In particular, node authentication is required (as demonstrated in Conductor), and only those nodes scheduled to adapt data should receive session keys in order to access data plaintext.

Applications can require different security policies in different situations. An application should be able to select a specific security policy (or compose one as exemplified in [17]) and enforce it. Seraphim provides a framework that allows users or applications to enforce their own security policies in active networks, but it relies on a trusted third-authority to authenticate the security policy [4]. Conductor instead relies on one or more security schemes, selected by the initiator and/or the planner of a connection, to ensure that only those schemes are used.

IPsec [15] provides authentication, encryption and other security services at the IP layer. IPsec is primarily designed for point-to-point services, in contrast to virtual link encryption where many points are involved. If we used IPsec for Conductor security, a channel from each Conductor-enabled node to the planner (or a bridge node introduced in Section 7.2) or vice versa, bound with specific security association (SA) or SA bundles, would need to be independently established and maintained. ISAKMP [16] provides a framework to establish an SA, but it still requires a key exchange protocol such as IKE [9]. Via each channel, a node could authenticate itself or transmit signed planning information to the planner. Similarly, these channels allow the planner to authenticate itself or

transmit a signed plan or encrypted session key to each node. As illustrated in the chain authentication scheme (Section 4.2.2), intermediate nodes can sometimes provide additional information, which allows a node to be authenticated when it otherwise would not. Because IPsec channels are independent, the intermediate nodes are hidden from an IPsec channel and cannot provide such help. If IPsec is also to be used to protect user data transmitted from one Conductor-enabled node to another, a corresponding IPsec channel needs to be built as well. An SA must be separately set up for each individual virtual link.

11. Conclusions

Open architecture systems will not always consist of fully trusted nodes. Data transmissions of differing sensitivity will have different requirements about which adaptation nodes can be trusted to handle their data. The complexity of open architectures and the speed required for controlling and interacting with them suggest that programs (the application, the underlying open architecture planning system, etc.) will frequently be required to make decisions on which open architecture components to trust with their data.

We have described a design and implementation for a system to handle these problems in a challenging case. Conductor assumes no user control or interaction when a new data transmission is being handled. Instead, Conductor must make all decisions itself, including security decisions, based on current conditions, predefined user preferences, and known characteristics of the data flow.

Conductor's security architecture allows individual data transmissions to use different security boxes to achieve different levels and styles of authentication security. These security boxes could be chosen by pre-set user preferences, interaction with other security systems (such as intrusion detection systems), or by intelligent analysis of the data stream and prevailing security conditions.

Our implementation of this design demonstrates the feasibility of the concept. The security mechanisms described here add relatively little overhead to the connection setup phase, other than cryptographic operations required for authentication. The ongoing transmission similarly pays few overhead costs beyond any cryptography that is necessary to achieve its security goals.

While designed for the Conductor system, the same security architecture could also be used for many other open architecture systems. While it does not incorporate other security features required for success of open architectures (such as mobile code safety), the Conductor mechanism is compatible with solutions to these problems as addressed by other research groups.

Overall, this work demonstrates that it is feasible to dynamically choose the open architecture nodes to be used for a sensitive data transmission. Further, it is possible to design a sufficiently general system to allow different users and applications to apply their own authentication requirements to the node selection process. As an early example of a system that attempts to provide this type of security for its users, the Conductor system also points out the necessity of securing the gathering of information used to choose a course of action, and the importance of securing the instructions on what that course of action will be.

References

[1] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith, “A secure active network environment architecture: realization in SwitchWare,” IEEE Network, Vol.12, No.3, May-June 1998.

[2] D. Balfanz and E. W. Felten, “A Java filter,” Technical Report 567-97, Dept of Computer Science, Princeton University, September 1997.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, “The KeyNote trust-management system version 2,” RFC 2704, September 1999.

[4] R. H. Campbell, Z. Liu, M. D. Mickunas, P. Naldurg, and S. Yi, “Seraphim: dynamic interoperable security architecture for active networks,” IEEE OPENARCH 2000, Tel-Aviv, Israel, March 2000.

