Using Real-time CORBA Effectively

Patterns & Principles

Carlos O’Ryan &
Douglas Schmidt
{coryan,schmidt}@uci.edu
University of California, Irvine

Irfan Pyarali
irfan@cs.wustl.edu
Comp. Sci. Dept.
Washington University,
St. Louis

www.cs.wustl.edu/~schmidt/tutorials-corbah.html/

Wednesday, July 18, 2001
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
C. O’Ryan, I. Pyarali, & D. Schmidt

QoS-enabled Middleware

Problems with Current Approaches

• Mission-critical system QoS requirements historically not supported by COTS
  • *i.e.*, COTS is too big, slow, buggy, incapable, & inflexible

• Likewise, the proprietary *multiple technology bases* in systems today limit effectiveness by impeding
  • *Assurability* (of QoS),
  • *Adaptability*, &
  • *Affordability*

Today, each combat system brings its own:

- networks
- computers
- displays
- software
- people

Applications

- Sensor Systems
  - Technology base: Proprietary MW Mercury Link16/11/4

- Command & Control System
  - Technology base: DII-COE POSIX ATM/Ethernet

- Engagement System
  - Technology base: Proprietary MW POSIX NTDS

- Weapon Control Systems
  - Technology base: Proprietary MW VxWorks FDDI/LANs

- Weapon Systems
  - Technology base: Proprietary MW POSIX VME/1553

Operating System

Endsystem

Wireless/Wireline Networks

Endsystem

Problems

• *Non-scalable* tactical performance

• *Inadequate QoS* control for joint operations
  • *e.g.*, distributed weapons control

• *High* software lifecycle costs
  • *e.g.*, many “accidental complexities” & low-level platform dependencies

Today, each combat system brings its own:

- networks
- computers
- displays
- software
- people
A More Effective Approach

Create the new generation of *middleware* technologies for distributed real-time & embedded systems that enable
1. Simultaneous control of *multiple QoS properties* &
2. Composable & customizable *common technology bases*

**Benefits**
- **Highly scalable** performance
  - *e.g.*, distributed resource mgmt.
- Enable **new operational capability**
  - *e.g.*, distributed weapons control
- Support **common technology bases**
  - *e.g.*, elevate *standardization* of COTS to *middleware* to control software lifecycle *costs* by minimizing lower-level dependencies
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
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
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

Real-time CORBA leverages the CORBA Messaging QoS Policy framework
Overview of the CORBA QoS Policy Framework

- CORBA defines a QoS framework that includes policy management for request priority, queueing, message delivery quality, timeouts, etc.
- QoS is managed through interfaces derived from `CORBA::Policy`
  - Each QoS Policy can be queried with its `PolicyType`
- Client-side policies are specified at 3 “overriding levels”
  1. ORB-level through `PolicyManager`
  2. Thread-level through `PolicyCurrent`
  3. Object-level through overrides in an object reference
- Client-side policies are validated via `object::validate_connection()`
- Server-side policies are specified by associating policy objects with POAs
  - *i.e.*, can be passed as arguments to `POA::create_POA()`
  - They may be stored in the tagged components of the IOR
Applying RT CORBA to Real-time Avionics

**Goals**
- Apply COTS & open systems to mission-critical real-time avionics

**Key System Characteristics**
- Deterministic & statistical deadlines
  - ~20 Hz
- Low latency & jitter
  - ~250 usecs
- Periodic & aperiodic processing
- Complex dependencies
- Continuous platform upgrades

**Key Results**
- Test flown at China Lake NAWS by Boeing OSAT II ‘98, funded by OS-JTF
  - www.cs.wustl.edu/~schmidt/TAO-boeing.html
- Also used on SOFIA project by Raytheon
  - sofia.arc.nasa.gov
- First use of RT CORBA in mission computing
- Drove Real-time CORBA standardization
Applying RT CORBA to Time-Critical Targets

