Using RT CORBA in Distributed Real-time Systems

Angelo Corsaro [acorsaro@selex-si.com]
Carlos O’Ryan [coryan@atd.com]
Irfan Pyarali [irfan@oomworks.com]
Douglas Schmidt [d.schmidt@vanderbilt.edu]

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

Caveat: Requirements & Historical Limitations of CORBA for Real-time Systems

Requirements
• Location transparency
• Performance transparency
• Predictability transparency
• Reliability transparency

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

Potential Sources of Priority Inversion
Real-Time CORBA Overview

- RT CORBA adds QoS control to regular CORBA to improve 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 (but by no means all) important real-time application development challenges

An Example DRE Application

- Consider an application where cooperating drones explore a surface & report its properties periodically
  - e.g., color, texture, etc.
- Drones aren’t very “smart,”
  - e.g., they can fall off the “edge” of the surface if not stopped
- Thus, a controller is used to coordinate their actions
  - e.g., it can order them to a new position

Designing the DRE Application

- End-users talk to a Base Station object
  - e.g., they define high-level exploration goals for the drones
  - The Base Station provides set-points for the controllers
- The Controller object controls the drones remotely using Drone objects
- Drone objects are proxies for the underlying drone vehicles
  - e.g., they expose operations for controlling & monitoring individual drone behavior
  - Each drone sends information obtained from its sensors back to the Base Station via a Controller object

Defining Application Interfaces with CORBA IDL

```idl
interface Drone {
    void turn (in float degrees);
    void speed (in short mph);
    void reset_odometer () ;
    short odometer () ;
    // …
};
```

```idl
interface Controller {
    void edge_alarm () ;
    void battery_low () ;
    //…
};
```

```idl
interface Base_Station {
    Controller new_controller (in string name)
        raises (Lack_Resources);
    void set_new_target (in float x, in float y, in float w, in float h);
    //…
};
```

exception Lack_Resources {}
QoS-related Application Design Challenges

- Our example application contains the following QoS-related design challenges
  1. Obtaining portable ORB end-system priorities
  2. Preserving priorities end-to-end
  3. Enforcing certain priorities at the server
  4. Changing CORBA priorities
  5. Supporting thread pools effectively
  6. Buffering client requests
  7. Synchronizing objects correctly
  8. Configuring custom protocols
  9. Controlling network & end-system resources to minimize priority inversion
  10. Avoiding dynamic connections
  11. Simplifying application scheduling
  12. Controlling request timeouts

www.cs.wustl.edu/~schmidt/report-doc.html

Portable End-to-End Priorities

- Problem: How can we map global priorities onto heterogeneous native OS host thread priorities consistently end-to-end?
- Solution: Use Standard RT CORBA priority mapping interfaces

Obtaining Portable ORB End-system Priorities

- OS-independent design supports heterogeneous real-time platforms
- CORBA priorities are “globally” unique values that range from 0 to 32767
- Users can map CORBA priorities into native OS priorities in custom ways
- No silver bullet, but rather an “enabling technique”
- e.g., can’t magically turn a general-purpose OS into a real-time OS!

Preserving Priorities End-to-End

- Problem: How can we ensure requests don’t run at the wrong priority on the server?
  - e.g., this can cause major problems if edge_alarm() operations are processed too late!!!
- Solution: Use RT CORBA priority model policies
Preserving Priorities End-to-End

- **RT CORBA priority model policies**
  - **SERVER_DECLARED**
    - Server handles requests at the priority declared when object was created
  - **CLIENT_PROPAGATED**
    - Request is executed at the priority requested by client
    - Priority is encoded as part of client request

### Changing CORBA Priorities

- **Problem:** How can RT CORBA client application change the priority of operations?
- **Solution:** Use the `RTCCurrent` to change the priority of the current client thread explicitly

### Changing CORBA Priorities at the Client

- **An RTCCurrent object can also be used to query the priority**
  - Values are expressed in the CORBA priority range
  - The behavior of RTCCurrent is thread-specific

```c++
// Get the ORB’s RTCurrent object
obj = orb->resolve_initial_references ("RTCurren");

RTCORBA::Current_var rt_current =
    RTCORBA::Current::narrow (obj);

// Change the current CORBA priority & thread priority
rt_current->the_priority (VERY_HIGH_PRIORITY);

// Invoke the request at <VERY_HIGH_PRIORITY> priority
// The priority is propagated (see previous page)
controller->edge_alarm ();
```