[5] M. Clifford, C. Lavine, and M. Bishop, “The solar trust model: authentication without limitation,” 14th Annual Computer Security Applications Conference, December 1998, Phoenix, AZ.

[6] The Cryptix cryptographic library, http://www.cryptix.org.

[7] R. Dandekar and S. Schwab, “A new hop-hop integrity model for active nets,” AMP Project, NAI Labs, May 2000.

[8] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh, “Protocol Boosters,” IEEE Journal on Selected Areas in Communication (Special Issue on Protocol Architectures for 21st Century Applications) Vol. 16, No. 3, April 1998.

[9] D. Harkins and D. Carrel, “The Internet key exchange (IKE),” RFC 2409, November 1998.

[10] IBM JDK 1.1.8, available at http://www.ibm.com/java/jdk.

[11] International Telecommunication Union, “Information technology – open systems interconnection – the directory: authentication framework,” ITU-T Recommendation X.509, 1995.

[12] A. W. Jackson, “Security issues for stateless programmable networks,” panel session on security issues in programmable networks, IEEE OPENARCH 1999, New York, NY, March 1999.

[13] Java™ Cryptography Architecture, http://java.sun.com/security.

[14] S. Kent, “Privacy enhancement for Internet electronic mail: part II: certificate-based key management,” RFC 1422, BBN, February 1993.

[15] S. Kent and R. Atkinson, “Security architecture for the Internet protocol,” RFC 2401, November 1998.

[16] D. Maughan, M. Schertler, M. Schneider, and J. Turner, “Internet security association and key management protocol (ISAKMP),” RFC 2408, November 1998.

[17] P. McDaniel, A. Prakash, and P. Honeyman, "Antigone: a flexible framework for secure group communication," Proceedings of the 8th USENIX Security Symposium, August 1999.

[18] G. McGraw and E. W. Felten, Securing Java: Getting Down to Business with Mobile Code, John Wiley & Sons, Inc, 1999. Also available at http://www.securingjava.com.

[19] S. Murphy and T. Samples, "Secure active network prototypes," NAI Labs, Network Associates, 1999. Available at http://www.darpa.mil/ito/psum1999/G796-0.html.

[20] S. Murphy, "Design issues affecting security for active networks," panel session on security issues in programmable networks, IEEE OPENARCH 1999, New York, NY, March 1999.

[21] B. C. Neuman and T. Ts'o., "Kerberos: An Authentication Service for Computer Networks," IEEE Communication Magazine, 32(9): 33-38, September 1994.

[22] NIST (National Institute of Standards and Technology), "Secure hash standard," FIPS Publication, 180-1, April 1995. Also available at http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[23] Panda project home page, available at http://fmg-www.cs.ucla.edu/Panda.

[24] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko, "Automated planning for open architectures," short paper presented at IEEE OPENARCH 2000, Tel-Aviv, Israel, March 2000.

[25] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signature and public key cryptosystems," Communications of the ACM, Vol. 21, No. 2, 1978.

[26] A. D. Rubin and D. E. Geer, "Mobile code security," IEEE Internet Computing, November 1998.

[27] V. Varadharajan, R. Shankaran, and M. Hitchens, "Active networks and security," Proceedings 22nd National Information Systems Security Conference, Vol.1, Arlington, VA, October 1999.

[28] M. Yarvis, A. Rudenko, P. Reiher, K. Eustice and G. Popek, "Conductor: enabling distributed adaptation," UCLA Tech Report, CSD-TR-010025, June 2001. Available at: http://fmg-www.cs.ucla.edu/Conductor/CSD-TR-010025.ps.

[29] B. Zenel and D. Duchamp, "A General Purpose Proxy Filtering Mechanism Applied to the Mobile Environment," Proceedings of MobiCom ' 97, Budapest, Hungary, October 1997.

