QoS-enabled Middleware for Managing and Controlling High-Speed Networks

Douglas C. Schmidt
Associate Professor &
Director of the Center for
Distributed Object computing
Computer Science Dept.
Washington University, St. Louis
www.cs.wustl.edu/~schmidt/

Sponsors
NSF, DARPA, Bellcore/Telcordia, BBN, Boeing, CDI/GDIS, Hughes, Kodak, Lockheed, Lucent, Microsoft, Motorola, Nokia, Nortel, OCI, OTI, SAIC, Siemens SCR, Siemens MED, Siemens ZT, Sprint, USENIX

April 13th, 1999

Motivation: the High-speed Network Software Crisis

- Symptoms
  - Network element *hardware* gets smaller, faster, cheaper
  - Control/management *software* gets larger, slower, more expensive

- Culprits
  - *Inherent* and *accidental* complexity

- Solution Approach
  - Manage & control network elements via QoS-enabled embedded middleware

www.arl.wustl.edu/arl/

Washington University, St. Louis

General Switch Management Protocol (GSMP)

- Features
  - Setup & release connections
  - Add & delete point-to-multipoint leaves
  - Manage ATM switch ports
  - Request configuration information & statistics
  - Low-level C APIs send RFC 2297 GSMP ATM messages

GSMP Proxy Server Configuration

- Features
  - Supports standard CORBA programming API
  - Can use standard ORB
  - Transparent to existing GSMP servers
  - Scales to distributed configuration
    - *i.e.*, one CP can control multiple switches

Low-level C APIs send RFC 2297 GSMP ATM messages
Embedded Middleware

**GSMP Embedded ORB Configuration**

**Features**
- Leverages middleware flexibility and standardization
- Multiple protocols can be supported
  - GSMP in-line bridging, IIOP, etc.

**Domain Challenges**
- High-speed (20 Gbps) ATM switches
- Low-latency and statistical real-time deadlines
- COTS infrastructure, open systems, and small footprint

**Problem: Lack of QoS-enabled Middleware**
- Many telecom applications require QoS guarantees
  - e.g., call-processing, network/switch management, wireless
- Building these applications manually is hard
- Existing middleware doesn’t support QoS effectively
  - e.g., CORBA, DCOM, DCE, Java
- Solutions must be integrated horizontally & vertically

**Candidate Solution: CORBA**

**Goals of CORBA**
- Simplify distribution by automating
  - Object location & activation
  - Parameter marshaling
  - Demultiplexing
  - Error handling
- Provide foundation for higher-level services

[www.cs.wustl.edu/~schmidt/corba.html](http://www.cs.wustl.edu/~schmidt/corba.html)
Caveat: Limitations of CORBA for QoS

- Lack of QoS specifications
- Lack of QoS enforcement
- Lack of real-time programming features
- Lack of performance optimizations

Features
1. End-to-end priority propagation
2. Protocol properties
3. Thread pools
4. Explicit binding
5. Mutex IDL

Our Approach: The ACE ORB (TAO)
- An open-source, standards-based, real-time, high-performance CORBA ORB
- Runs on POSIX, Win32, & embedded RT platforms e.g., VxWorks, Chorus, LynxOS
- Leverages ACE

The ADAPTIVE Communication Environment (ACE)
- A concurrent OO networking framework
- Available in C++ and Java
- Ported to POSIX, Win32, and RTOS
- x-Kernel
- SysV STREAMS

Related work

ACE Overview

TAO Overview

www.cs.wustl.edu/~schmidt/TAO.html

Our Approach: The ACE ORB (TAO)

www.cs.wustl.edu/~schmidt/ACE.html

Overview of the Real-time CORBA Specification

www.cs.wustl.edu/~schmidt/RT-ORB.ps.gz

Related work

Related work

www.cs.wustl.edu/~schmidt/RT-ORB.ps.gz
ACE and TAO Statistics

- Over 30 person-years of effort
  - ACE > 185,000 LOC
  - TAO > 100,000 LOC
  - TAO IDL compiler > 100,000 LOC
  - TAO CORBA Object Services > 150,000 LOC
- Ported to UNIX, Win32, MVS, and RTOS platforms
- Large user community
  - www.cs.wustl.edu/~schmidt/ACE-users.html
- Supported commercially
  - ACE → www.riverace.com
  - TAO → www.ociweb.com

Optimization Challenges for QoS-enabled ORBs

Key Challenges

- Alleviate priority inversion and non-determinism
- Reduce demultiplexing latency/jitter
- Ensure protocol flexibility
- Specify QoS requirements
- Schedule operations
- Eliminate (de)marshaling overhead
- Minimize footprint

Problem: Optimizing Complex Software

Common Problems →

- Optimizing complex software is hard
- Small “mistakes” can be costly

Solution Approach (Iterative) →

- Pinpoint overhead via white-box metrics
  - e.g., Quantify and VMEtro
- Apply patterns and framework components
- Revalidate via white-box and black-box metrics
**Definition**

- Optimization principle patterns document rules for avoiding common design and implementation problems that can degrade the performance, scalability, and predictability of complex systems.

