Implementing Real-time CORBA with Real-time Java

Ray Klefstad, Mayur Deshpande, Carlos O’Ryan, & Doug Schmidt
{coryan, schmidt}@uci.edu    {klefstad, mayur}@ics.uci.edu
University of California, Irvine

Wednesday, 18 July 2001
Research Sponsored by AFOSR
Motivation for QoS-enabled Middleware

Trends
- Hardware keeps getting smaller, faster, & cheaper
- Software keeps getting larger, slower, & more expensive

Historical Challenges
- Building distributed systems is hard
- Building them on-time & under budget is even harder

New Challenges
- Many mission-critical distributed applications require real-time QoS guarantees
  - e.g., combat systems, online trading, telecom
- Building QoS-enabled applications manually is tedious, error-prone, & expensive
- Conventional middleware does not support real-time QoS requirements effectively
Overview of CORBA

- Common Object Request Broker Architecture (CORBA)
  - A family of specifications
  - OMG is the standards body
  - Over 800 companies
- CORBA defines *interfaces*, not *implementations*
- It simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security

CORBA shields applications from heterogeneous platform *dependencies*
- e.g., languages, operating systems, networking protocols, hardware
Real-Time CORBA Overview

- RT CORBA adds QoS control to regular CORBA to improve the application predictability, e.g.,
  - Bounding priority inversions &
  - Managing resources end-to-end

Policies & mechanisms for resource configuration/control in RT-CORBA include:

1. Processor Resources
   - Thread pools
   - Priority models
   - Portable priorities

2. Communication Resources
   - Protocol policies
   - Explicit binding

3. Memory Resources
   - Request buffering

- These capabilities address some important real-time application development challenges
Motivations for Using Java & RT-Java

- Easier, faster development
  - Memory management is simpler
  - Language support for concurrency and synchronization
  - Large and powerful library
  - Advanced language features (Class loading, Reflection)
- Large programmer base (and growing!)
  - Over 1,000,000
  - Taught at many universities
  - Easily picked up by C++, Ada Programmers
- Real-time Specification for Java
  - Improved semantics & features for programming Real-time systems
Overview of Real-time Spec for JAVA (RTSJ)

- Mitigates features of Java that inhibit real-time behavior:
  - Garbage collection is optional, via:
    - NoHeapRealTimeThreads, Scoped Memory, Immortal Memory
  - Access to Physical Memory
  - Finer time granularity (High Resolution Timer where Available)
  - Stronger guarantees on threads
    - Highest priority runnable thread is always run
  - Minimized thread context switching
    - ~1-2 Microseconds down from 100s
Overview of RTSJ (cont’d)

• Features that improve RT behavior:
  • Explicit Asynchrony support
    • AsyncEventHandlers (AEH): Semantics similar to threads but:
      • No resources used while sleeping
      • Run in context of current thread
      • Can’t pass data to them
    • Asynchronous transfer of control
    • Asynchronous thread termination

• Extensive Scheduling support
  • Schedulable Objects (threads, AEHs), Admission control, feasibility analysis
  • Extend base fixed-priority preemptive scheduler to fit needs
Motivation for ZEN

• Integrate best aspects of several key technologies:
  • Java: Easy, fast programming
  • RT-Java: Development of real-time systems with Java
    • However, no standard facilities for developing distributed real-time applications (yet)
  • CORBA: Standards-based distributed Applications
  • RT-CORBA: CORBA with QoS capabilities

• ZEN project goals:
  • Make development of distributed real-time embedded (DRE) systems easier, faster & more portable
  • Provide open-source RT-CORBA ORB written in RT-Java to enhance international R&D efforts
Technical Goals for ZEN

• Real-time & embedded performance
  • Small initial footprint: grow only as needed
  • Low startup latency
  • Bounded jitter for ORB/POA operations
  • Eliminate sources of priority inversion
  • Allow control of RT-Java features by Apps

• Highly configurable
  • Statically
    • Select components once at compile time
    • Self-contained executable that can be burned to EEPROM
    • Highly optimized
  • Dynamically
    • Add components as needed at run-time
    • Dynamic class loading
    • Latency, deadline issues

• Easily extensible
  • e.g., new protocols, Object Adapters, IOR formats, etc.
ZEN Design Methodology

