The Demise of Transparent Distributed Operating Systems

Mark Kampe*
Sun Microsystems - Software Systems Group
(markk@west.sun.com)

* WARNING: this man is a well-known trouble-maker, whose views seldom represent those of his employer.

Friends, Romans, OS weenies,
Lend me your ears;
I come to bury Transparent Distributed Operating Systems, not to praise them.
The evil that men do lives after them,
The good is oft interred with their bones;
So let it be with system call transparency.
Experience hath said they were ambitious;
If it were so, it was a grievous fault;
And grievously will they answer for it.

Overview of Presentation
- Transparent Distributed OS ... looking backwards
  - what problems did they hope to solve?
  - how are we solving those problems today?
- Transparency ... why it doesn't work
  - transparent distribution and "Deutsch's Seven Falacies"
  - the challenge of transparent High Availability
- Distributed computing ... looking forward
  - looser (humbler) coupling, contemporary examples
  - evolution of distributed service interfaces and platforms

Transparent Distributed Operating System
- start with an existing operating system
  - with all of its APIs and supported applications
- run copies on multiple computers in a LAN
- change OS to coordinate all system resources
  - processes see the union of the resources on all nodes
- applications can run on any computer in cluster
  - share resources and interact is if on a single system
- creates the illusion of a Single System Image (SSI)
### Distributed OS: Costs and Benefits

- **Costs**
  - distributed resource management is very complex
  - coordination interactions are time-consuming
  - interdependencies create many new modes of failure

- **Benefits**
  - thousands of legacy applications become distributed
  - thousands of legacy applications become HA
  - all without having to change a line of application code
  - if we can get it right, the benefits seem compelling

---

### Goals of Transparent Distributed OSs

- transparent distribution across multiple machines
  - legacy apps become distributed, no changes required

- **scalable performance**
  - increase system capacity by adding more machines

- **sharing of centralized resources**
  - work-group computing, lower cost of ownership

- **highly available services**
  - if one computer fails, others continue providing service

---

### Goal: Scalable Performance

- single-system-image illusion is very expensive
  - many copies of many resources must be kept in sync
  - distributed consensus/synchronization is complex
  - synchronization is the opposite of parallelism
  - tightly coupled synchronization does not scale well

- SMP (or even NUMA) is much more efficient
  - only one in-memory copy of each resource
  - synchronization achieved by atomic instructions

- n-way SMPs greatly out-perform n-way clusters

---

### Goal: Resource Sharing

- client/server solutions provide resource sharing
  - network file systems, network printers, naming servers

- these provide 99% of global distributed OS benefits
  - all users see the same files, devices, authentication, ...

- with better performance at a fraction of the cost
  - much simpler protocols and synchronization
  - can be implemented outside of the operating system

- better heterogeneity support
  - client/server protocols need not be tied to particular OS
**Goal: Simplified Management**

- there are protocols for remote management
  - Simple Network Management Protocol
  - Common Information Model/Web Based Enterprise Mgt
- there are protocols to automate configuration
  - Dynamic Host Configuration Protocol
  - Service Discovery Protocol, JINI, ...
- there are tools for network-wide configuration
  - LDAP based Directory Enabled Networking

**Surviving Goals: Transparency**

- transparent distribution
  - applications can exploit resources on other machines
  - applications can serve clients on other machines
  - OS hides all details of location and topology
- transparent high availability
  - each service can be provided by multiple applications
  - if one app fails, clients can be redirected to another
  - OS can automatically provide the required redirection
- can transparent distributed OS's achieve these?

**Transparency: Peter Deutsch's "Seven Falacies of Network Computing"**

- the network is reliable
- there is no latency (instant response time)
- the available bandwidth is infinite
- the network is secure
- the topology of the network does not change
- there is one administrator for the whole network
- the incremental cost of transporting more data is zero

**Performance Falacies**

(latency, bandwidth, cost of transport)

- network communication is not instantaneous
  - there are limits on how fast transactions can be processed
  - there are no "simultaneous" multi-node state changes
- network bandwidth is limited
  - we can saturate the network before we run out of CPUs
- sending more data aggravates both problems
- distributed applications will only perform well if the ratio of computation to communication is high
The Reliability Fallacy

• messages will be delayed or lost
  – senders cannot assume messages are received in order
  – senders cannot assume all messages are received
  – senders cannot assume no ack means no receipt
• nodes will fail or fall off the network
  – active session state can be lost in mid-operation
• distributed apps must be robust in the face of failures
  – failure detection and rebinding and consistency checking
  – atomic transactions with automatic roll-back and replay

