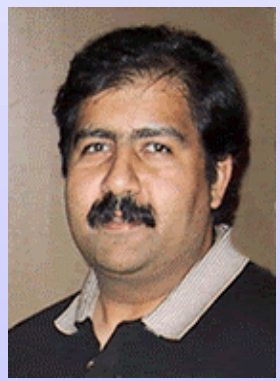
A Guided Tour of Real-time CORBA
Part 2

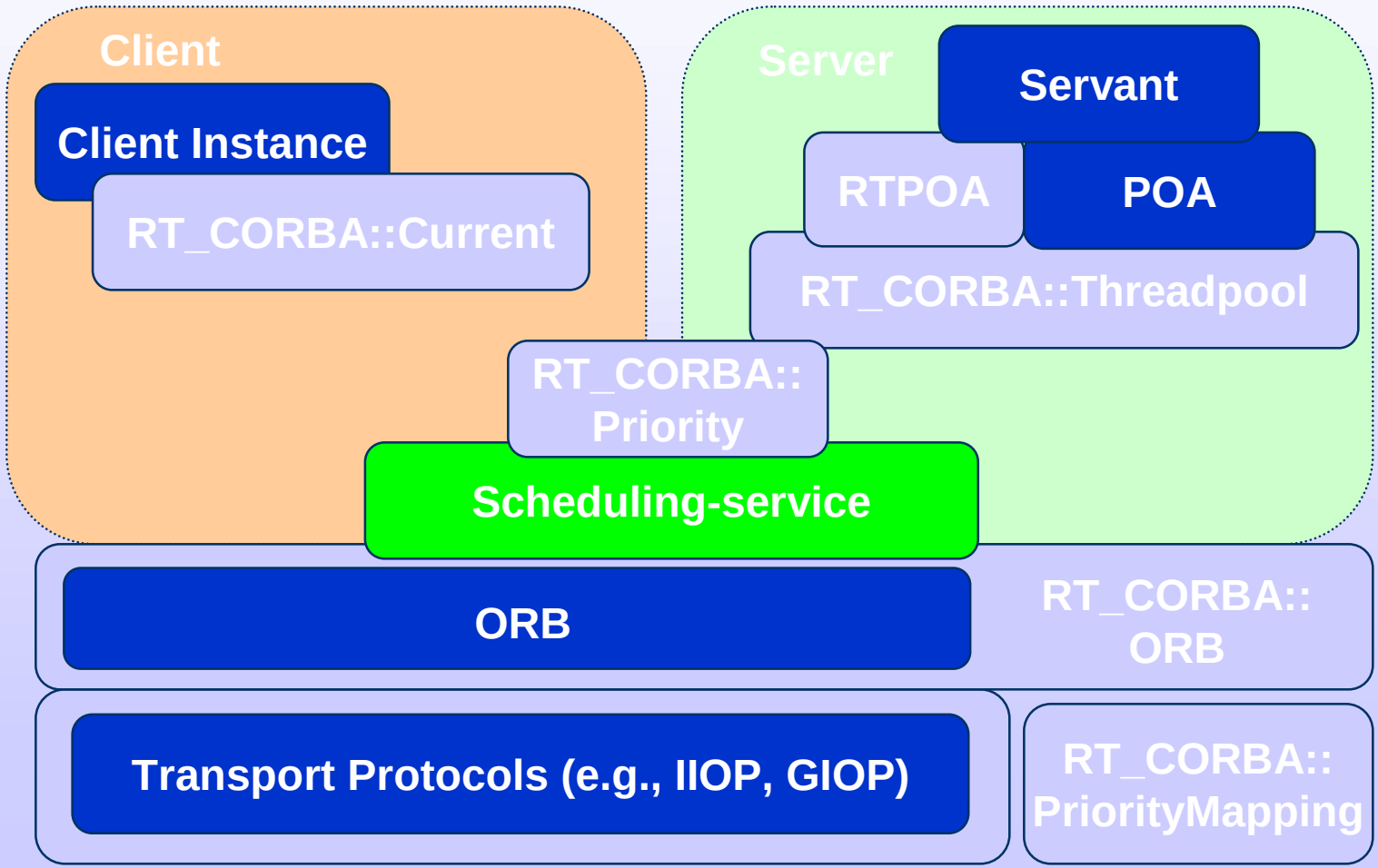
Dr. Shahzad Aslam Mir
Chief Technology Officer
PrismTech Corporation
sam@prismtechnologies.com
www.prismtechnologies.com