Goals
- Detect, identify, track, & destroy time-critical targets

Challenges are also relevant to TBMD & NMD

Key System Characteristics
- Real-time mission-critical sensor-to-shooter needs
- Highly dynamic QoS requirements & environmental conditions
- Multi-service & asset coordination

Key Solution Characteristics
- Adaptive & reflective
- High confidence
- Safety critical
- Efficient & scalable
- Affordable & flexible
- COTS-based

Adapted from "The Future of AWACS", by LtCol Joe Chapa

Joint Forces Global Info Grid
Applying RT CORBA to Hot Rolling Mills

Goals
- Control the processing of molten steel moving through a hot rolling mill in real-time

System Characteristics
- Hard real-time process automation requirements
  - *i.e.*, 250 ms real-time cycles
- System acquires values representing plant’s current state, tracks material flow, calculates new settings for the rolls & devices, & submits new settings back to plant

Key Software Solution Characteristics
- Affordable, flexible, & COTS
  - Product-line architecture
  - Design guided by patterns & frameworks
- Windows NT/2000
- Real-time CORBA

www.siroll.de
Goals
• Examine glass bottles for defects in real-time

System Characteristics
• Process 20 bottles per sec
  • i.e., ~50 msec per bottle
• Networked configuration
  • ~10 cameras

Key Software Solution Characteristics
• Affordable, flexible, & COTS
  • Embedded Linux (Lem)
  • Compact PCI bus + Celeron processors
• Remote booted by DHCP/TFTP
  • Real-time CORBA
An Example Distributed Application

- Consider an application where cooperating drones explore a surface & report its properties periodically
  - e.g., color, texture, etc.
- This is a simplification of various autonomous vehicle use-cases

- 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 Application

- End-users talk to a **Base_Station** object
  - *e.g.*, they define high-level exploration goals for the drones

- The **Base_Station** 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
  - This interaction is an example of **Asynchronous Completion Token & Distributed Callback** patterns
Defining Application Interfaces with CORBA IDL

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

interface Controller {
    void edge_alarm ();
    void battery_low ();
};

exception Lack_Resources {};

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

• Each Drone talks to one Controller
  • e.g., Drones send hi-priority alarm messages when they detect an edge
• The Controller should take corrective action if a Drone detects it’s about to fall off an edge!
• The Base_Station interface is a Controller factory
  • Drones use this interface to create their Controllers during power up
  • End-users use this interface to set high-level mobility targets
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

• The remainder of this tutorial illustrates how these challenges can be addressed by applying RT CORBA capabilities
Obtaining Portable ORB End-system Priorities

- **Problem**: Mapping CORBA priorities to native OS host priorities
- **Solution**: Standard RT CORBA priority mapping interfaces
  - 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 onto native OS priorities in custom ways
  - No silver bullet, but rather an “enabling technique”
    - *i.e.*, can’t magically turn a general-purpose OS into a real-time OS!
Priority Mapping Example

• Define a priority mapping class that always uses native priorities in the range 128-255
  • e.g., this is the top half of LynxOS priorities

```c++
class MyPriorityMapping : public RTCORBA::PriorityMapping {
    CORBA::Boolean to_native (RTCORBA::Priority corba_prio,
                              RTCORBA::NativePriority &native_prio) {
        native_prio = 128 + (corba_prio / 256);
        // In the [128,256) range...
        return true;
    }

    // Similar for CORBA::Boolean to_CORBA ();
};
```

• **Problem**: How do we configure this new class?
• **Solution**: Use TAO’s `PriorityMappingManager`
TAO’s `PriorityMappingManager`

• TAO provides an extension that uses a *locality constrained* object to configure the priority mapping:

```c++
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv); // The ORB

// Get the PriorityMappingManager
CORBA::Object_var obj =
    orb->resolve_initial_references ("PriorityMappingManager");
TAO::PriorityMappingManager_var manager =
    TAO::PriorityMappingManager::_narrow (obj);

// Create an instance of your mapping
RTCORBA::PriorityMapping *my_mapping =
    new MyPriorityMapping;

// Install the new mapping
manager->mapping (my_mapping);
```