The StableTopology Falacy

• continuous cluster membership changes
  – new nodes join cluster, old nodes die and leave cluster
• individual interconnects go down and up
  – available routes between host pairs change
  – path length, bandwidth, and latency change
  – partial connectivity and split-brain can occur
• distributed apps cannot assume static configuration
  – dynamic discovery/rendezvous for clients and servers
  – globally consistent state requires consensus/fencing

The Management Falacy

• the assumption of global administration
  – different nodes may have different copies of cfg data
  – different nodes may have different managers
  – cannot assume all systems configured in the same way
• the assumption of any administration
  – not all systems are actively managed all the time
  – problems may go unattended for weeks or longer
• distributed apps must negotiate configuration parms
• automatic error detection/recovery may be required

The Security Falacy

• an OS can provide secure resource management
  – it controls all process access to system resources
• resources in a network are not under OS control
  – resources are represented by bits on an insecure wire
  – those bits are subject to eaves-dropping and forgery
• distributed applications cannot assume that
  – other programs cannot read their messages
  – that all messages are from the indicated sender
• distributed apps require privacy and authentication
OS API level transparency?

- OS objects/operations designed for single systems
  - files, processes, IPC channels, semaphores
  - authentication, privacy, integrity provided by the OS
  - one copy of everything made synchronization trivial
  - updates made in memory with atomic instructions
  - cheap and ubiquitous inter-process communication
- these ignore the realities of distributed computing
  - semantics that seem reasonable on a single system become too weak to encompass distributed systems

High Availability

- increasingly more systems are mission-critical
- we don't know how to build perfect components
  - all hardware and software will eventually fail
- delivering Highly Available service involves
  - continuing despite individual component failures
  - secondaries that take-over when primaries fail
- distributed operating systems
  - manage shared resources across many computers
  - could provide a virtual Highly Available platform

Failure Detection/Recovery Process

The Keys to High Availability

- low failure rates
  - fewer components experience fewer failures
  - simpler components experience fewer failures
- good failure compartmentalization
  - individual component failures have limited impact
  - very few single-point or coupled failures
- high repair/recovery rates
  - prompt error detection and fast recovery
  - high success probability, minimal service disruption
SSI Clusters and Failure Rates

- implementing a global view of system resources
  - requires frequent inter-system interactions
  - involves extensive and complexly shared state
- more interactions mean more failure opportunities
  - including failures during failure-recovery
- complex algorithms are likely to have more bugs
- shared state creates chances for coupled failures
  - shared state may represent a single point of failure
  - errors in state may be passed on to other systems

Cold v.s. Warm Fail-Overs

- cold fail-over
  - start (or restart) all affected applications
  - in-progress sessions are lost, clients must reconnect
- warm fail-over
  - session state periodically check-pointed to secondary
  - secondary attempts to take over all active sessions
  - most sessions see no discontinuity in service
- cold fail-overs yield lower availability
  - slower recovery with greater loss of service

Warm Primary/Secondary Fail-Over

Successful Warm Fail-Overs

- secondary must have a complete snap-shot
  - secondary must know state of all active sessions
- secondary must have a consistent snap-shot
  - state updates should be atomic (all-or-nothing)
- secondary must have up-to-date snap-shot
  - snap-shot must reflect all confirmed transactions
- client must be able to resync with new secondary
  - learn of/recover from failed in-progress operations
  - ensure the validity of all current resource handles
Transparency v.s. Warm Fail-Over

- much application state resides in process memory
  - distributed shared is very slow and expensive
  - random memory snapshots may not be consistent
  - memory copies likely to carry disease from primary
- persistent application state is often kept in files
  - file updates tend to be relatively infrequent
  - file updates may not represent atomic transactions
- it is difficult to do "transparent" state replication
  - much better for apps to make explicit check-points

Cold fail-overs can be performed transparently
- but cold service fail-overs are slow and disruptive
Most apps weren't designed for warm-failover
- apps weren't written to checkpoint transactions
- apps weren't written to inherit existing sessions
OS cannot "infer" the transaction boundaries
- we don't know when the state is consistent
- we don't know exactly what data needs to be sent
Traditional OS APIs do not enable warm fail-overs

The Verdict on API Level Transparency

- OS APIs are inadequate for distributed computing
- OS APIs are inadequate to enable High Availability
- the resulting systems are expensive and complex
- the resulting systems do not scale well
- there are simpler ways to enable resource sharing
- there are better ways to simplify management
- Why continue beating our heads against this wall?
  Sic transit gloria mundi!
Where were did they go wrong?