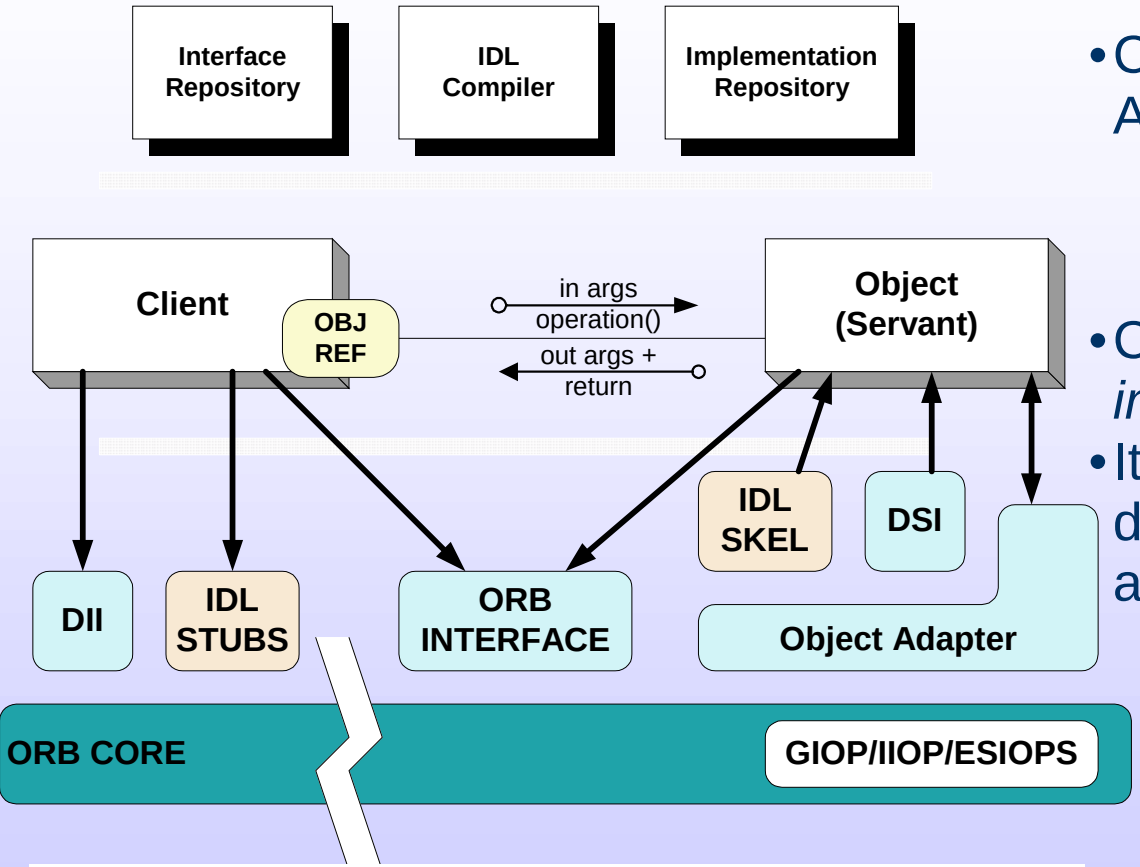
A Guided Tour of Real-time CORBA

Part 2a
Using RT-CORBA in fixed priority statically scheduled systems

Key RT-CORBA1.0 Pieces



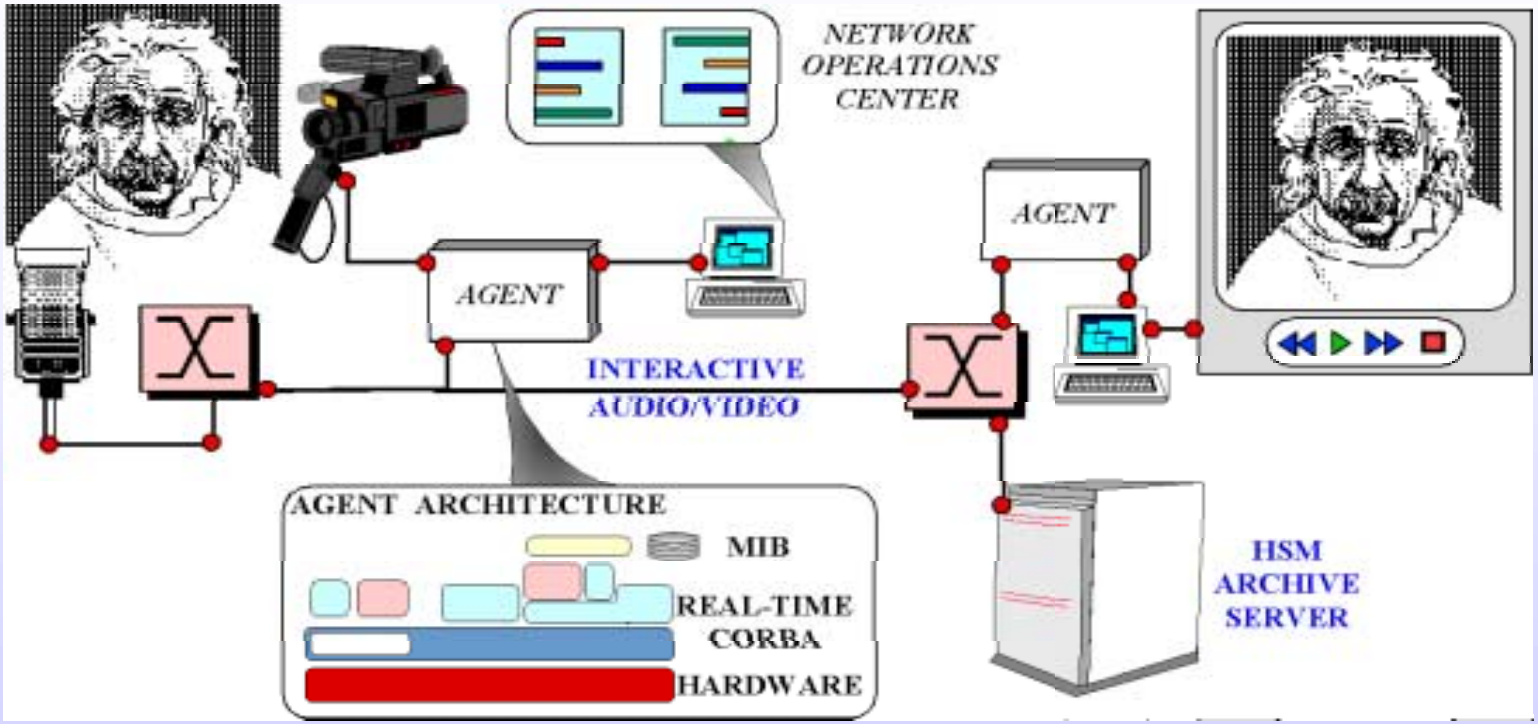
Overview of CORBA



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

- 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



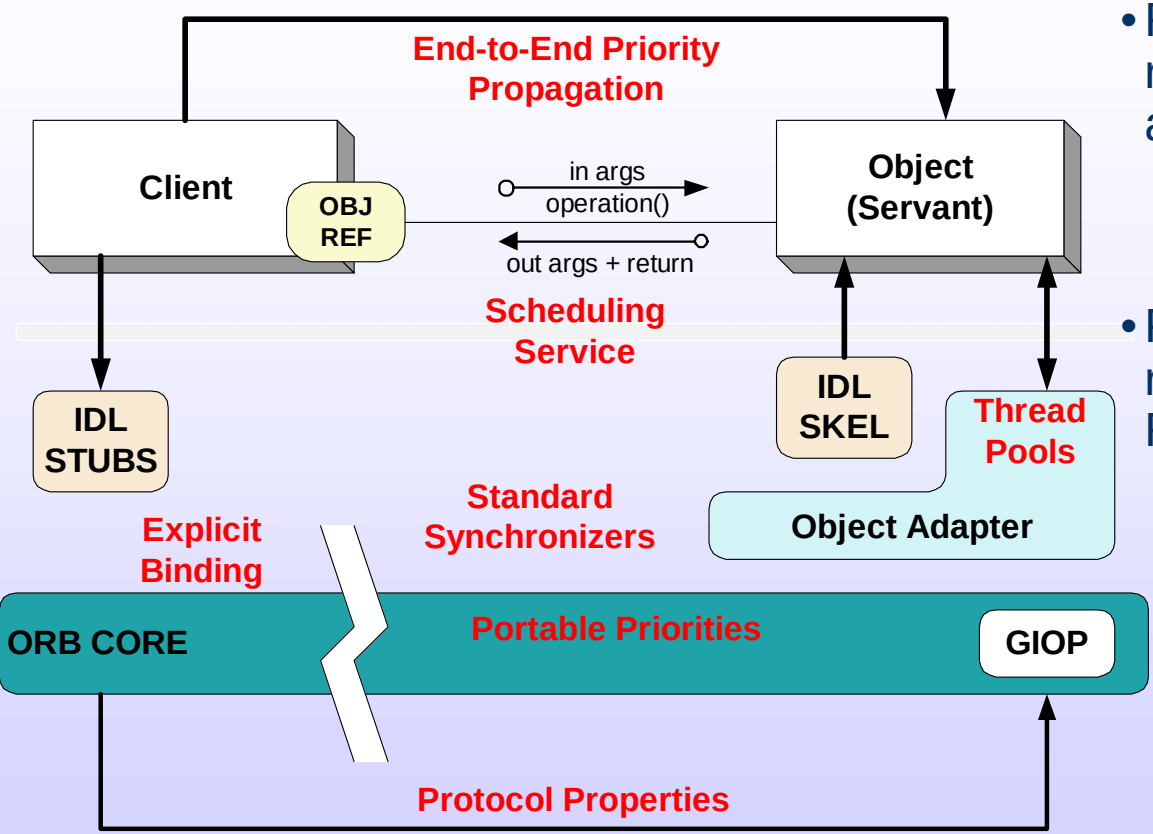
Requirements

- 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

Real-Time CORBA Overview



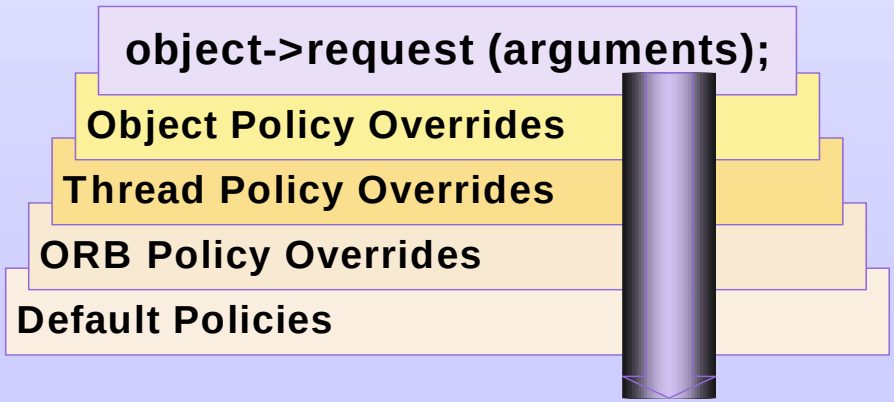
Real-time CORBA leverages the CORBA Messaging QoS Policy framework

- 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

Overview of the CORBA QoS Policy Framework

- CORBA defines a QoS framework that includes policy management for *request priority, queueing, message delivery quality, timeouts, connection management, etc*.
  - QoS is managed through interfaces derived from **`CORBA::Policy`**
    - Each QoS Policy can be queried with its **`PolicyType`**

- Client-side policies can be specified at 3 "overriding levels"
  1. ORB-level via **`PolicyManager`**
  2. Thread-level via **`PolicyCurrent`**
  3. Object-level via *overrides* in an object reference

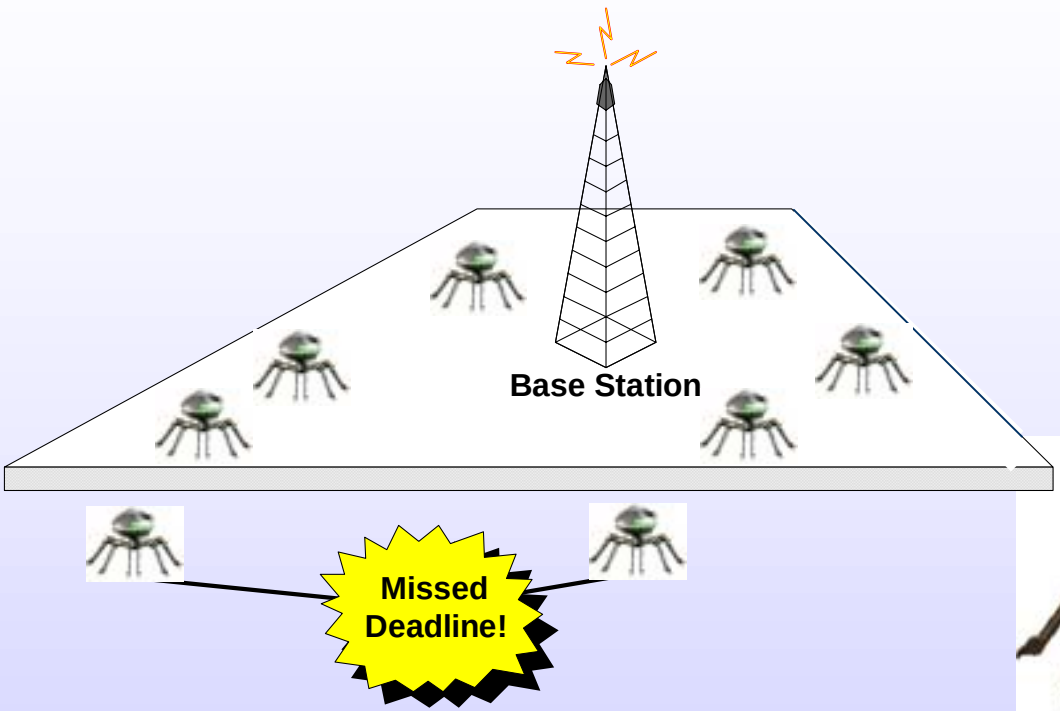


- Server-side policies can be specified at 3 overriding levels
  1. ORB-level through **`PolicyManager`**
  2. POA-level passed as arguments to **`POA::create_POA()`**
  3. Some policies can be set at the Object-level through the **`RTPOA`**
- Server-side policies can be stored in the tagged components of the IOR

IIOP VERSION	HOST	PORT	OBJECT KEY	TAGGED COMPONENTS

- Client-side policies are validated via **`Object::_validate_connection()`**

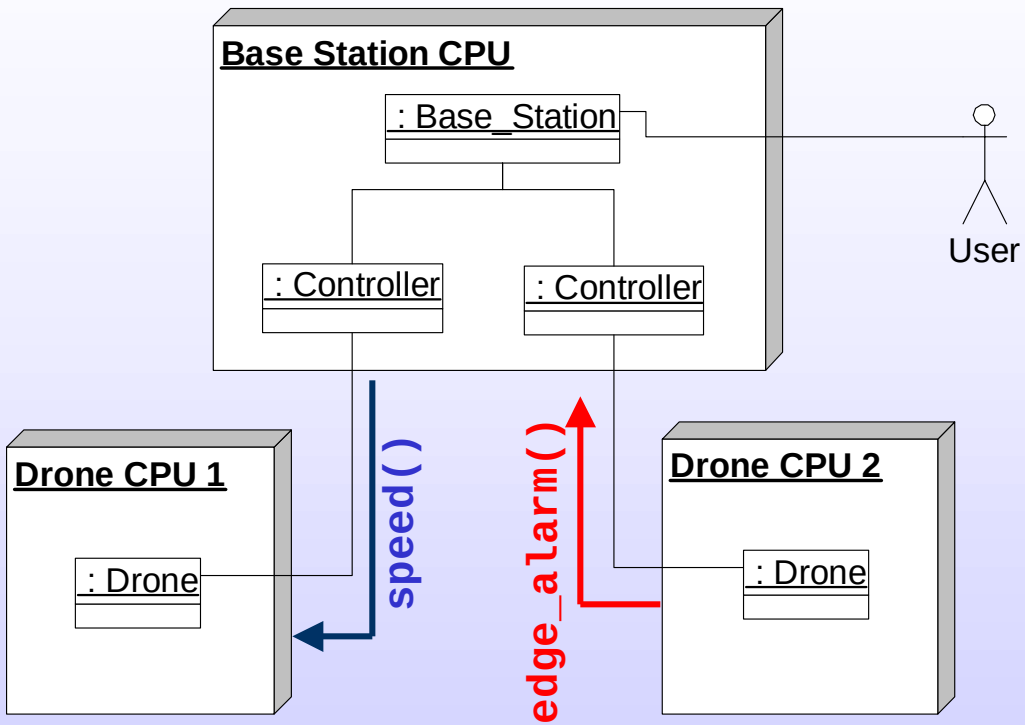
An Example DRE Application



- 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

```
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 ();
  //…
};
```

