Applying Adaptive & Reflective Middleware to Optimize Distributed Embedded Systems

Christopher D. Gill, Washington University
cdgill@cs.wustl.edu

Douglas C. Schmidt, University of California, Irvine
schmidt@uci.edu

Thursday, January 18, 2001
There are multiple COTS layers & research/business opportunities

Historically, mission-critical apps were built directly atop hardware & OS:
- Tedious, error-prone, & costly over lifecycles

Standards-based COTS middleware helps:
- Manage end-to-end resources
- Leverage HW/SW technology advances
- Evolve to new environments & requirements

The domain-specific services layer is where system integrators can provide the most value & derive the most benefits

Key R&D challenges include:
- Layered QoS specification & enforcement
- Separating policies & mechanisms across layers
- Time/space optimizations for middleware & apps
- Layered resource management & optimization
- High confidence
- Stable & robust adaptive systems

Prior R&D programs have address some, but by no means all, of these issues.
Pros & Cons of COTS

Many hardware & software APIs and protocols are now standardized, e.g.:
- Intel x86 & Power PC chipsets
- TCP/IP, ATM
- POSIX & JVMs
- CORBA ORBs & components
- Ada, C, C++, RT Java

COTS standards promote reuse via “narrow-waist” architectures

However, they also limit design choices, e.g.:
- Networking protocols
- Concurrency & scheduling
- Demultiplexing
- Caching
- Fault tolerance
- Security

Historically, COTS tightly couples functional with QoS aspects
- e.g., due to lack of “hooks”
**Promising New Approach:**
Adaptive & Reflective Middleware

**Adaptive & reflective middleware** is middleware whose functional or QoS-related properties can be modified either:
- **Statically**, e.g., to better allocate resources that can optimized *a priori* or
- **Dynamically**, e.g., in response to changes in environment conditions or requirements

**Research Challenges**

**Preserve critical set of application QoS properties end-to-end**
- *e.g.*, efficiency, predictability, scalability, dependability, & security

**Achieve load invariant performance & system stability**

**Maximize longevity** in wireless & mobile environments
- *e.g.*, control power-aware hardware via power-aware middleware

**Automatically generate & integrate multiple QoS properties**
COTS Challenges for Embedded Systems

- **APPLICATIONS**
- **DOMAIN-SPECIFIC SERVICES**
  - Inadequate support for QoS specification & enforcement
  - Inadequate time/space optimizations
  - Inadequate flexibility & customizability

Conventional solutions to this problem are either:
- **Tedious**
  - *e.g.*, reimplement application from scratch
- **Proprietary**
  - *e.g.*, reimplement middleware from scratch
- **Manual & ad hoc**
  - *e.g.*, subset existing COTS middleware
Applying Reflection as an Optimization Technique

To illustrate the benefits of reflection as an optimization technique, consider the evolution of compiler technology:

- **C Program**
  - C Compiler
  - Internal Rep.
    - Ix86
    - VAX
    - 68K

- **C++ Program**
  - C++ Compiler
  - Internal Rep.
    - PPC
    - MIPS
    - 88K

- **Ada Program**
  - Ada Compiler
  - Internal Rep.
    - 1751
    - 32K
    - HPPA
Applying Reflection as an Optimization Technique

- Modern compilers, such as GNU GCC, support
  - A common internal representation (still hand-written) for each programming language
    - Based on generalizing the language semantics
  - A generated optimizer that is customized automatically for each target backend
    - Based on reflective assessment of algebraic target machine description

1. Read the target machine description

2. Use discrimination network to analyze the optimization rules & opportunities

3. Generate an optimizer that is customized for the particular platform/language

Key Benefit of “Static” Reflection
- New targets can be supported by writing a new machine description, rather than writing a new code generator/optimizer
Applying Reflection to Optimize Middleware Statically

Conventional middleware for embedded systems is developed & optimized in a manner similar to early compiler technologies:

- Developing, verifying, validating, & evolving all these components separately is costly, time-consuming, tedious, & error-prone.
- Moreover, it is even harder to hand-configure support for dynamic platform variations & complex application use-cases.
- The problem only gets worse as more middleware, target platforms, & complex applications emerge.

- Separate hand-written & hand-optimized implementations for each embedded target platform, e.g., various OS/network/HW configurations.
- Conventional middleware requires separate tools and interfaces hand-written for each ORB middleware specification, e.g., CORBA, Java RMI, COM+.

Apply Reflection to Optimize Middleware Statically

CORBA Application

<table>
<thead>
<tr>
<th>CORBA ORB &amp; Assorted Tools</th>
<th>WinNT Impl</th>
<th>Solaris Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>VxWorks Impl</td>
<td>WinNT</td>
<td>Solaris</td>
</tr>
<tr>
<td></td>
<td>VxWorks</td>
<td></td>
</tr>
</tbody>
</table>

Java Application

<table>
<thead>
<tr>
<th>Java RMI &amp; Assorted Tools</th>
<th>WinNT Impl</th>
<th>Solaris Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Impl</td>
<td>WinNT</td>
<td>Solaris</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td></td>
</tr>
</tbody>
</table>

COM+ Application

<table>
<thead>
<tr>
<th>COM+ ORB &amp; Assorted Tools</th>
<th>WinNT Impl</th>
<th>Win98 Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>WinCE Impl</td>
<td>WinNT</td>
<td>Win98</td>
</tr>
<tr>
<td></td>
<td>WinCE</td>
<td></td>
</tr>
</tbody>
</table>
Applying Reflection to Optimize Middleware Statically

- The functional and QoS-related aspects of middleware can be improved greatly by advanced R&D on the following topics:
  - A common internal representation (ideally auto-generated) for each middleware specification
    + Based on generalizing the middleware semantics
  - A generated implementation that is optimized automatically for each target platform & application use-case
    + Based on reflective assessment of platform descriptions & application use-case

1. Read the target platform description & application requirements
2. Use discrimination network to analyze the optimization rules & opportunities
3. Generate middleware that is customized for a particular platform & application use-case
Applying Reflection to Optimize Middleware Dynamically

Applying reflection as an optimization is even more relevant to middleware than compilers due to dynamism & global resources:

Key System Characteristics
- Integrate observing & predicting of current status & delivered QoS to inform the meta-layer
- Meta-layer applies reflection to adapt system policies & mechanisms to enhance delivered QoS
Key Research Challenge: Providing QoS Guarantees for Multiple Adaptive Feedback Loops

Goals
- Ensuring stable QoS support at varying granularity & scope levels for integrated, multi-property feedback paths across different locations & time scales
- Determining patterns, protocols, & architectures necessary to integrate COTS components
Concluding Remarks

• Researchers & developers of distributed systems face common challenges, e.g.:
  • Connection management, service initialization, error handling, flow control, event demuxing, distribution, concurrency control, fault tolerance synchronization, scheduling, & persistence

• The application of formal methods along with patterns, frameworks, & components can help to resolve these challenges

• Carefully applying these techniques can yield efficient, scalable, predictable, & flexible middleware & applications