• It would be nice if this feature were standardized in RT CORBA…
  • The current specification doesn’t standardize this in order to maximize ORB implemteer options, e.g., link-time vs. run-time bindings
Preserving Priorities End-to-End

**Problem**: Requests could 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
- **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 encoded as part of client request)
Applying **CLIENT_PROPAGATED**

- Drones send critical messages to **Controllers** in the **Base_Station**
- **edge_alarm()** runs at the highest priority in the system
- **Battery_low()** runs at a lower priority in the system

```c++
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = rtorb->create_priority_model_policy
  (RTCORBA::CLIENT_PROPAGATED,
   DEFAULT_PRIORITY /* For non-RT ORBs */);

// Create a POA with the correct policies
PortableServer::POA_var controller_poa =
  root_poa->create_POA ("Controller_POA",
    PortableServer::POAManager::_nil (),
    policies);

// Activate one Controller servant in <controller_poa>
controller_poa->activate_object (my_controller);
...
// Export object reference for <my_controller>
```

- Note how **CLIENT_PROPAGATED** policy is set on the server & exported to the client along with an object reference
Changing CORBA Priorities at the Client

- **Problem**: How can RT-CORBA client applications change the priority of operations?
- **Solution**: Use the `RTCCurrent` to change the priority of the current thread explicitly
  - An `RTCCurrent` can also be used to query the priority
  - Values are expressed in the `CORBA priority` range
  - Behavior of `RTCCurrent` is thread-specific

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

RTCORBA::RTCCurrent_var rt_current = 
    RTCORBA::RTCCurrent::_narrow (obj);

// Change the current CORBA 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 ();
```
Design Interlude: The RTORB Interface

**Problem:** How can the ORB be extended without changing the CORBA::ORB API?

**Solution:** Use the *Extension Interface* pattern from POSA2

- Use `resolve_initial_references()` interface to obtain the extension
- Thus, non real-time ORBs and applications are not affected by RT CORBA enhancements!

```cpp
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

CORBA::Object_var obj =
    orb->resolve_initial_references ("RTORB");

RTCORBA::RTORB_var rtorb =
    RTCORBA::RTORB::_narrow (obj);

// Assuming this narrow succeeds we can henceforth use RT
// CORBA features
```
Applying **SERVER_DECLARED**

**Problem:** Some operations must always be invoked at a fixed priority
- *e.g.*, the `Base_Station` methods are not time-critical, so they should always run at lower priority than the `Controller` methods

**Solution:** Use the RT CORBA **SERVER_DECLARED** priority model

```cpp
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = rtorb->create_priority_model_policy
  (RTCORBA::SERVER_DECLARED, LOW_PRIORITY);

// Create a POA with the correct policies
PortableServer::POA_var base_station_poa =
  root_poa->create_POA ("Base_Station_POA",
    PortableServer::POAManager::_nil (),
    policies);

// Activate the <Base_Station> servant in <base_station_poa>
base_station_poa->activate_object (base_station);
```

- By default, **SERVER_DECLARED** objects inherit the priority of their RTPOA
- It’s possible to override this priority on a per-object basis, however!
Supporting Thread Pools Effectively

**Problem:** Pre-allocating 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 to configure server POAs to support
- Different levels of service
- Overlapping of computation & I/O
- Priority partitioning
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);
};

These are factory methods for controlling the life-cycle of RT-CORBA thread pools
Installing Thread Pools on a RT-POA

// From previous page
RTCORBA::ThreadpoolId pool_id = // ...

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

// Create policy list for RT-POA
CORBA::PolicyList RTPOA_policies(1);
RTPOA_policies.length (1);
RTPOA_policies[0] = tp_policy;

// Create POAs
PortableServer::POA_var rt_poa_1 =
    root_poa->create_POA ("RT-POA_1", // POA name
        PortableServer::POAManager::_nil (),
        RTPOA_policies);  // POA policies

PortableServer::POA_var rt_poa_2 =
    root_poa->create_POA ("RT-POA_2", // POA name
        PortableServer::POAManager::_nil (),
        RTPOA_policies);  // POA policies
Extended RT POA Interface

• RT CORBA extends the POA interface via inheritance