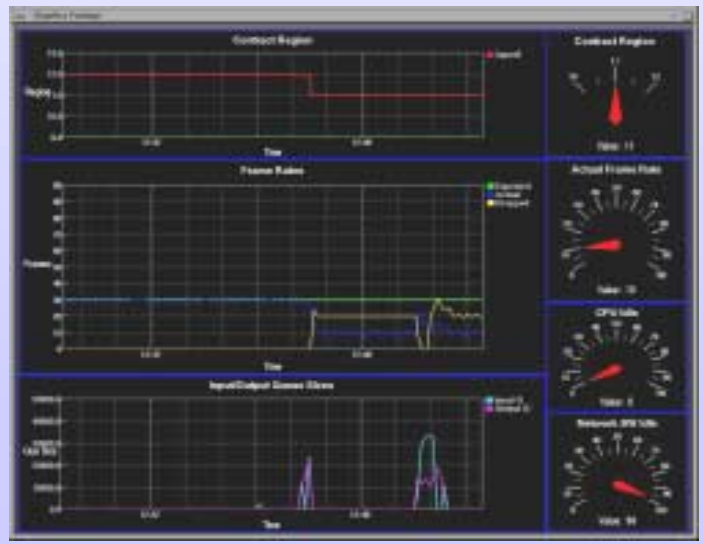
```
interface Base_Station {
  Controller new_controller
       (in string name)
    raises (Lack_Resources);
  void set_new_target
       (in float x, in float y,
        in float w, in float h);
  //……
};
exception Lack_Resources {};
```

- Each **`Drone`** talks to a **`Controller`**
  - *e.g.*, **`Drones`** send hi-priority **`edge_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

This API is a simplification of various semi-autonomous vehicle use-cases

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*

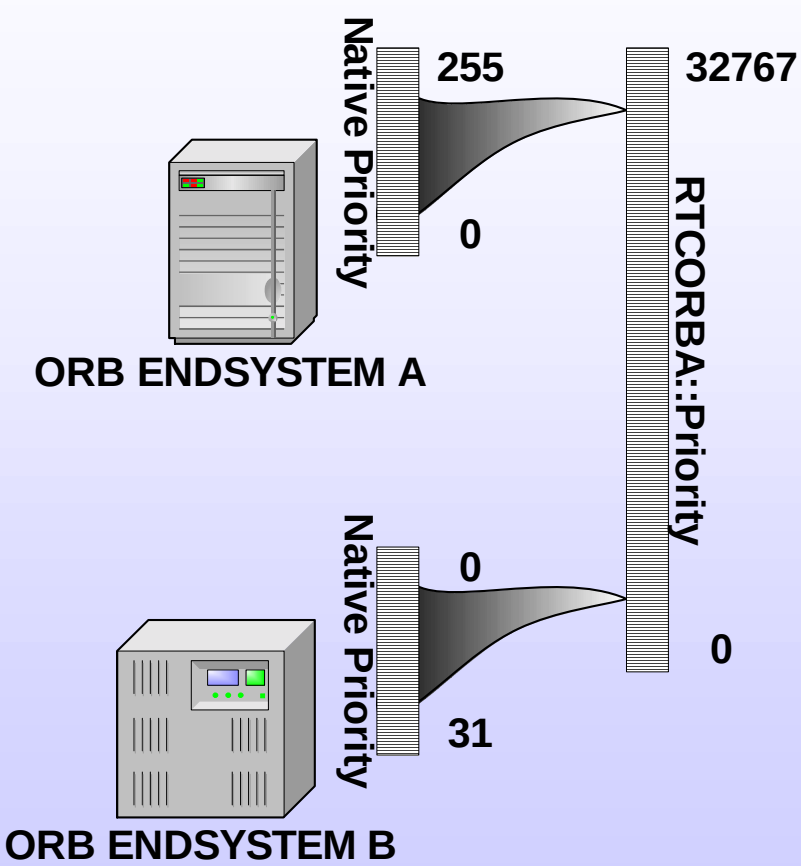


www.cs.wustl.edu/~schmidt/report-doc.html

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 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 class to map CORBA priorities to native OS priorities & vice versa

```
class My_Priority_Mapping : public RTCORBA::PriorityMapping {
  CORBA::Boolean to_native
       (RTCORBA::Priority corba_prio,
        RTCORBA::NativePriority &native_prio) {
    // Only use native priorities in the range [128-255), e.g.
    // this is the top half of LynxOS thread priorities.
    native_prio = 128 + (corba_prio / 256);
    return true;
  }

  CORBA::Boolean to_corba (RTCORBA::NativePriority native_prio,
                           RTCORBA::Priority &corba_prio) {
    if (native_prio < 128)
       return false;

    corba_prio = (native_prio - 128) * 256;
    return true;
  }
};
```

Setting Custom Priority Mapping

- **Problem:** How do we configure the `PriorityMapping` that the ORB should use?
- **Solution:** Use TAO's `PriorityMappingManager`!

TAO's PriorityMappingManager

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

```
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 My_Priority_Mapping;

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

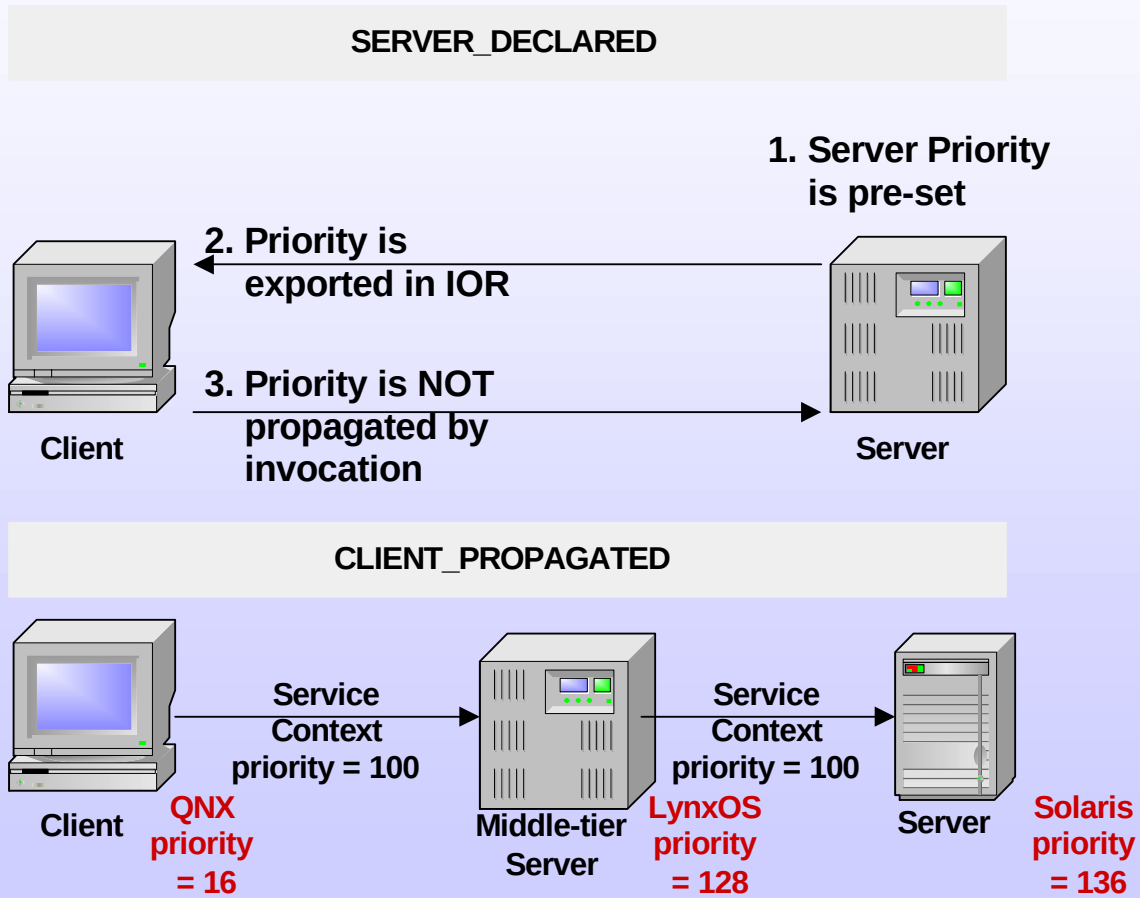
- It would be nice if this feature were standardized in RT CORBA…
  - The current specification doesn't standardize this in order to maximize ORB implementer options, *e.g.*, link-time vs. run-time bindings

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



Applying **CLIENT_PROPAGATED** on Server

- 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

```
CORBA::PolicyList policies (1);
policies.length (1);

policies[0] = rtorb->create_priority_model_policy
   (RTCORBA::CLIENT_PROPAGATED,
    DEFAULT_PRIORITY); // Default when client is non-RT ORB

// 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

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

Changing CORBA Priorities at the Client

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

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

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 ();
```

Design Interlude: The RT-ORB Interface

- **Problem:** How can an ORB be extended to support RT-CORBA *without* changing the `CORBA::ORB` interface?

- **Solution:** Use *Extension Interface* pattern from POSA2 book <www.posa.uci.edu>
  - Use `resolve_initial_references()` interface to obtain the extension
  - Thus, non real-time ORBs and applications are not affected by RT CORBA enhancements!

Getting the RTORB Extension Interface