**Key Principle Patterns Used in TAO**

<table>
<thead>
<tr>
<th>Principle Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Optimize for the common case</td>
<td>Remove gratuitous waste</td>
</tr>
<tr>
<td>2. Remove inefficient general-purpose functions with efficient special-purpose ones</td>
<td>Replace inefficient general-purpose functions with efficient special-purpose ones</td>
</tr>
<tr>
<td>3. Shift computation in time, e.g., precompute</td>
<td>Store redundant state to speed-up expenses operations</td>
</tr>
<tr>
<td>4. Don’t be tied to reference implementations/models</td>
<td>Pass hints between layers and components</td>
</tr>
<tr>
<td>5. Use efficient/predictable data structures</td>
<td>Don’t be tied to reference implementations/models</td>
</tr>
</tbody>
</table>

**ORB Latency and Priority Inversion Experiments**

- **Method**
  - Vary ORBs, hold OS constant
  - Solaris real-time threads
  - High priority client $C_0$ connects to servant $S_0$ with matching priorities
  - Clients $C_1 \ldots C_n$ have same lower priority
  - Clients $C_1 \ldots C_n$ connect to servant $S_1$
  - Clients invoke two-way CORBA calls that cube a number on the servant and returns result

**ORB Latency and Priority Inversion Results**

- Synopsis of Results
  - TAO’s latency is lowest for large # of clients
  - TAO avoids priority inversion
    - *i.e.*, high priority client always has lowest latency
  - Primary overhead stems from concurrency and connection architecture
    - *e.g.*, synchronization and context switching

**ORB Jitter Results**

- **Definition**
  - Jitter $\rightarrow$ standard deviation from average latency

- **Synopsis of Results**
  - TAO’s jitter is lowest and most consistent
  - CORBAplus’ jitter is highest and most variable
**Problem: Improper ORB Concurrency Models**

**Common Problems**
- High context switching and synchronization overhead
- Thread-level and packet-level priority inversions
- Lack of application control over concurrency model

www.cs.wustl.edu/~schmidt/CACM-arch.ps.gz

---

**Problem: ORB Shared Connection Models**

**Common Problems**
- Request-level priority inversions
  - Sharing multiple priorities on a single connection
- Complex connection multiplexing
- Synchronization overhead

www.cs.wustl.edu/~schmidt/RTAS-98.ps.gz

---

**Problem: High Locking Overhead**

**Common Problems**
- Locking overhead affects latency and jitter significantly
- Memory management commonly involves locking

www.cs.wustl.edu/~schmidt/RTAS-98.ps.gz

---

**Solution: TAO’s ORB Endsyste Architecture**

**Solution Approach**
- Integrate scheduler into ORB endsystem
- Support multiple scheduling strategies
- Co-schedule threads

**Principle Patterns**
- Pass hints, precompute, optimize common case, remove gratuitous waste, store state, don’t be tied to reference implementations & models

www.cs.wustl.edu/~schmidt/RTAS-98.ps.gz
Problem: Reducing Demultiplexing Latency

**Design Challenges**
- Minimize demultiplexing layers
- Provide $O(1)$ operation demultiplexing through all layers
- Avoid priority inversions
- Remain CORBA-compliant

www.cs.wustl.edu/~schmidt/POA.ps.gz

Solution: TAO’s Request Demultiplexing Optimizations

Demuxing
- www.cs.wustl.edu/~schmidt/{ieee_tc-97,COOTS-99}.ps.gz

Perfect hashing
- www.cs.wustl.edu/~schmidt/gperf.ps.gz

Synopsis of Results =>
- Perfect Hashing
  - Highly predictable
  - Low-latency
- Others strategies slower

Principle Patterns =>
- Precompute, use predictable data structures, remove gratuitous waste
Embedded Middleware

Real-time ORB/OS Performance Experiments

Method
- Vary OS, hold ORBs constant
- Single-processor Intel Pentium II 450 Mhz, 256 Mbytes of RAM
- Client and servant run on the same machine
- Servant \( C_i \) connects to servant \( S_i \) with priority \( P_i \)
  \( i \) ranges from 1 to 50
- Clients invoke two-way CORBA calls that cube a number on the servant and returns result

www.cs.wustl.edu/~schmidt/RT-OS.ps.gz

Real-time ORB/OS Performance Results

High-priority Client Latency
Low-priority Clients Latency

Real-time ORB/OS Jitter Results

High-priority Client Jitter
Low-priority Clients Jitter
Problem: Hard-coded ORB Messaging and Transport Protocols

Existing ORBs don’t support “pluggable protocols”

GIOP/IIOP are not sufficient, e.g.:
- GIOP message footprint may be too large
- TCP lacks necessary QoS
- Legacy commitments to existing protocols

One Solution: Hacking GIOP

GIOP requests include fields that aren’t needed in homogeneous embedded applications
- e.g., GIOP magic #, GIOP version, byte order, request principal, etc.