- Start with a flexible, extensible design based on TAO Design Patterns

- Build upon TAO implementation experience
- Write the simplest concrete implementation of each abstract class and factories which create instances of them
- Extend family of factories and concrete classes to support alternative features, protocols, etc.
- Develop our own IDL compiler using parser generators and visitors
- Build real-time CORBA and real-time Java into ZEN ground-up
  - No penalty for not using RT-features
- Work on optimizing the common cases
ZEN Architectural Design

• “Layered Pluggability”
  • Pluggable Messaging: GIOP, IIOP, IIOP
  • Pluggable POAs: RT-POA, Min-POA
  • Pluggable IOR formats: file:, IOR:, IOR:, http:, etc
  • Pluggable Resolvers: RootPOA, Naming_Service

<table>
<thead>
<tr>
<th></th>
<th>RT-POA</th>
<th>Min-POA</th>
<th>POA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request/Handler</td>
<td>Thread-Pool</td>
<td>Thread/Request</td>
<td></td>
</tr>
<tr>
<td>GIOP</td>
<td></td>
<td>UIOP</td>
<td></td>
</tr>
</tbody>
</table>

University of California, Irvine
ZEN Architectural Design (cont’d)

• Strategize
  • Strategized POA creation with dynamically loadable POA Policies
  • Strategized Request Handling
    • Same thread
    • Thread-Pool
ZEN Architectural Design (Cont’d)

• Aspectize
  • Modularization of cross-cutting concerns into different “Aspects” of the software
• Error/Debug/Logging
• Real Time?
• Reflection?

<table>
<thead>
<tr>
<th>REAL TIME</th>
<th>Debug / Logging On/Off</th>
<th>POA</th>
<th>REFLECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

University of California, Irvine
Applying RT-Java Features in ZEN

• Scoped Memory
  • Not Garbage Collected
  • Disposed as a whole after the scope is exited.
    • Almost stack-like in behavior
  • Limitations on access to/from Heap

• Possible Applications
  • CDR Streams buffer allocation
  • Demarshalled arguments could be allocated from Scoped Memory
    • But this affects the language mapping
    • May require extensions to the IDL compiler
  • Or the RT-CORBA Thread Pools

---

Marshall (params) → CDRStream

Scoped Memory

Send out to network

SM is reclaimed
Applying RT-Java Features (cont’d)

• Real-time Threads
  • RT-CORBA introduces policies to control POA thread pools
  • Application cannot control type of thread
  • Research question: Can NoHeap threads be used in the POA?

• Scheduling
  • Dynamic Scheduling RFP is a nice fit with RTSJ scheduling support
Applying RT-Java Features in ZEN (cont’d)

- **Asynchrony**
  - Use AsyncEventHandlers for Thread-Pool
  - Data passing to AEHs is not allowed. This makes things complicated.
  - New patterns? 😊
  - Use in HalfSync/HalfAsnch pattern for the Asynch part
Current Status And Future Plans

- Working ORB with RootPOA implementation (GIOP1.0)
- IDL compiler generates valid output
- Interoperable with TAO C++ ORB

Initial Performance metrics
- On dual-cpu 933Mhz P-III (512Mb RAM)
- Debian Linux 2.4.1 Java v1.3 (Blackdown) with JIT.
- Round-trip time: ~700 usecs (loopback) (after 10,000 iters)
- Memory footprint (Server): ~120 KB (stable state)

Future Plans
- RT-ORB, RT-POA, Strategized Threading models
- Dynamic Scheduling
- Reflection: Dynamic fine-tuning/reconfigurability, power management(?)
- Wireless systems applicability
Concluding Remarks

- Many features yet to be implemented in ZEN
- Scope for very interesting research
- RT-CORBA: Dynamic scheduling is still to be standardized
- RT-Java: No implementation available yet for testing
- Distributed-RT Java JSR
URLs for Further Reference

• ZEN web page:
  • http://www.zen.uci.edu
• JacORB web page:
  • http://www.jacorb.org
• RT-Java JSR:
  • http://java.sun.com/aboutJava/communityprocess/jsr/jsr_001_real_time.html
• Dynamic scheduling RFP:
  • http://www.omg.org/techprocess/meetings/schedule/Dynamic_Scheduling_RFP.html
• Distributed RT-Java JSR:
  • http://java.sun.com/aboutJava/communityprocess/jsr/jsr_050_drt.html
• AspectJ web page:
  • http://www.aspectJ.org