```
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 that <_narrow> succeeds we can henceforth use RT
// CORBA features
```

- The **`resolve_initial_references()`** method takes a string representing the desired *extension interface*
- It either returns an object reference to the requested extension interface or it returns nil

Enforcing CORBA Priorities

- **Problem:** How to ensure that certain operations always run 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

Applying **SERVER_DECLARED** on Server

- By default, **SERVER_DECLARED** objects inherit the priority of their **RTPOA**
  - As shown later, this priority can be overridden on a per-object basis!

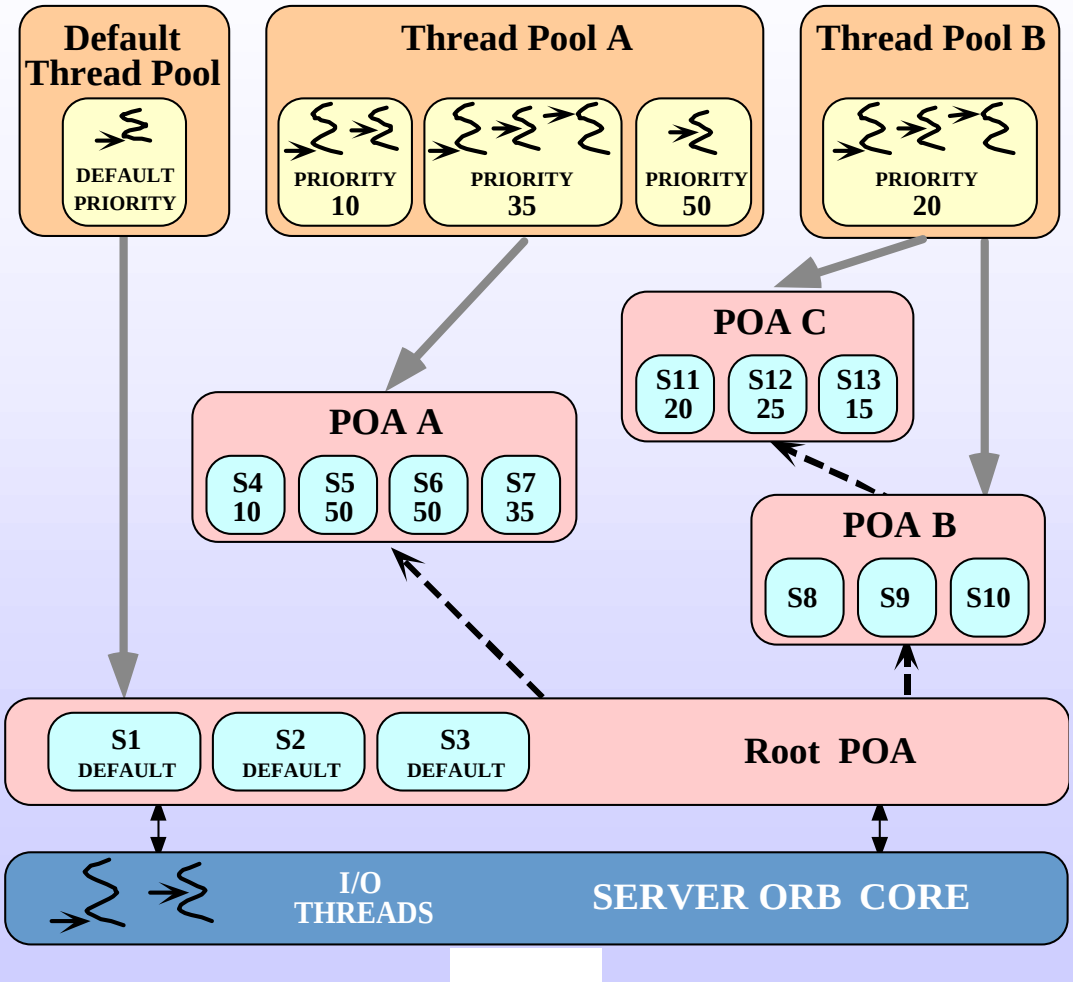
```
CORBA::PolicyList policies (1);
policies.length (1);

policies[0] = rtorb->create_priority_model_policy
   (RTCORBA::SERVER_DECLARED,
    LOW_PRIORITY); // Default priority of activated objects

// 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);
```

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

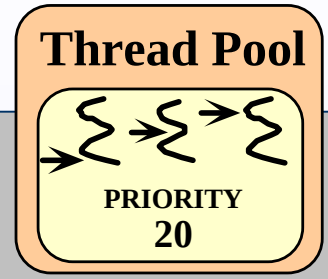
- **Solution:** Use RT CORBA thread pools

RT-CORBA Thread Pools



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

Creating & Destroying Thread Pools

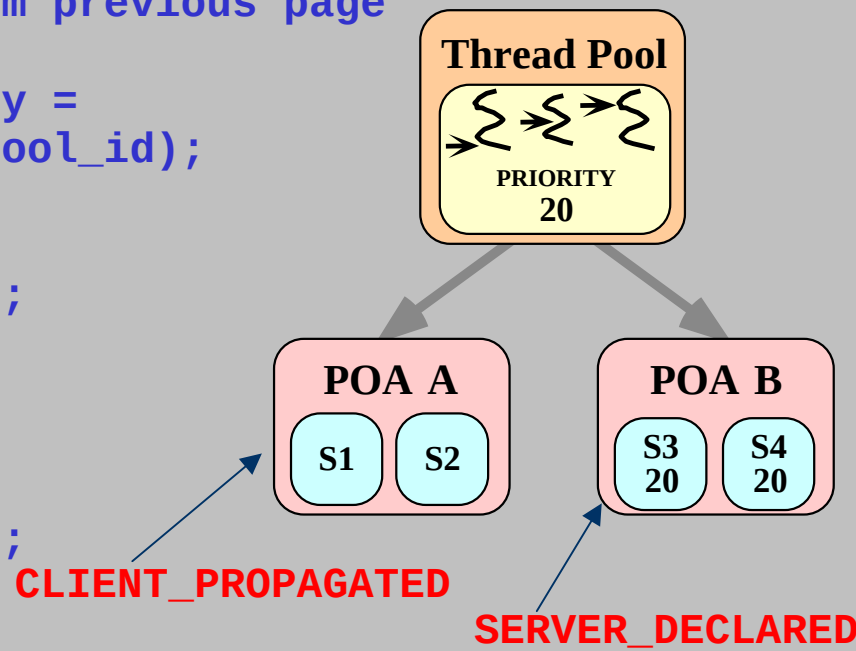



```
interface RTCORBA::RTORB {
   typedef unsigned long ThreadpoolId;

   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);
};
```

These are factory methods for controlling the life-cycle of RT-CORBA thread pools

Installing Thread Pools on an RT-POA

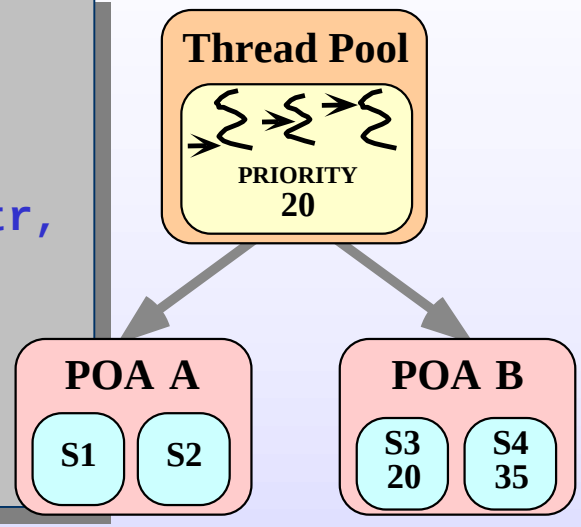
```
RTCORBA::ThreadpoolId pool_id = // From previous page
// Create Thread Pool Policy
RTCORBA::ThreadpoolPolicy_var tp_policy =
   rt_orb->create_threadpool_policy (pool_id);

// Create policy lists for RT-POAs
CORBA::PolicyList RTPOA_policies_a (2);
RTPOA_policies_a.length (2);
RTPOA_policies_a[0] = tp_policy;
RTPOA_policies_a[1] =
    // Set CLIENT_PROPAGATED policy...
CORBA::PolicyList RTPOA_policies_b (2);
RTPOA_policies_b.length (2);
RTPOA_policies_b[0] = tp_policy;
RTPOA_policies_b[1] = // Set SERVER_DECLARED policy...

// Create the RT-POAs
PortableServer::POA_var rt_poa_a = root_poa->create_POA
  ("POA A", PortableServer::POAManager::_nil (), RTPOA_policies_a);
PortableServer::POA_var rt_poa_b = root_poa->create_POA
  ("POA B", PortableServer::POAManager::_nil (), RTPOA_policies_b);
```



Extended RT POA Interface

- RT CORBA extends the POA interface via inheritance

```
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);
    // ...
  };
```



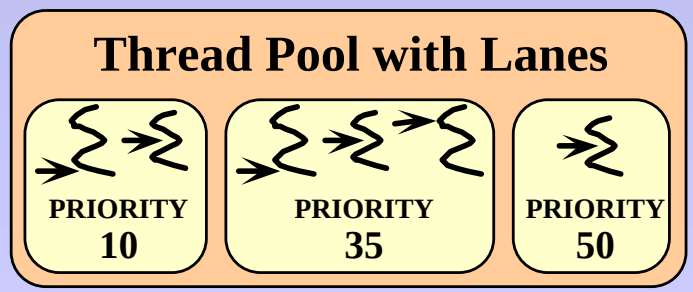
- Methods in this interface can override default **SERVER_DECLARED** priorities

```
    // Activate object with default priority of RTPOA
    My_Base_Station *station = new My_Base_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);
```

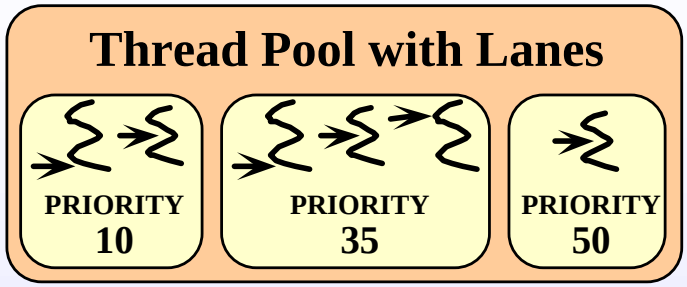
Partitioning Thread Pools

- **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);
};
```