```plaintext
module RTPortableServer {
    local interface POA : PortableServer::POA {
        PortableServer::ObjectId
        activate_object_with_priority
            (in PortableServer::Servant servant_ptr,
             in RTCORBA::Priority priority)
        raises (ServantAlreadyActive,
                WrongPolicy);
    }
}

// ...
```

• Methods in this interface can override default SERVER_DECLARED priorities

```plaintext
// Activate object with default priority of RTPOA
MyBase_Station *station = new MyBase_Station;
base_station_poa->activate_object (station);

// Activate another object with a specific priority
RTPortableServer::POA_var rt_poa =
    RTPortableServer::POA::_narrow (base_station_poa);
rt_poa->activate_object_with_priority (another_servant,
                                       ANOTHER_PRIORITY);
```
Creating Thread Pools with Lanes

• **Problem**: 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*, each lane has a different priority

```c
interface RTCORBA::RTORB {
  struct ThreadpoolLane {
    Priority lane_priority;
    unsigned long static_threads;
    unsigned long dynamic_threads;
  };
  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

// 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
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
Buffering Client Requests

• **Problem**: Some types of applications need more buffering than is provided by the OS I/O subsystem
  - *e.g.*, to handle “bursty” client traffic

• **Solution**: Buffer client requests in ORB

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

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

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

// 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 the I/O subsystem of the OS
Thread Borrowing

- **Problem**: Request bursts or long running requests may exhaust maximum number of static and dynamic threads in the lane.
- **Solution**: Lane borrows thread from a lower priority lane.

**Restoring threads**
- Priority is raised when thread is borrowed.
- When there are no more requests, borrowed thread is returned and priority is restored.
Thread Pools Implementation Strategies

• There are two general strategies to implement RT CORBA thread pools:
  1. 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
  2. Use the *Leader/Followers* pattern to demultiplex I/O events into threads in the pool *without* requiring additional I/O threads

• Each strategy is appropriate for certain application domains
  • *e.g.*, certain hard-real time applications cannot incur the non-determinism & priority inversion of additional request queues

• To evaluate each approach we must understand their consequences
  • Their pattern descriptions capture this information
  • Good metrics to compare RT-CORBA implementations
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 queuing
- Predictability reduced without `_bind_priority_band()` implicit operation
**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.

<table>
<thead>
<tr>
<th>Handles</th>
<th>Concurrent Handles</th>
<th>Iterative Handles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle Sets</td>
<td>UDP Sockets + WaitForMultipleObjects()</td>
<td>TCP Sockets + WaitForMultipleObjects()</td>
</tr>
<tr>
<td>Concurrent Handle Sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterative Handle Sets</td>
<td>UDP Sockets + select()/poll()</td>
<td>TCP Sockets + select()/poll()</td>
</tr>
</tbody>
</table>
**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 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
Synchronizing Objects Consistently

• **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 interface to ensure that consistent mutex semantics are enforced across ORB & application domains

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

`create_mutex()` is a factory method
Configuring Custom Protocols

**Problems:** Selecting communication protocol(s) is crucial to obtaining QoS
- TCP/IP is inadequate to provide end-to-end *real-time* response
- Thus, communication between **Base_Station**, **Controllers**, & **Drones** must use a different protocol
- Moreover, some messages between **Drone** & **Controller** cannot be delayed

**Solution:** Protocol selection policies
- Both server-side & client-side policies are supported
- Some policies control protocol selection, others configuration
- Order of protocols indicates protocol preference
- Some policies are exported to client in object reference

Ironically, RT-CORBA specifies only protocol properties for TCP!
Example: Configuring protocols

• First, we create the protocol properties

```cpp
RTCORBA::ProtocolProperties_var tcp_properties =
    rtorb->create_tcp_protocol_properties (64 * 1024, /* send buffer */
                                           64 * 1024, /* recv buffer */
                                           false, /* keep alive */
                                           true, /* dont_route */
                                           true /* no_delay */);
```