- trying to make different things appear the same, instead of merely enabling distributed computing
- trying to make remote a special case of local, rather than recognizing local as a special case of remote
- using interfaces based on objects and operations that were never designed for distributed computing
- failing to impose rational structure on where data resides, and computation is performed
- failing to recognize protocols (rather than APIs) as the basis for client/server and peer/peer interactions

What did they get right?

- the notion of a "cluster" (changing federation of systems who cooperate in a defined set of services)
- global name spaces/location independent resources
- unifying local and remote operations/interactions
- hiding location and topology from the applications
- cluster membership/failure/recovery mechanisms
- distributed concensus/synchronization mechanisms
- robust, replicated, persistent storage

Moving Forward

- systems must be more loosely coupled
  - tight coupling is expensive, complicated, and error prone
  - minimize interactions, maximize independence
  - only synchronize operations when absolutely necessary
- objects and operations must be network-aware
  - client server interactions must recognize Deutch's lessons
  - requests can be delayed or fail, objects can be revoked
- objects state must be designed with fail-over in mind
  - stateless server protocols are intrinsically robust
  - state updates in committed, all-or-none transactions

Simpler Clusters: Web Farms

- examples
  - proxy servers, cache servers, HTTP servers
- large farms of completely independent servers
  - individual servers do not maintain persistent state
  - simple systems with very few coupled failures
- client/server interface not an API, but a protocol
  - HTTP with stateless servers and idempotent operations
  - front-end load-balancing switch detects server failures
  - switch automatically redirects traffic to good servers
Typical Web Farm

Internet

load balancing switch

server

server

server

server

Simpler Clusters: File Servers

- examples
  - NFS, Andrew, SMB
- small groups of coupled servers
  - secondary copies mirror updates to the primary server
  - consistency protocols much simpler than SSI clusters
- client/server interface not an API, but a protocol
  - often with stateless servers and idempotent operations
  - upon server failure, client automatically finds new server
  - in most cases, the application doesn't see the failure

Typical File Server Configuration

Intranet

FS A

server

FS A'

server

FS B

server

FS B'

server

client

client

client

client

updates

updates

Simpler Clusters: Database Servers

- examples
  - Oracle Parallel Server
- coupled servers with dual-ported RAID disks
  - highly optimized checkpointing between servers
  - more efficient/robust than general OS state replication
- client/server interface not an API, but a protocol
  - with committed/abortable update transactions
  - semi-transparent fail-overs (transactions may fail)
  - use IP aliasing to redirect clients to new servers
Highly Available Database Server

- client service network
- switch
- system 1 (checkpoints, locks) → system 2
- dual ported disks

Simpler Clusters: app servers

- examples
  - iPlanet, App-Center 2000
- two-tiered solution
  - applications execute in independent stateless servers
  - session state is maintained in HA back-end database
- client/server interface not an API, but a protocol
  - front-end distributes requests to any available server
  - if server fails, request can be re-sent to new server
  - new server obtains session state from database

Two Tiered Application Server

- Internet
- load balancing switch
- app server → app server → app server
- Highly Available Database (session state/object store)

Evolving Service Interfaces

- most legacy OS APIs proved to be inappropriate
  - they ignored Deutch's Seven Falacies
  - they were too low level, yielding poor data-modularity
  - they were platform-centric and impeded heterogeneity
- what interfaces might be more appropriate?
  - interfaces that acknowledge client/server boundaries
  - interfaces that allow servers to manage their data
  - remote service protocols (e.g. IMAP, LDAP, RPC)
  - ORB-brokered, well behaved, remote object interfaces
Evolving Distributed Computing Platforms

- the original goal
  - to hide the network from the applications
  - to enable legacy applications to exploit the network
- that didn't work
  - the solution proved to be simpler than the problem
- a revised goal
  - embrace distributed computing as a new paradigm
  - find architectural constructs that exploit its strengths
  - find new programming models that make it simple

Lessons we can (re)learn from this

Albert Einstein:
"Everything should be made as simple as possible
... but no simpler!"

Millions of grand-fathers (including mine):
"Oh what a tangled web we weave, when first we
practice to deceive."