Figure Captions

Figure 1. Planning process in Conductor

Figure 2. Security box in Conductor

Figure 3. Secured Planning process in Conductor. The security scheme is dynamically selected.

Figure 4.   CAs with flat distribution

Figure 5.   Certificate collection in the *chain* scheme along the path toward the planner

Figure 6. All certificates that may be finally available at planner D

Figure 7. Certificate collection in the reverse direction in the *chain* scheme

Figure 8. Key distribution with C selected

Figure 9. Replay attack by C during key distribution (C is not selected)

Figure 10. Replay counteraction with random number

Figure 11. Configuration of Conductor-enabled nodes and CAs in the chain scenario

Figure 12. Plan setup latency with different security schemes or no security (confidence level: 99%)

Figure 13. Comparison of plan setup latency in the chain scheme (confidence level: 99%)

Figure 14. Average bandwidth consumption per link

Figure 15. Handling multiple security schemes in a single connection with four Conductor-enabled nodes

Figure 16. Using bridge nodes to accommodate even more security schemes in a connection. As in Figure 1 and 3, A, B, C, and D are four Conductor-enabled nodes involved in a connection, where D is the planner. The parts in shaded area correspond to the additions or changes from Figure 15.

Figure 1

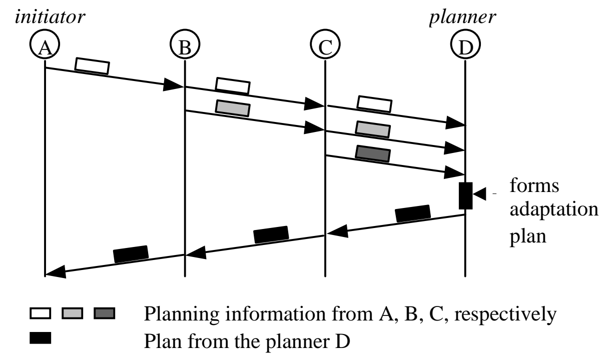


Figure 2

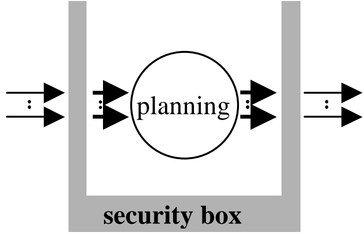


Figure 3

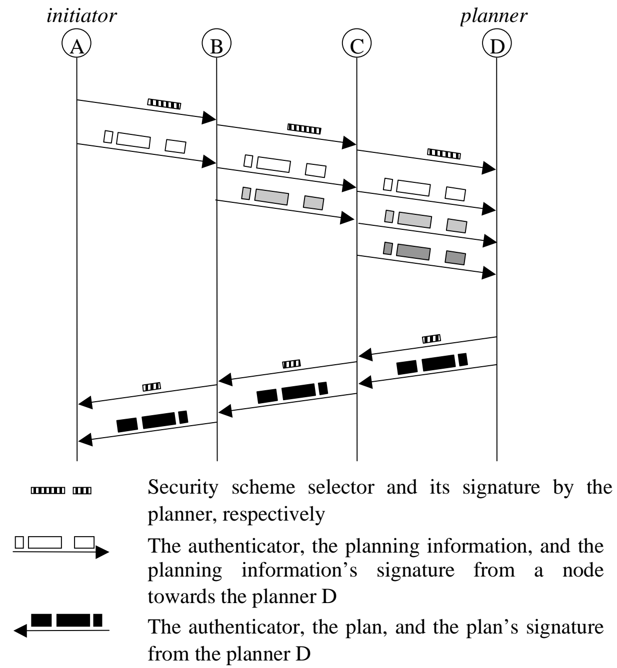


Figure 4

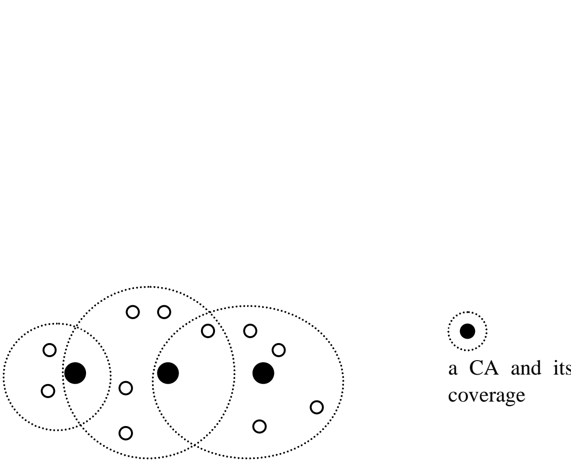


Figure 5

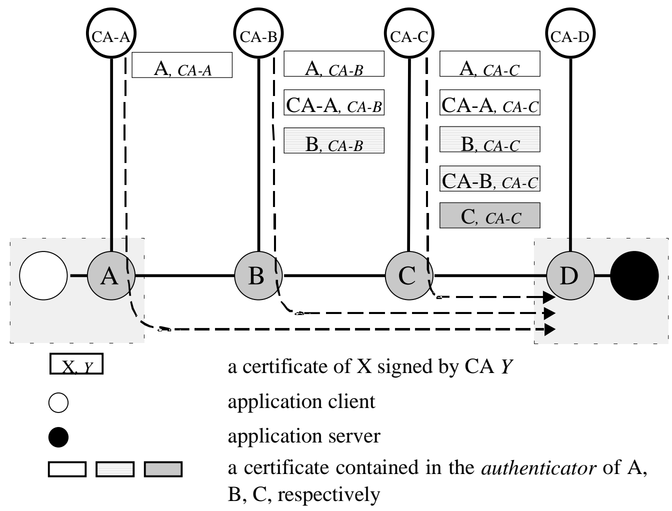


Figure 6

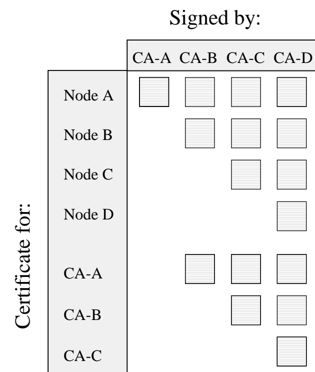


Figure 7

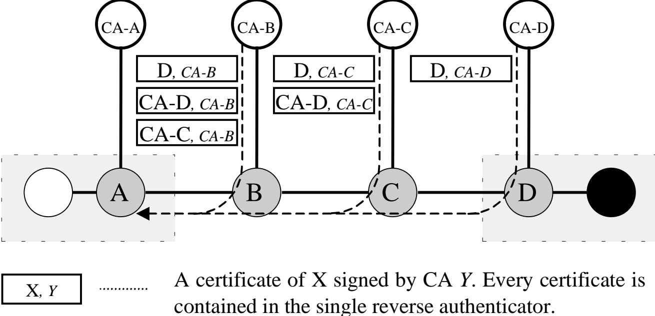


Figure 8

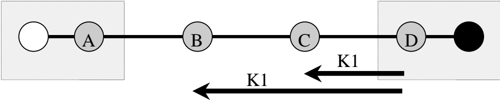


Figure 9

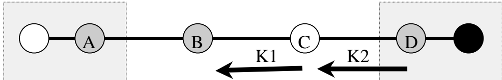
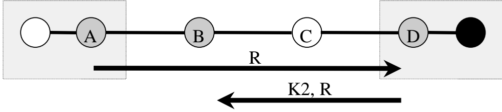