### Thread Pooling

- **Problem:** How can we pre-allocate threading resources on the server portably & efficiently?
  - e.g., the Base_Station must have sufficient threads for all its priority levels
- **Solution:** Use RT CORBA thread pools
RT CORBA Thread Pools

• Pre-allocation of threads
• Partitioning of threads
• Bounding of thread usage
• Buffering of additional requests

Creating & Destroying Thread Pools

interface RTCORBA::RTORB {
  typedef unsigned long ThreadpoolId;
  ThreadpoolId create_threadpool
  (in unsigned long stacksize,
   in unsigned long static_threads,
   in unsigned long dynamic_threads,
   in Priority default_priority,
   in boolean allow_request_buffering,
   in unsigned long max_buffered_requests,
   in unsigned long max_request_buffer_size);
  void destroy_threadpool (in ThreadpoolId
                           threadpool)
                           raises (InvalidThreadpool);
};

Partitioning Thread Pools

• Problem: How can we prevent exhaustion of threads by low priority requests?
  • e.g., many requests to the Base Station methods use up all the threads in the thread pool so
    that no threads for high-priority Controller methods are available

• Solution: Partition thread pool into subsets, which are called lanes, where each lane has a
  different priority

Creating Thread Pools with Lanes

interface RTCORBA::RTORB {
  struct ThreadpoolLane {
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };
  typedef sequence<ThreadpoolLane> ThreadpoolLanes;
  ThreadpoolId create_threadpool_with_lanes
  (in unsigned long stacksize,
   in ThreadpoolLanes lanes,
   in boolean allow_borrowing,
   in boolean allow_request_buffering,
   in unsigned long max_buffered_requests,
   in unsigned long max_request_buffer_size);
};

typedef sequence<ThreadpoolLane> ThreadpoolLanes;
ThreadpoolId create_threadpool_with_lanes
  (in unsigned long stacksize,
   in ThreadpoolLanes lanes,
   in boolean allow_borrowing,
   in boolean allow_request_buffering,
   in unsigned long max_buffered_requests,
   in unsigned long max_request_buffer_size);

It's possible to "borrow" threads from lanes with lower priorities
Configuring Thread Pool Lanes

```cpp
// Define two lanes
RTCORBA::ThreadpoolLane high_priority = {
  10 /* Priority */,
  3 /* Static Threads */,
  0 /* Dynamic Threads */};
RTCORBA::ThreadpoolLane low_priority = {
  5 /* Priority */,
  7 /* Static Threads */,
  2 /* Dynamic Threads */};
RTCORBA::ThreadpoolLanes lanes(2); lanes.length(2);
lanes[0] = high_priority; lanes[1] = low_priority;
RTCORBA::ThreadpoolId pool_id = rt_orb->create_threadpool_with_lanes
(1024 * 10, // Stacksize (10k)
lanes, // Thread pool lanes
false, // No thread borrowing
false, 0, 0); // No request buffering
```

When a thread pool is created it's possible to control certain resource allocations
- e.g., stacksize, request buffering, & whether or not to allow "borrowing" across lanes

When you run out of Threads...

- **Problem:** How can we prevent bursts or long-running requests from exhausting maximum number of static & dynamic threads in the lane?
- **Solution:** Use the Real-time CORBA thread pool lane borrowing feature

Thread Borrowing

- Higher priority lanes can borrow threads from lower priority lanes

### Restoring threads
- Priority is raised when thread is borrowed
- When there are no more requests, borrowed thread is returned & priority is restored

Managing Bursty Requests

- **Problem:** How can we support real-time applications that need more buffering than is provided by the OS I/O subsystem
  - e.g., to handle "burstly" client traffic
- **Solution:** Buffer client requests in ORB
Buffering Client Requests

- RT CORBA thread pool buffer capacities can be configured according to:
  1. Maximum number of bytes and/or
  2. Maximum number of requests

Thread Pool

Thread Pool

Lane

Prio = 100

Prio = 200

Configuring Request Buffering

```c
// Create a thread pool with buffering
RTCORBA::ThreadpoolId pool_id = rt_orb->create_threadpool (1024 * 10, // Stacksize
                                                          4, // Static threads
                                                          10, // Dynamic threads
                                                          DEFAULT_PRIORITY,
                                                          true, // Enable buffering
                                                          128, // Maximum messages
                                                          64 * 1024); // Maximum buffering
```

```c
// Create Thread Pool Policy
RTCORBA::ThreadpoolPolicy_var tp_policy = rt_orb->create_threadpool_policy (pool_id);
```

```c
// Use that policy to configure the RT-POA
```