TAO’s gioplite option save 15 bytes per-request, yielding these calls-per-second:

<table>
<thead>
<tr>
<th></th>
<th>Marshaling-enabled</th>
<th>Marshaling-disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>GIOP</td>
<td>2.876</td>
<td>2.937</td>
</tr>
<tr>
<td>GIOPlite</td>
<td>2.883</td>
<td>2.978</td>
</tr>
</tbody>
</table>

The result is a measureable, but small (2%), improvement in throughput/latency

Our pluggable protocols framework will all much greater decreases in IOP request sizes, as well as more flexible support for multiple transport protocols

However, there will be no changes required to the standard CORBA programming model

Features
- Pluggable ORB messaging and transport protocols
- Highly efficient and predictable behavior

Principle Patterns
- Replace general-purpose functions (protocols) with special-purpose ones
- This makes ORBs inflexible and inefficient
Pinpointing ORB Overhead with VMEIO Timeprobes

- Timeprobes use VMEIO monitor, which measures end-to-end time
- Timeprobe overhead is minimal, i.e., 1 μsec

ORB & VME One-way Overhead Results

Synopsis of Results
- ORB overhead is relatively constant and low
  - e.g., ~110 μsecs per end-to-end operation
- Bottleneck is VME driver and OS, not ORB

Ethernet & VME Two-way Latency Results

Synopsis of Results
- VME protocol is much faster than Ethernet
  - No application changes are required to support VME

VxWorks running on PowerPC over 320 Mbps VME & Ethernet

Washington University, St. Louis
### Client Whitebox Latency Results

<table>
<thead>
<tr>
<th>Direction</th>
<th>Client Activities</th>
<th>Absolute Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outgoing</td>
<td>1. Initialization</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>2. Get object reference</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3. Parameter marshal</td>
<td>operation dependent</td>
</tr>
<tr>
<td></td>
<td>4. ORB messaging send</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5. ORB transport send</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>6. I/O send</td>
<td>operation dependent</td>
</tr>
<tr>
<td>Incoming</td>
<td>7. I/O receive</td>
<td>operation dependent</td>
</tr>
<tr>
<td></td>
<td>8. ORB transport recv</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>9. ORB messaging recv</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>10. Parameter demarshal</td>
<td>operation dependent</td>
</tr>
</tbody>
</table>

### Server Whitebox Latency Results

<table>
<thead>
<tr>
<th>Direction</th>
<th>Server Activities</th>
<th>Absolute Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoming</td>
<td>1. I/O receive</td>
<td>operation dependent</td>
</tr>
<tr>
<td></td>
<td>2. ORB transport recv</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>3. ORB messaging recv</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>4. Parsing object key</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>5. POA demux</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6. Servant demux</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>7. Operation demux</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8. Parameter demarshal</td>
<td>operation dependent</td>
</tr>
<tr>
<td></td>
<td>9. User upcall</td>
<td>servant dependent</td>
</tr>
<tr>
<td>Outgoing</td>
<td>10. Return value marshal</td>
<td>operation dependent</td>
</tr>
<tr>
<td></td>
<td>11. ORB messaging send</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>12. ORB transport send</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>13. I/O send</td>
<td>operation dependent</td>
</tr>
</tbody>
</table>

### Problem: Overly Large Memory Footprint

- **Problem**
  - ORB footprint is too big for many telecom apps
- **Unnecessary Features**
  - DSI & DII
  - Dynamic Any
  - Interface Repository
  - Advanced POA features
  - CORBA/COM interworking

- Problem
  - ORB footprint is too big for many telecom apps

- Unnecessary Features
  - DSI & DII
  - Dynamic Any
  - Interface Repository
  - Advanced POA features
  - CORBA/COM interworking

### Solution: Minimum CORBA

<table>
<thead>
<tr>
<th>Component</th>
<th>CORBA</th>
<th>Minimum CORBA</th>
<th>Percentage Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>POA</td>
<td>281,896</td>
<td>207,216</td>
<td>26.5</td>
</tr>
<tr>
<td>ORB Core</td>
<td>347,080</td>
<td>330,304</td>
<td>4.8</td>
</tr>
<tr>
<td>Dynamic Any</td>
<td>131,305</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>CDR Interpreter</td>
<td>68,687</td>
<td>68,775</td>
<td>0</td>
</tr>
<tr>
<td>IDL Compiler</td>
<td>10,488</td>
<td>10,512</td>
<td>0</td>
</tr>
<tr>
<td>Pluggable Protocols</td>
<td>14,610</td>
<td>14,674</td>
<td>0</td>
</tr>
<tr>
<td>Default Resources</td>
<td>7,919</td>
<td>7,975</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>861,985</td>
<td>639,456</td>
<td>25.8</td>
</tr>
</tbody>
</table>