• Next, we configure the list of protocols to use

```cpp
RTCORBA::ProtocolList plist; plist.length (2);
plist[0].protocol_type = MY_PROTOCOL_TAG;
plist[0].trans_protocol_props =
    /* Use ORB proprietary interface */
plist[1].protocol_type = IOP::TAG_INTERNET_IOP;
plist[1].trans_protocol_props = tcp_properties;
RTCORBA::ClientProtocolPolicy_ptr policy =
    rtorb->create_client_protocol_policy (plist);
```
C. O’Ryan, I. Pyarali, & D. Schmidt
QoS-enabled Middleware

Controlling Network Resources

• **Problems:**
  • Control jitter due to connection setup
  • Avoiding request-level (“head-of-line”) priority inversions
  • Minimizing thread-level priority inversions

• **Solution:** Use explicit binding mechanisms, *e.g.,*
  • **Connection pre-allocation**
    • Eliminates a common source of operation jitter
  • **Private Connection Policy**
    • Guarantees non-multiplexed connections
  • **Priority Banded Connection Policy**
    • Invocation priority determines which connection is used

```
stop() turn() query_state()

prio 200

OBJ REF

prio 200

prio 100

stop() turn() query_state()

ORB CORE
```
Pre-allocating Network Connections

• **Problem:** Dynamically establishing connections from the base station to/from the drones can result in unacceptable jitter, which can be detrimental to time-critical applications

• **Solution:** Pre-allocate one or more connections using the `Object::_validate_connection()` operation, which is defined in the CORBA Message specification

```cpp
Drone_var drone = ...; // Obtain reference to a drone

// Pre-establish connections using current policy overrides
CORBA::PolicyList_var inconsistent_policies;

// The following operation causes a _bind_priority_band()
// “implicit” request to be sent to the server
CORBA::Boolean successful =
    drone->_validate_connection (inconsistent_policies);
```
Priority Banded Connection Policy

**Problem:** To minimize priority inversions, high-priority operations should not be queued behind low-priority operations.

**Solution:** Use different connections for different priority ranges via the RT CORBA PriorityBandedConnectionPolicy.

```
// Create the priority bands.
RTCORBA::PriorityBands bands (2);
bands.length (2);
// We can have bands with a range
// of priorities...
bands[0].low  = 0;
bands[0].high = 150;
// ... or just a “range” of 1!
bands[1].low  = 200;
bands[1].high = 200;

CORBA::Policy_var policy = rtorb->
create_priority_banded_connection_policy (bands);
```
Private Connection Policy

**Problem**: To minimize priority inversions, some applications cannot share a connection between multiple objects
- *e.g.*, sending a `stop()` request should use exclusive, pre-allocated resources

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

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

**Problem:** Although RT-CORBA gives developers control over system resources it has two deficiencies:
- It can be tedious to configure all the various policies
- Application developer must select the right priority values

**Solution:** Apply the RT-CORBA Scheduling Service to simplify application scheduling
- Developers just declare the current *activity*
  - *i.e.*, a named chain of requests scheduled by the infrastructure
- Properties of an activity are specified using an (unspecified) external tool
- Note that the Scheduling Service is an optional part of the RT-CORBA 1.0 specification

```c++
// Find the scheduling service
RTCosScheduling::ClientScheduler_var scheduler = ... ;

// Schedule the `edge_alarm` activity
scheduler->schedule_activity ("edge_alarm");

controller->edge_alarm ();
```

The client-side programming model is simple
Server-side Scheduling

// Obtain a reference to the scheduling service
RTCosScheduling::ServerScheduler_var scheduler = ... ;

CORBA::PolicyList policies; // Set POA policies

// The scheduling service configures the RT policies
PortableServer::POA_var rt_poa = scheduler->create_POA
   ("ControllerPOA",
    PortableServer::POAManager::_nil (),
    policies);