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
```

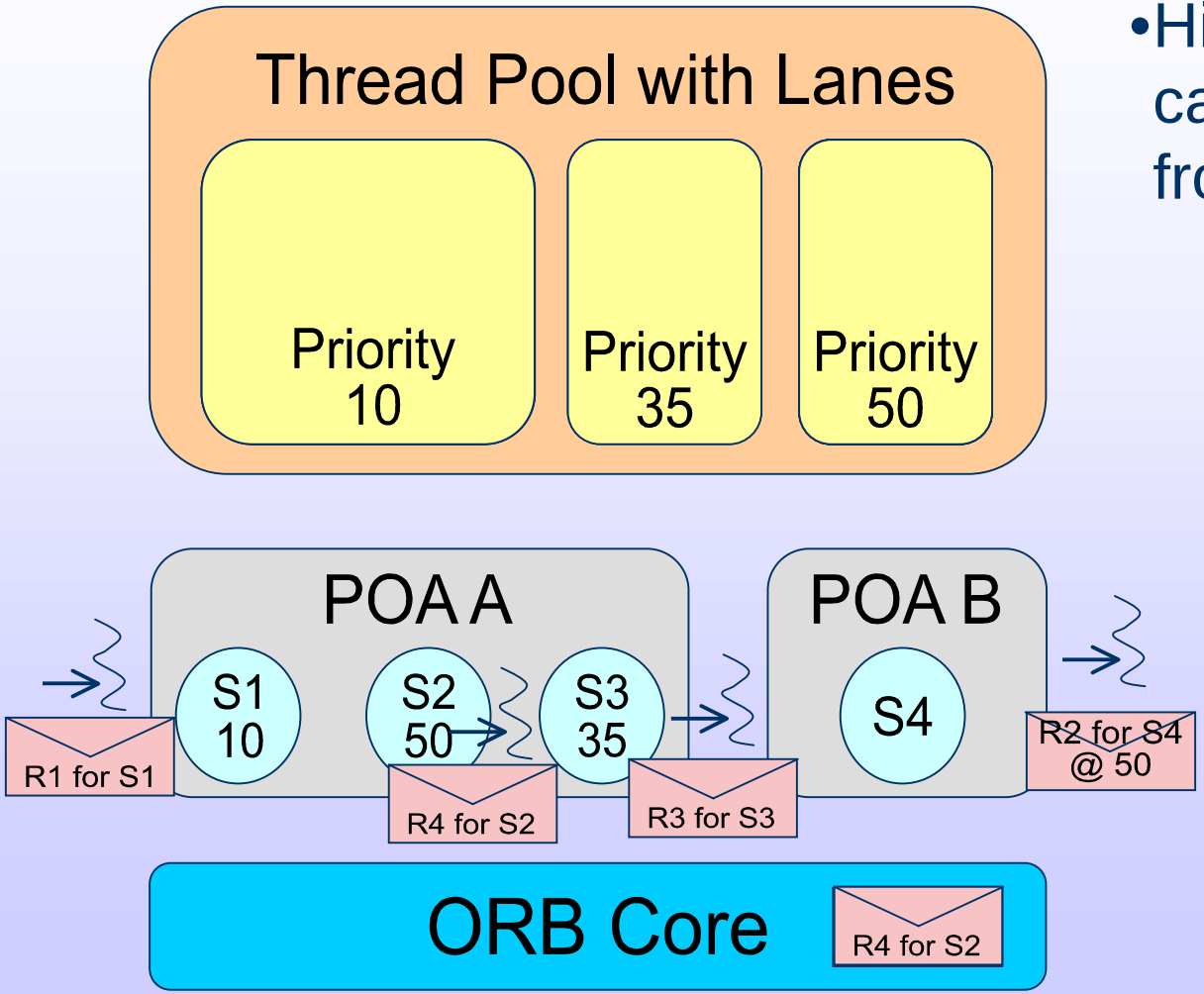
When a thread pool is created it's possible to control certain resource allocations

- *e.g.*, stacksize, request buffering, & whether or not to allow "borrowing" across lanes

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

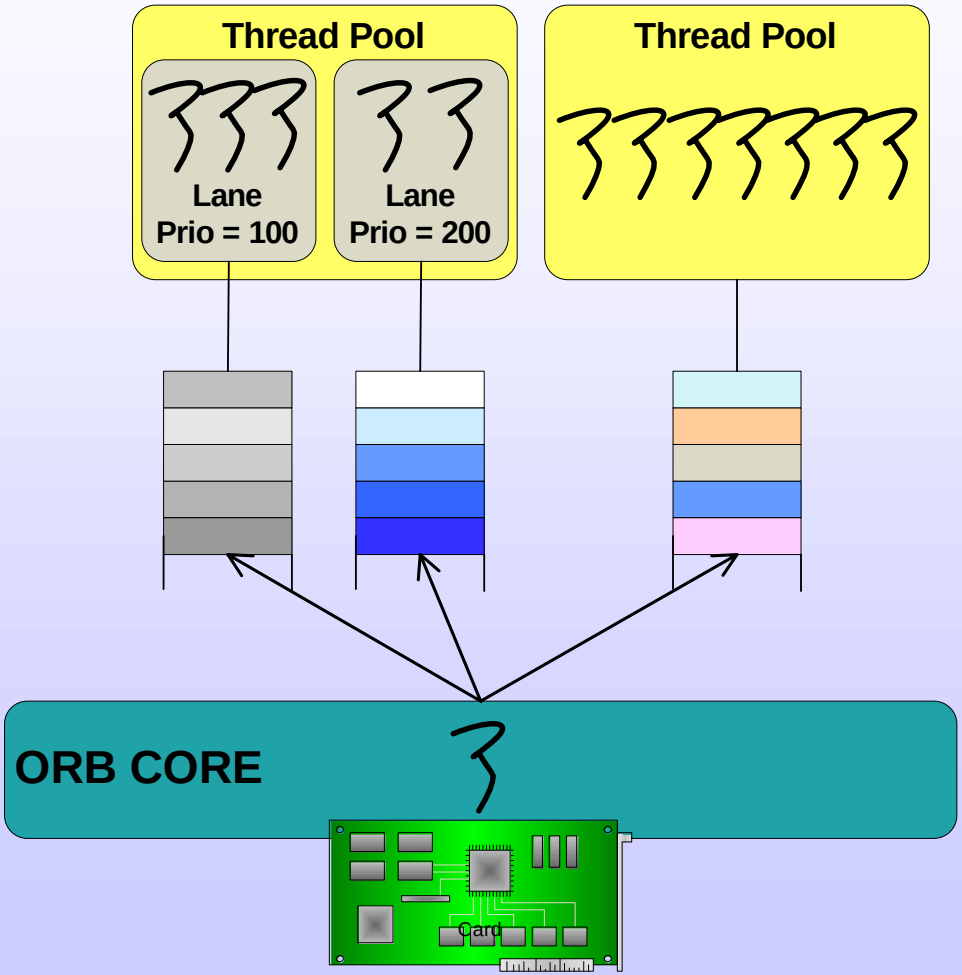
Restoring threads

- 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

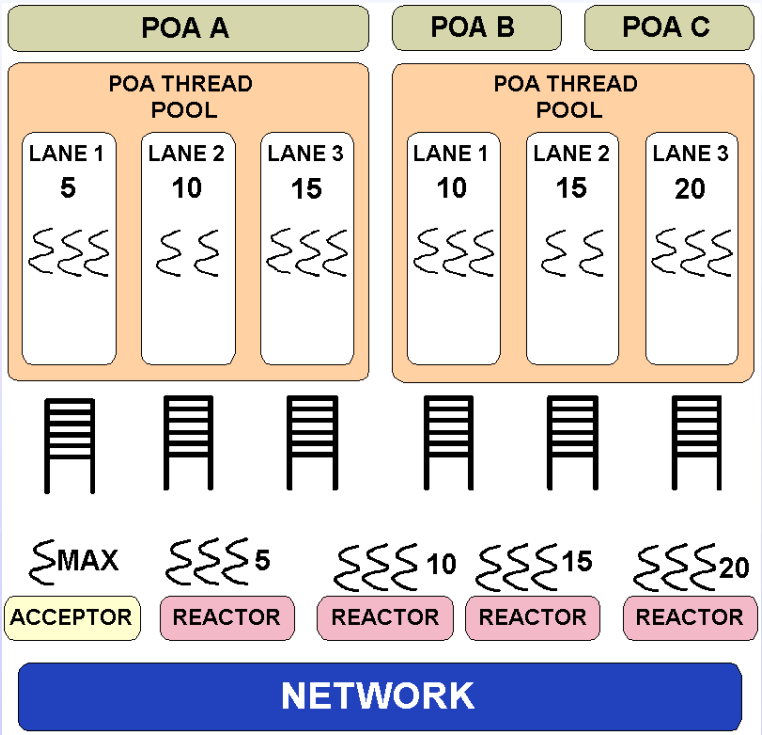
Configuring Request Buffering

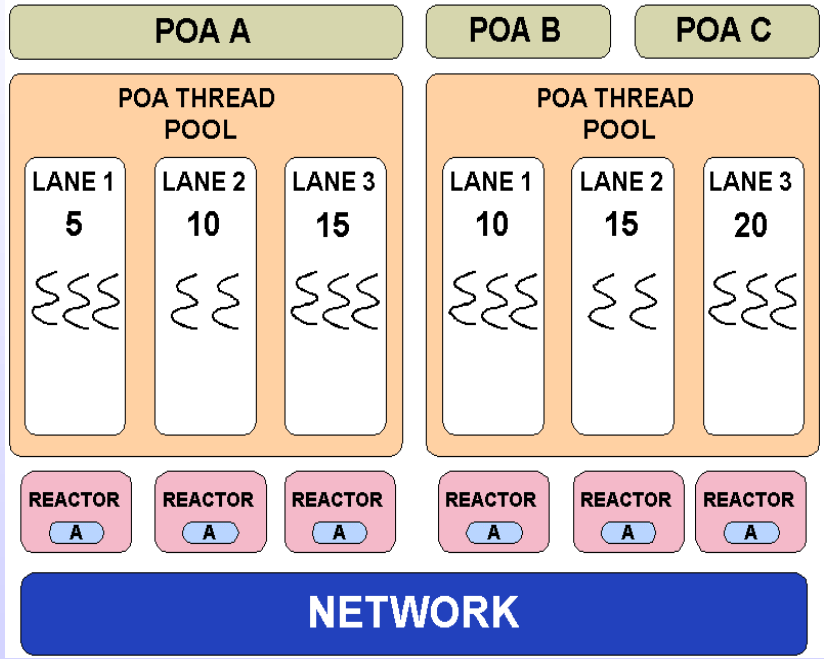
```
// Create a thread pool with buffering
RTCORBA::ThreadpoolId pool_id =
   rt_orb->create_threadpool (1024 * 10,  // Stacksize
                              4,          // Static threads
                              10,         // Dynamic threads
                              DEFAULT_PRIORITY,
 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

// 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 I/O subsystem of underlying OS

Thread Pools Implementation Strategies

- There are two general strategies to implement RT CORBA thread pools:





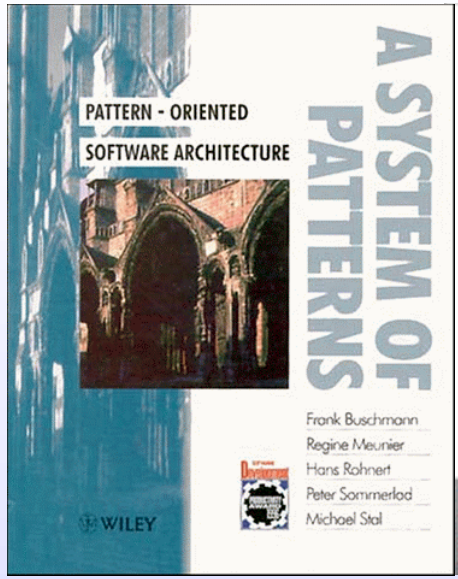
- Use the *Half-Sync/Half-Async* pattern to have I/O thread(s) buffer client requests in a queue & then have worker threads in the pool process the requests

- Use the *Leader/Followers* pattern to demultiplex I/O events into threads in the pool *without* requiring additional I/O threads

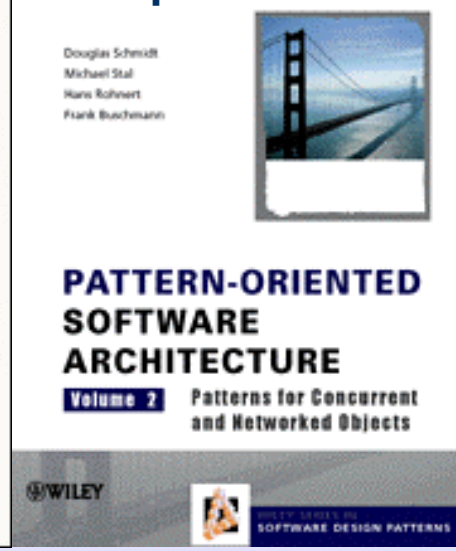
Overview of Patterns & Pattern Languages

Patterns

- Present *solutions* to common software *problems* arising within a certain *context*
- Help resolve key design forces
- Capture recurring structures & dynamics among software participants to facilitate reuse of successful designs
- Generally codify expert knowledge of design constraints & "best practices"

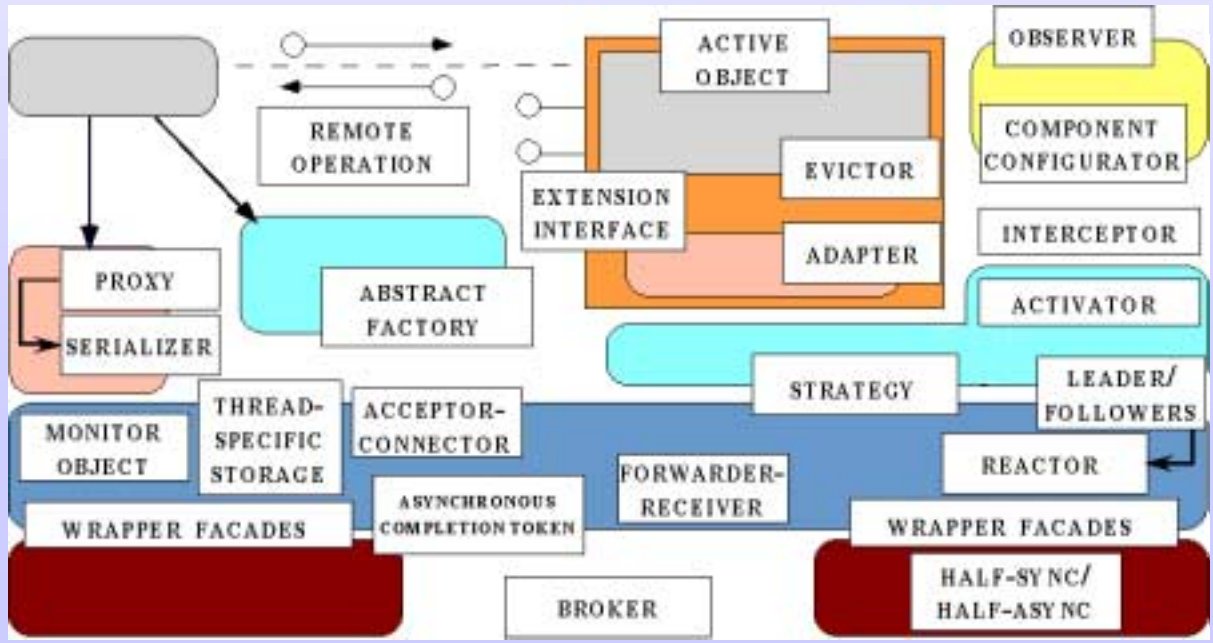


www.posa.uci.edu



Pattern Languages

- Define a *vocabulary* for talking about software development problems
- Provide a *process* for the orderly resolution of these problems
- Help to generate & reuse software *architectures*



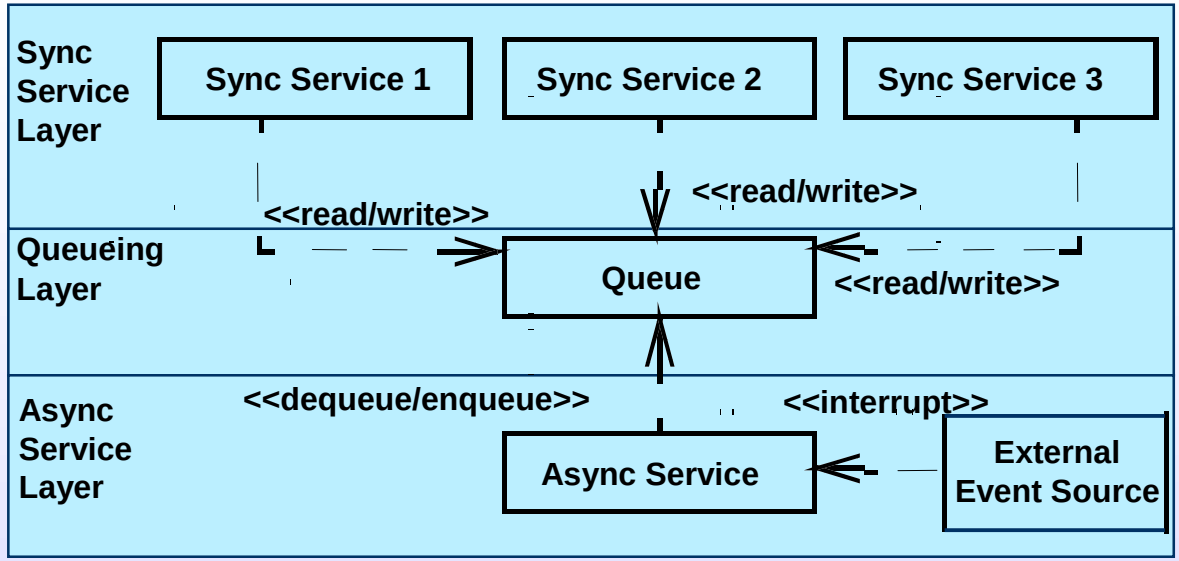
Evaluating Thread Pools Implementations

- 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

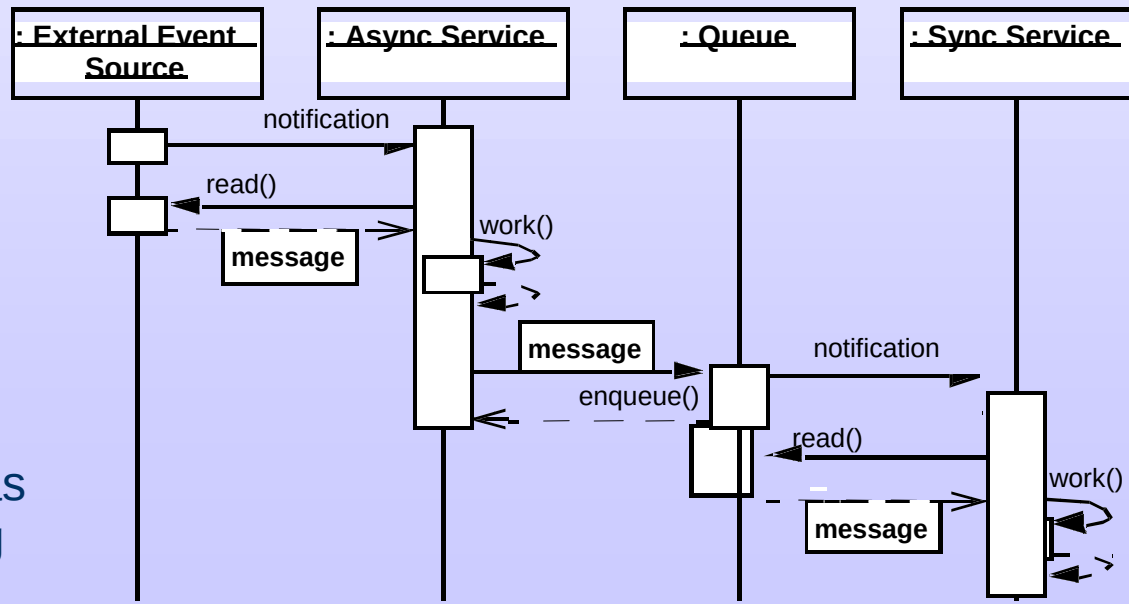
Capability	Description
Feature support	Request buffering & thread borrowing
Scalability	Endpoints & event demultiplexers required
Efficiency	Data movement, context switches, memory allocations, & synchronizations required
Optimizations	Stack & thread specific storage memory allocations
Priority inversion	Bounded & unbounded priority inversion incurred in each implementation