Figure 10



`

Figure 11

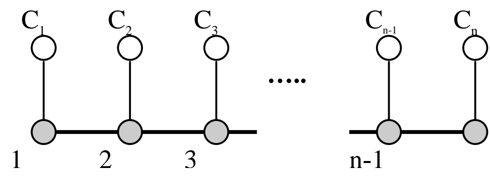
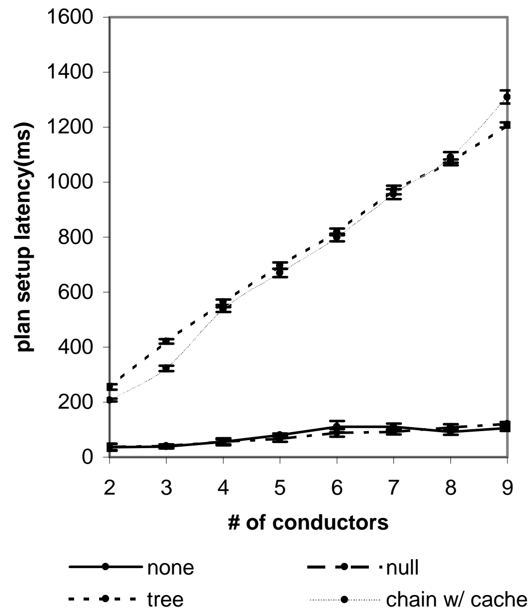