Applying Minimum CORBA subsetting to TAO reduces memory footprint by ~25% and increases ORB determinism
**Lessons Learned Developing QoS-enabled ORBs**

- Avoid dynamic connection management
- Minimize dynamic memory management and data copying
- Avoid multiplexing connections for different priority threads
- Avoid complex concurrency models
- Integrate ORB with OS and I/O subsystem and avoid reimplementing OS mechanisms
- Guide ORB design by empirical benchmarks and patterns

**Concluding Remarks**

- Researchers and developers of distributed, real-time, embedded telecom applications confront common challenges
  - e.g., service initialization and distribution, error handling, flow control, scheduling, event demultiplexing, concurrency control, persistence, fault tolerance
- Successful researchers and developers apply **patterns**, **frameworks**, and **components** to resolve these challenges
- Careful application of patterns can yield efficient, predictable, scalable, and flexible middleware
  - *i.e.*, middleware performance is largely an “implementation detail”
- Next-generation ORBs for telecom will be highly QoS-enabled, though many research challenges remain

**Summary of TAO Research Project**

**Completed work**
- First POA and first deployed real-time CORBA scheduling service
- Pluggable protocols framework
- Minimized ORB Core priority inversion and non-determinism
- Reduced latency via demuxing optimizations
- Co-submitters on OMG’s real-time CORBA spec

**Ongoing work**
- Dynamic/hybrid scheduling of CORBA operations
- Distributed QoS, ATM I/O Subsystem, & open signaling
- Implement Real-time CORBA spec
- Tech. transfer via DARPA Quorum program and [www.ociweb.com](http://www.ociweb.com)
**Summary: Real-time Optimizations in TAO**

**Next Steps: New TAO Features and Optimizations**

- New Features
  - Real-time CORBA
  - Minimum CORBA
  - CORBA Messaging
  - Fault tolerance
- New Optimizations
  - Startup time
  - Event Channel
  - POA hierarchies

www.cs.wustl.edu/~schmidt/TAO-status.html

---

**Next Steps: Integrating TAO with ATM I/O Subsystem**

**Key Features**

- Vertical integration of QoS through ORB, OS, and ATM network
- Real-time I/O enhancements to Solaris kernel
- Provides rate-based QoS end-to-end
- Leverages APIC features for cell pacing and zero-copy buffering

~schmidt/RIO.ps.gz

---

**Next Steps: Strategized Scheduling Framework**

1. Specify BT_operation characteristics and dependencies
2. Populate BT_info repository
3. Assign static priority and static dependency
4. Map static priorities, dynamic priorities, and static dependency into dispatching priority and dispatching dependency
5. Assess schedulability
6. Assign dispatching queue configuration
7. Supply dispatching queue configuration to the ORB
8. Configure queues based on dispatching priority
9. Supply static portions of dispatching priority and dispatching dependency to the ORB

www.cs.wustl.edu/~schmidt/dynamic.ps.gz
Next Steps: Open ATM Signaling & Control

Efficiency
- Sockets for data transfer
- Uses CORBA for control messages and properties
- To get high performance

Flexibility
- Sockets for data transfer
- Uses CORBA for control messages and properties

Web URLs for Additional Information
- These slides: ~schmidt/TAO4.ps.gz
- More information on CORBA: ~schmidt/corba.html
- More info on ACE: ~schmidt/ACE.html
- More info on TAO: ~schmidt/TAO.html
- TAO Event Channel: ~schmidt/JSAC-98.ps.gz
- TAO static scheduling: ~schmidt/TAO.ps.gz
- TAO dynamic scheduling: ~schmidt/dynamic.ps.gz
- ORB Endsystem Architecture: ~schmidt/RIO.ps.gz
- Pluggable protocols: ~schmidt/pluggable_protocols.ps.gz

Web URLs for Additional Information (cont’d)
- Performance Measurements:
  - Demuxing latency: ~schmidt/COOTS-99.ps.gz
  - SII throughput: ~schmidt/SIGCOMM-96.ps.gz
  - DII throughput: ~schmidt/GLOBECOM-96.ps.gz
  - ORB latency & scalability: ~schmidt/ieee_tc-97.ps.gz
  - IIOP optimizations: ~schmidt/JSAC-99.ps.gz
  - Concurrency and connection models: ~schmidt/RT-perf.ps.gz
  - RTOS/ORB benchmarks:
    ~schmidt/RT-OS.ps.gz
    ~schmidt/words-99.ps.gz

Washington University, St. Louis