The Half-Sync/Half-Async Pattern

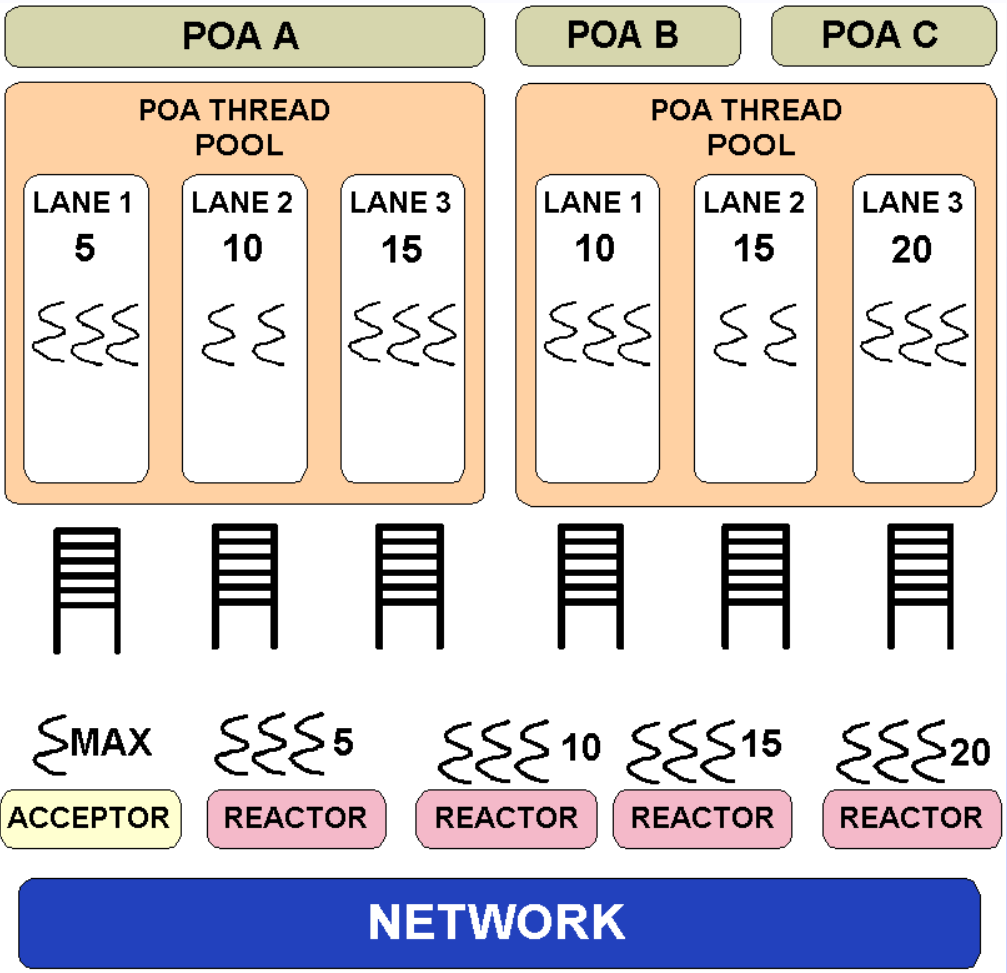
Intent
The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



- 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

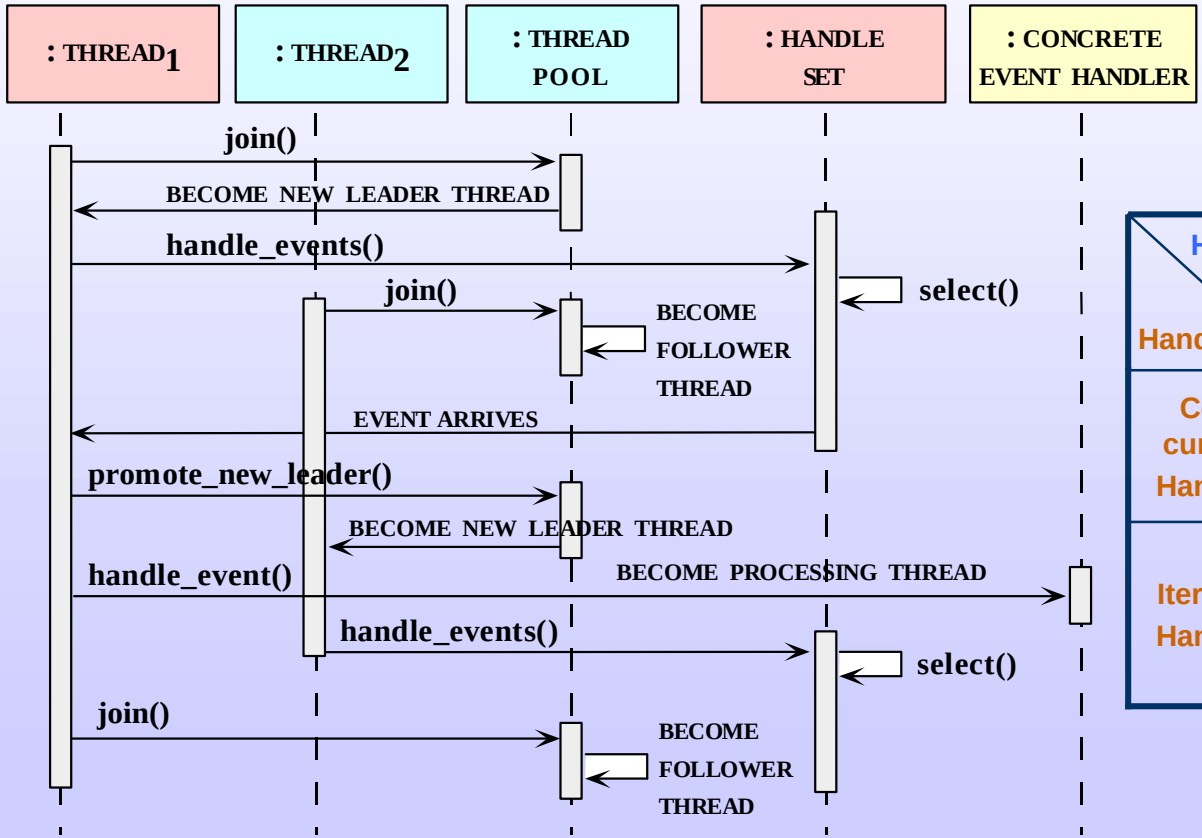
- Less efficient because of queueing
- Predictability reduced without _bind_priority_band() implicit operation

Evaluation of Half-Sync/Half-Async Thread Pools

Criteria	Evaluation
Feature Support	***Good:*** supports request buffering & thread borrowing
Scalibility	***Good:*** I/O layer resources shared
Efficiency	***Poor:*** high overhead for data movement, context switches, memory allocations, & synchronizations
Optimizations	***Poor:*** stack & TSS memory not supported
Priority Inversion	***Poor:*** some unbounded, many bounded

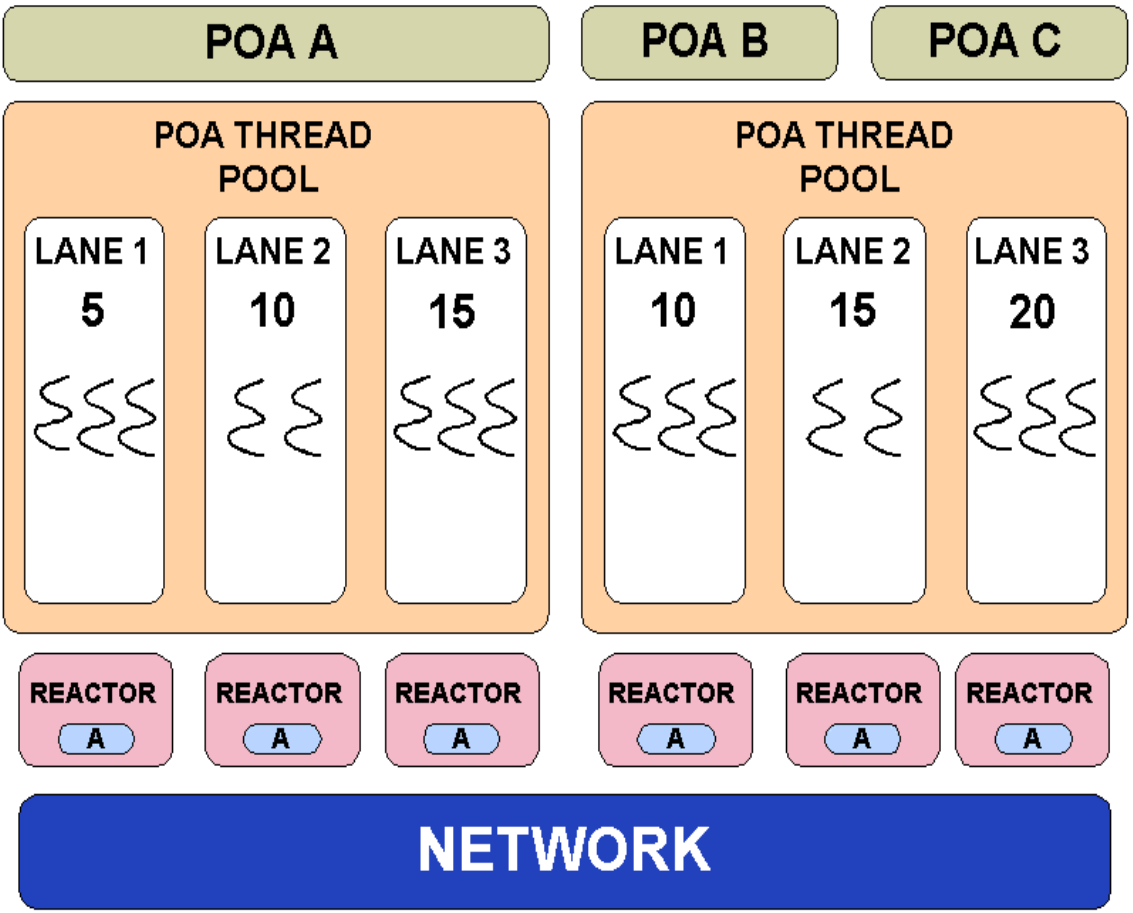
The Leader/Followers Pattern

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



Handle Sets / Handles	Concurrent Handle Sets	Iterative Handle Sets
Con-current Handles	UDP Sockets + `WaitForMultipleObjects()`	UDP Sockets + `select()/poll()`
Iterative Handles	TCP Sockets + `WaitForMultipleObjects()`	TCP 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