Figure 12



`

Figure 13

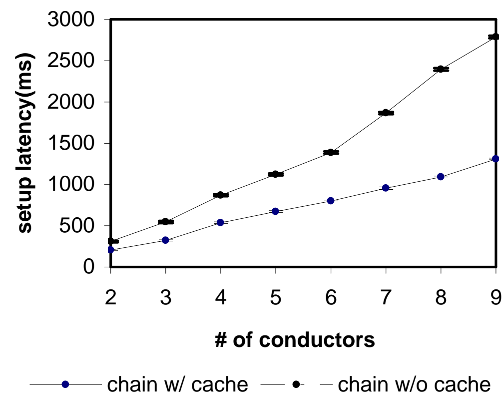


Figure 14

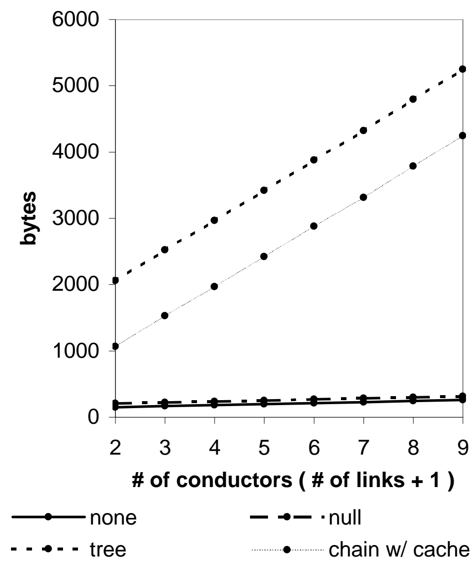


Figure 15

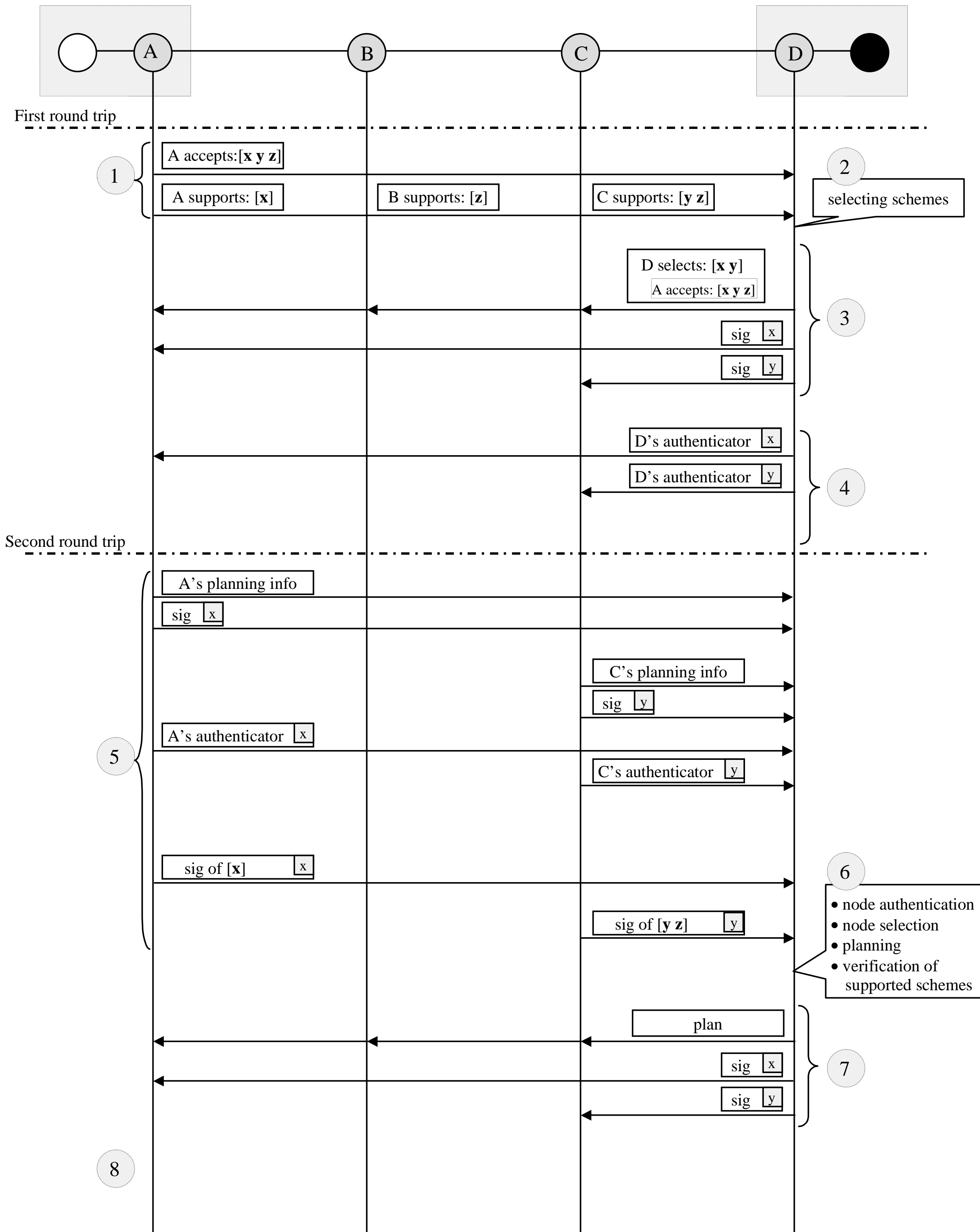


Figure 16

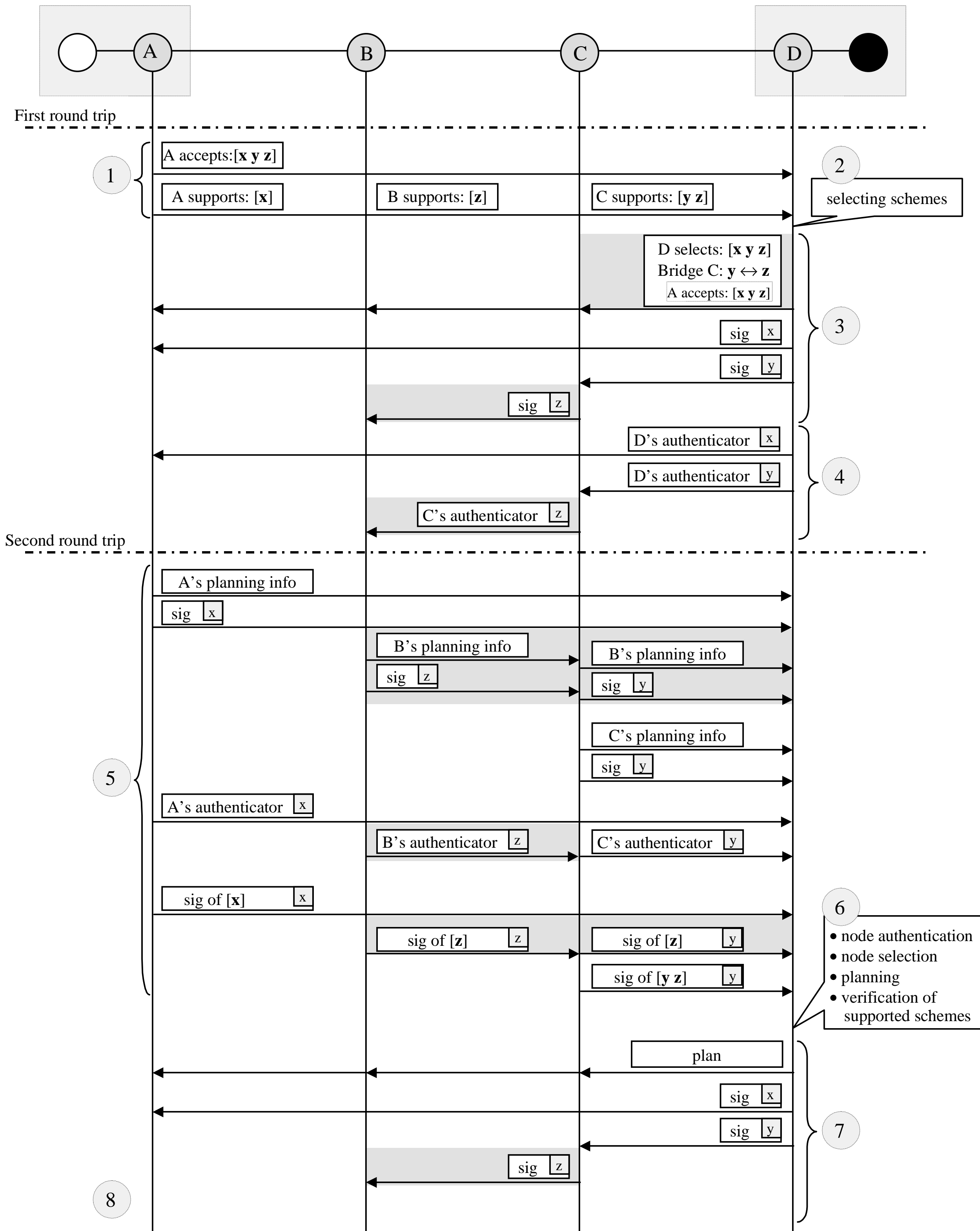


TABLE I
CA AND ITS COVERAGE IN THE *CHAIN* SCENARIO

CA	nodes that CA can certify
C_1	1, C_2
C_2	2, C_1, C_3
…	…
C_i	i, C_{i-1}, C_{i+1}
…	…
C_n	n, C_{n-1}

Table II
ACCEPTED AND SUPPORTED SCHEMES BY EACH NODE

schemes / nodes	accepted schemes	supported schemes
A	x, y, z	x
B		z
C		y, z
D	x, y, z	x, y

Author's Vitae

Jun Li is a Ph.D. candidate at the University of California, Los Angeles. He received a B.S. in computer science from Peking University in 1992, an M.S. in computer science from Institute of Software, Chinese Academy of Sciences, in 1995. His interests include Internet protocols and applications, Internet security, wireless network security, and open architectures.

Mark Yarvis is a Ph.D. candidate at the University of California, Los Angeles. He received a B.S. in computer science from UCLA in 1991 and an M.S. in computer science from UCLA in 1998. His interests include mobile networking, and infrastructural computing, and adaptive software techniques.

Peter Reiher received his Ph.D. in computer science from UCLA in 1987. Dr. Reiher is an associate adjunct professor in the Computer Science Department at UCLA. His research is in the fields of distributed operating systems, distributed file systems, parallel discrete event simulation, naming issues in distributed systems, mobile computing, and security issues in mobile and distributed systems.