- Since some RT ORBs don’t use queues to avoid priority inversions, an ORB can reject a request to create a thread pool with buffers
- This design is still compliant, however, since the maximum buffer capacity is always 0
- Moreover, queueing can be done within I/O subsystem of underlying OS

Thread Pools Implementation Strategies

- There are two general strategies to implement RT CORBA thread pools:
  - Use the Leader/Followers pattern to demultiplex I/O events into threads in the pool without requiring additional I/O threads
  - Use the Half-Sync/Half-Async pattern to have I/O thread(s) buffer client requests in a queue & then have worker threads in the pool process the requests

Evaluating Thread Pools Implementations

- RT CORBA spec under-specifies many quality of implementation issues
  - e.g.: Thread pools, memory, & connection management
  - Maximizes freedom of RT CORBA developers
  - Requires application developers to understand ORB implementation
  - Effects schedulability, scalability, & predictability of their application
  - Examine patterns underlying common thread pool implementation strategies
  - Evaluate each thread pool strategy in terms of the following capabilities

<table>
<thead>
<tr>
<th>Capability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature support</td>
<td>Request buffering &amp; thread borrowing</td>
</tr>
<tr>
<td>Scalability</td>
<td>Endpoints &amp; event demultiplexers required</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Data movement, context switches, memory allocations, &amp; synchronizations required</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Stack &amp; thread specific storage memory allocations</td>
</tr>
<tr>
<td>Priority inversion</td>
<td>Bounded &amp; unbounded priority inversion incurred in each implementation</td>
</tr>
</tbody>
</table>
The Half-Sync/Half-Async Pattern

Intent

The Half-Sync/Half-Async architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance.

- This pattern defines two service processing layers—one async and one sync—along with a queueing layer that allows services to exchange messages between the two layers.
- The pattern allows sync services, such as servant processing, to run concurrently, relative both to each other and to async services, such as I/O handling & event demultiplexing.

Queue-per-Lane Thread Pool Design

Design Overview

- Single acceptor endpoint
- One reactor for each priority level
- Each lane has a queue
- I/O & application-level request processing are in different threads

Pros

- Better feature support, e.g.,
- Request buffering
- Thread borrowing
- Better scalability, e.g.,
- Single acceptor
- Fewer reactors
- Smaller IORs
- Easier piece-by-piece integration into the ORB

Cons

- Less efficient because of queueing
- Predictability reduced without _bind_priority_band() implicit operation

Evaluation of Half-Sync/Half-Async Thread Pools

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Support</td>
<td>Good: supports request buffering &amp; thread borrowing</td>
</tr>
<tr>
<td>Scalability</td>
<td>Good: I/O layer resources shared</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Poor: high overhead for data movement, context switches, memory allocations, &amp; synchronizations</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Poor: stack &amp; TSS memory not supported</td>
</tr>
<tr>
<td>Priority Inversion</td>
<td>Poor: some unbounded, many bounded</td>
</tr>
</tbody>
</table>

The Leader/Followers Pattern

Intent: The Leader/Followers architectural pattern provides an efficient concurrency model where multiple threads take turns sharing event sources to detect, demux, dispatch, & process service requests that occur on the event sources.

- TCP Sockets + select() / poll()
- UDP Sockets + select() / poll()
**Reactor-per-Lane Thread Pool Design**

### Design Overview
- Each lane has its own set of resources
  - *i.e.*, reactor, acceptor endpoint, etc.
- I/O & application-level request processing are done in the same thread

### Pros
- Better performance
- No extra context switches
- Stack & TSS optimizations
- No priority inversions during connection establishment
- Control over all threads with standard thread pool API

### Cons
- Harder ORB implementation
- Many endpoints = longer IORs
- Certain features hard to support

---

**Evaluation of Leader/Followers Thread-Pools**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Support</td>
<td>Poor: not easy to support request buffering or thread borrowing</td>
</tr>
<tr>
<td>Scalibility</td>
<td>Poor: I/O layer resources not shared</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Good: little or no overhead for data movement, memory allocations, or synchronizations</td>
</tr>
<tr>
<td>Optimizations</td>
<td>Good: stack &amp; TSS memory supported</td>
</tr>
<tr>
<td>Priority Inversion</td>
<td>Good: little or no priority inversion</td>
</tr>
</tbody>
</table>