// Activate the servant, and obtain a reference to it.
rt_poa->activate_servant (my_controller);
CORBA::Object_var controller =
   rt_poa->servant_to_reference (my_controller);

// Configure the resources required for this object
// e.g., setup interceptors to control priorities
scheduler->schedule_object (controller, "CTRL_000");
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)

  • **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

  • **Enhanced views of time**: Defines interfaces to control & query “clocks” (orbos/1999-10-02)

  • **RT Notification Service**: Currently in progress in the OMG (orbos/00-06-10), looks for RT-enhanced Notification Service

  • **Dynamic Scheduling**: Currently in progress in the OMG (orbos/98-02-15) to address additional policies for dynamic & hybrid static/dynamic scheduling
Controlling Request Timeouts

• **Problem**: Our **Controller** object should not block indefinitely when trying to stop a drone that’s fallen off an edge!

• **Solution**: Override the timeout policy in the **Drone** object reference

```cpp
// 10 milliseconds (base units are 100 nanosecs)
CORBA::Any val; val <<= TimeBase::TimeT (100000UL);

// Create the timeout policy
CORBA::PolicyList policies (1); policies.length (1);
policies[0] = orb->create_policy
   (Messaging::RELATIVE_RT_TIMEOUT_POLICY_TYPE, val);

// Override the policy in the drone
CORBA::Object_var obj = drone->_set_policy_overrides
   (policies, CORBA::ADD_OVERRIDE);

Drone_var drone_with_timeout = Drone::_narrow (obj);
try { drone_with_timeout->speed (0); }
catch (const CORBA::TIMEOUT &e) { // Handle exception }
```
Reliable Oneways

• **Problem**: Traditional CORBA one-way operation semantics are not precise enough for real-time applications

• **Solution**: Use the SyncScope policy to control one-way semantics
Asynchronous Method Invocation

- **Problem**: clients block waiting for requests to complete, yet some requests take too long, or many requests must be issued simultaneously
- **Solution**: use the AMI interfaces to separate (in time and space) the thread issuing the request from the thread processing the reply
Open Issues with the Real-Time CORBA Specification

1. No standard APIs for setting & getting priority mappings & priority transforms
2. No compelling use-cases for server-set client protocol policies
3. Semantic ambiguities
   • Valid policy configurations & their semantics
     • *e.g.*, should a protocol property affect all endpoints or just some?
   • Resource definition & allocation
   • Mapping of threads to connection endpoints on the server
4. The bounds on priority inversions is a quality of implementation
   • No requirement for I/O threads to run at the same priority as request processing threads

**Bottom-line:** RT CORBA applications remain dependant on implementation details
Additional Information

• CORBA 2.4 specification (includes RT-CORBA)
  • www.omg.org/technology/documents/formal/corbaiiop.htm

• Patterns for concurrent & networked objects
  • www.posa.uci.edu

• ACE & TAO open-source middleware
  • www.cs.wustl.edu/~schmidt/ACE.html
  • www.cs.wustl.edu/~schmidt/TAO.html

• CORBA research papers
  • www.cs.wustl.edu/~schmidt/corba-research.html

• CORBA tutorials
  • www.cs.wustl.edu/~schmidt/tutorials-corba.html
Concluding Remarks

• RT CORBA 1.0 is a major step forward for QoS-enabled middleware
  - e.g., it introduces important capabilities to manage key ORB end-system/network resources

• We expect that these new capabilities will increase interest in--and applicability of--CORBA for distributed real-time & embedded systems

• RT CORBA 1.0 doesn’t solve *all* real-time development problems, however
  - It lacks important features:
    - Standard priority mapping manager
    - Dynamic scheduling
      - Addressed in RT CORBA 2.0
  - Portions of spec are under-specified
    - Thus, developers must be familiar with the implementation decisions made by their RT ORB

• Our work on TAO has helped advance middleware for distributed real-time & embedded systems by implementing RT CORBA in an open-source ORB & providing feedback to users & OMG