---

**Consistent Synchronizers**

- **Problem**: An ORB & application may need to use the same type of mutex to avoid priority inversions
  - *e.g.*, using priority ceiling or priority inheritance protocols
- **Solution**: Use the `RTCORBA::Mutex` synchronizer

---

**Synchronizing Objects Consistently**

- The `RTCORBA::Mutex` interface ensure consistent mutex semantics, across ORB & application domains

```cpp
RTCORBA::Mutex_var mutex = rtorb->create_mutex();
...
mutex->lock(); // Critical section here...
mutex->unlock();
...
rtorb->destroy_mutex (mutex);
```
Network Resource Issues

**Problem:** How can we achieve the following?
- Control jitter due to connection setup
- Minimize thread-level priority inversions
- Avoid request-level ("head-of-line") priority inversions

**Solution:** Use RT CORBA explicit binding mechanisms

Controlling Network Resources

**Connection pre-allocation**
- Eliminates a common source of operation jitter

**Priority Banded Connection Policy**
- Invocation priority determines which connection is used
- **Private Connection Policy**
  - Guarantees non-multiplexed connections

Connection Banding

**Problem:** How can we minimize priority inversions, so that high-priority operations are not queued behind low-priority operations?

**Solution:** Program the client to use different connections for different priority ranges via the RT CORBA PriorityBandedConnectionPolicy

Priority Banded Connection Policy

Note how the `stop()` and `turn()` requests no longer share the same connection as `query_state()` requests.
**Controlling Connection Multiplexing**

- **Problem**: How can we minimize priority inversions by ensuring applications don’t share a connection between multiple objects running at different priorities?
  - *e.g.*, sending a `stop()` request should use exclusive, pre-allocated resources

- **Solution**: Use the RT CORBA `PrivateConnectionPolicy` to guarantee non-multiplexed connections

---

**Private Connection Policy**

Note how the `stop()` and `turn()` requests no longer share the same connection from client to server

```csharp
policies[0] = rtorb->create_private_connection_policy();
CORBA::Object_var object = drone->_set_policy_overrides(policies,
CORBA::ADD_OVERRIDES);
```

---

**Other Relevant CORBA Features**

- RT CORBA leverages other advanced CORBA features to provide a more comprehensive QoS-enabled ORB middleware solution, *e.g.*:
  - **Timeouts**: CORBA Messaging provides policies to control roundtrip timeouts
  - **Reliable oneways**: which are also part of CORBA Messaging
  - **Asynchronous invocations**: CORBA Messaging includes support for type-safe asynchronous method invocation (AMI)
  - **Enhanced views of time**: Defines interfaces to control & query “clocks” (orbos/1999-10-02)
  - **Dynamic Scheduling**: The Joint Submission (orbos/01-06-09) has been accepted & is now part of RT CORBA 1.2
  - **Real-time analysis & scheduling**: The RT CORBA 1.0 Scheduling Service is an optional compliance point for this purpose
    - However, most of the problem is left for an external tool

---

**RT-CORBA 1.x enables the development of statically scheduled, priority based, Distributed Real-Time systems**
RT-CORBA 1.x provides a series of mechanism and policies that allows end-to-end priority preservation and end-to-end resource reservation.

RT-CORBA 1.x does not provide any support for Dynamically Scheduled Distributed Real-Time Systems.

RT-CORBA 2.0

- Extends RT-CORBA 1.x to encompass both dynamic and static systems
- Introduces an End-to-End schedulable entity: the Distributable Thread
- The Specification sets the following goals:
  - Any scheduling discipline might be used
  - The parameters associated with the scheduling discipline might be changed at any time during the execution
  - The schedulable entities is a Distributable Thread that may span node boundaries, and carries scheduling context with it

Distributable Thread

- Object X
- Object Y
- Object Z

ORB A
ORB B
The thread migrates from ORB A to ORB B. It carries scheduling information so to enforce end-to-end QoS.

The thread migrates from ORB B to ORB A. It carries scheduling information so to enforce end-to-end QoS.
Concluding Remarks

- CORBA has become a proved solution for the development of a certain class of distributed real-time systems, *i.e.* priority based statically scheduled systems
- There are many interesting issues that are not addressed in the RT-CORBA 2.0 specification, *i.e.* interoperability, scheduling parameters update, network resource management
- There are already some project working on extending the CCM to support QoS, but more research is needed in this area