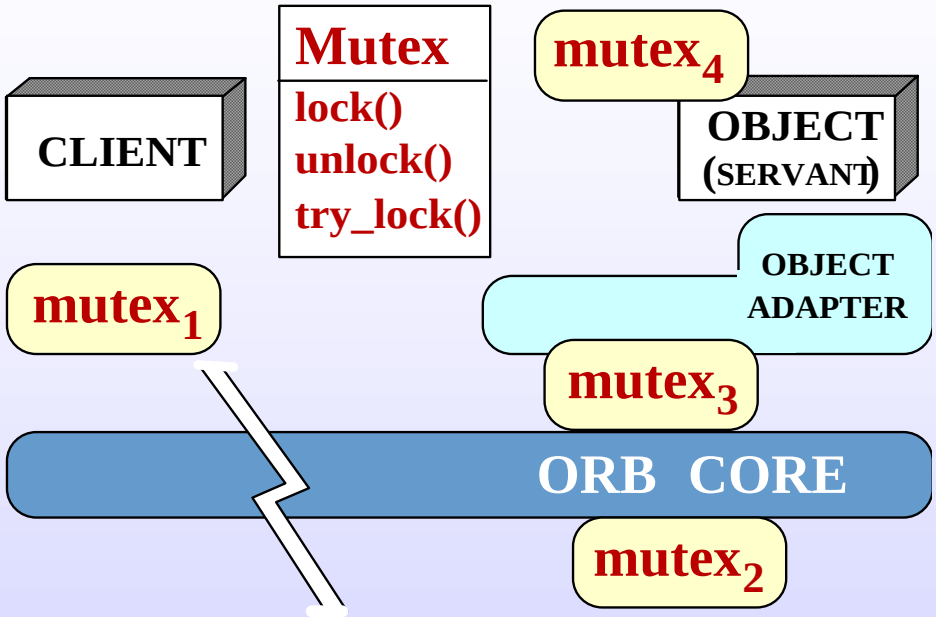
Evaluation of Leader/Followers Thread-Pools

Criteria	Evaluation
Feature Support	***Poor:*** not easy to support request buffering or thread borrowing
Scalibility	***Poor:*** I/O layer resources not shared
Efficiency	***Good:*** little or no overhead for data movement, memory allocations, or synchronizations
Optimizations	***Good:*** stack & TSS memory supported
Priority Inversion	***Good:*** little or no priority inversion

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

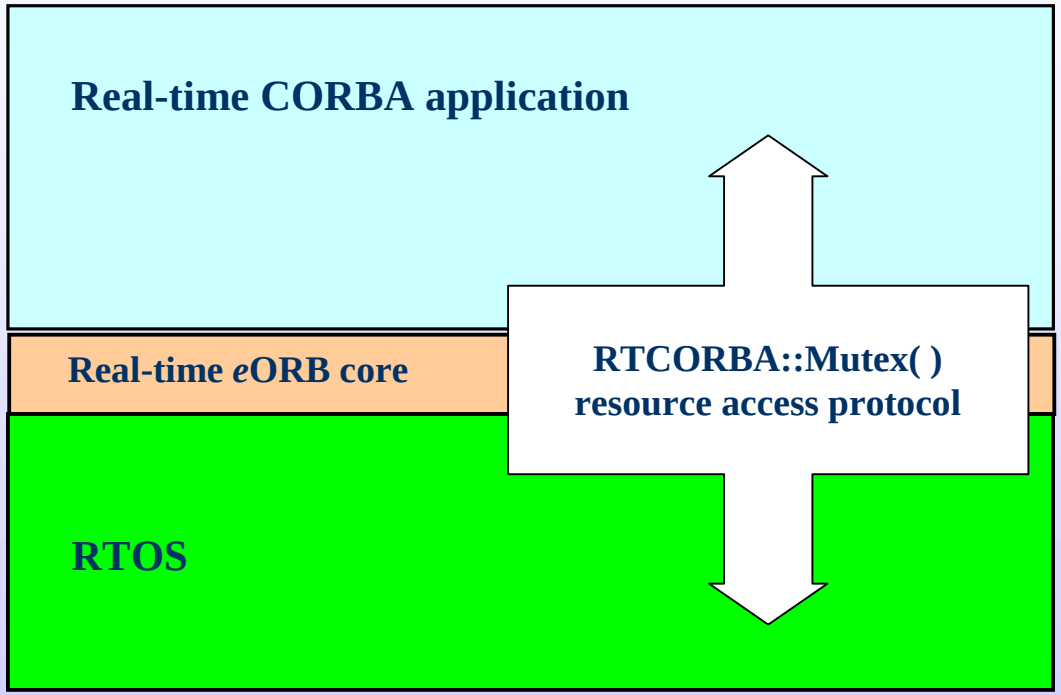


- The **RTCORBA::Mutex** interface ensure consistent mutex semantics, across ORB & application domains –
- What does this mean ?
- read on….

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

create_mutex() is a factory method

Synchronizing Objects Consistently



Synchronizing Objects Consistently

The **`RTCORBA::Mutex`** interface ensure consistent mutex semantics, across ORB & application domains – in effect a ***'priority cognizant mutex'.***

The RTCORBA Mutex is an open extensible synchronizer upon which more complicated synchronization mechanisms may be built.

There are many ways to implement the RTCORBA Mutex such that RTOS, ORB and application share the same synchronization semantics – e.g. Open pluggable mutex.

Some solutions have had RTOS consult middleware for synchronization e.g. Jensen et al. DEC Alpha and vxworks

The Mutex has associated with it a '*resource access protocol'* oft termed a *'priority protocol'* which real-time systems and OSs use to resolve resource contention based on a sliding scale of priority and order of other application criteria that identify their importance.

Literature referenced in earlier slides e.g. Liu and Buttazzo detail mathematically the stability of such algorithms providing mathematical schedulable task-set 'feasibility' proofs. (Liu – Chapter 8, Buttazzo Chapter 7)

The priority protocol used by the ORB must be the same as the one used in the RTOS to avoid anomalous behaviors, and bound, minimize, or eliminate the possibility of 'priority inversion'.

Synchronizing Objects Consistently

RTCORBA ORBs may supply implementations 'pluggably' in the **`RTCORBA::Mutex`** interface for any priority protocol that is appropriate for particular environments and RTOS's upon which the distributed real-time application is to be hosted.

Common resource access protocols include (both for uniprocessor and multi-processor systems)

- Basic Priority Inheritance Protocol
- Basic Priority Ceiling Protocol
- Stack Based Priority Ceiling Protocol
- Preemption Ceiling Protocol
- Nonpreemptive Critical Section Protocol
- Hybrid variants from various vendors of schedulability solutions e.g. TriPacific with DASPCP.

Criteria of selection are usually based on important characteristics such as

- WCRT (worst case response times)
- Worst chained blocking time,
- Transparency to application programmer,
- Bounding/reducing stack consumption, (and therefore memory)
- Probability of deadlock likelihood,
- Context switching overhead

Synchronizing Objects Consistently

In multi-processor systems we still have an important notion of end-to-end feasibility of schedulability and resource access.

Heterogenous end-to-end systems typically are what modern real-time CORBA systems look like. – Real-time CORBA 1.2 with dynamic distributed threads aims to solve this problem. It uses eligibility and feasibility criteria other than priority such as deadlines and EDF algorithm as an example.

There are models to analyze synchronization in such systems using Multi-processor priority-ceiling protocol (see Liu chapter 9).

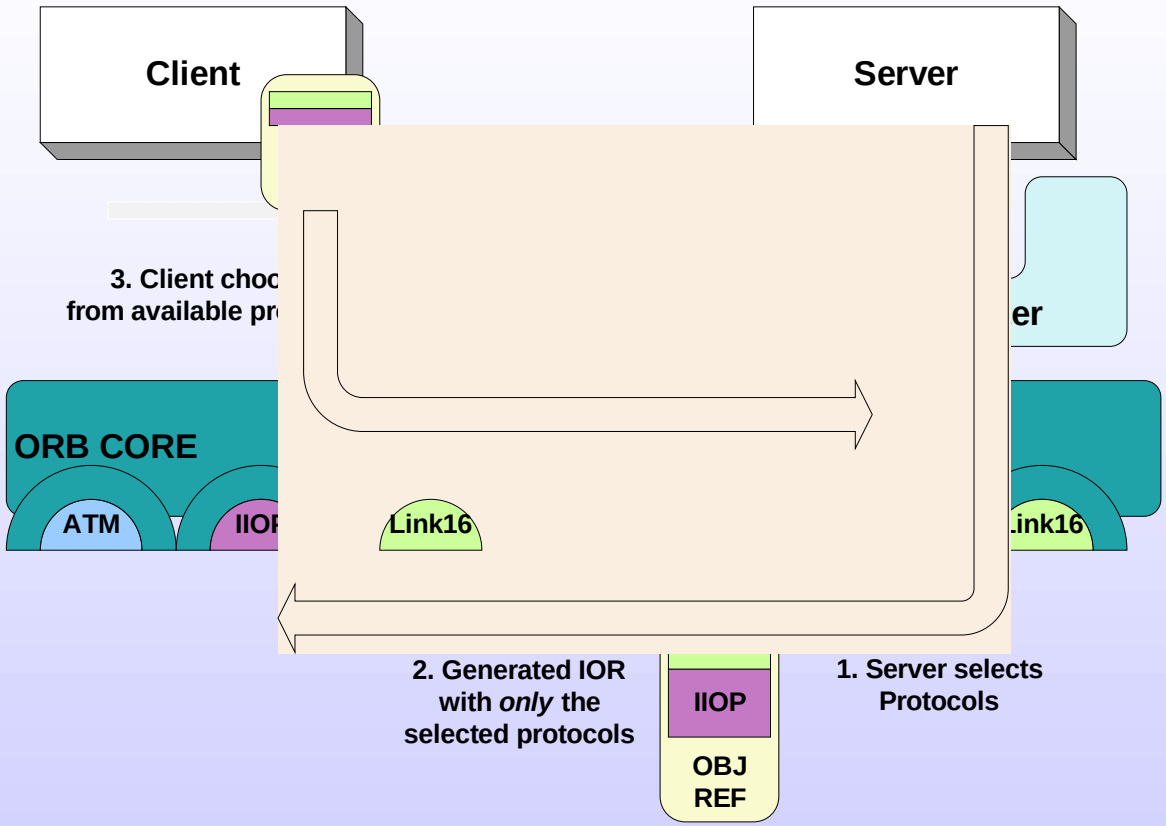
The notion of synchronicity vs. asynchronicity is a tradeoff that the RTCORBA engineer should be aware of, especially tradeoff of efficiency of async vs. simplicity of sync.

An important consideration in your analysis should be *overload analysis* and how to resolve it in a real-time CORBA system with all this synchronization that is provided.

Custom Protocol Configuration

- **Problem:** 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:** Use RT-CORBA Protocol Policies to select and/or configure communication protocols

Configuring Custom Protocols



- Both server-side & client-side policies are supported
- Some policies control protocol selection, others configuration
- Order of protocols indicates protocol preference

Ironically, RT-CORBA specifies only protocol properties for TCP!

Example: Configuring protocols

- First, we create the protocol properties

```
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

```
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);
```

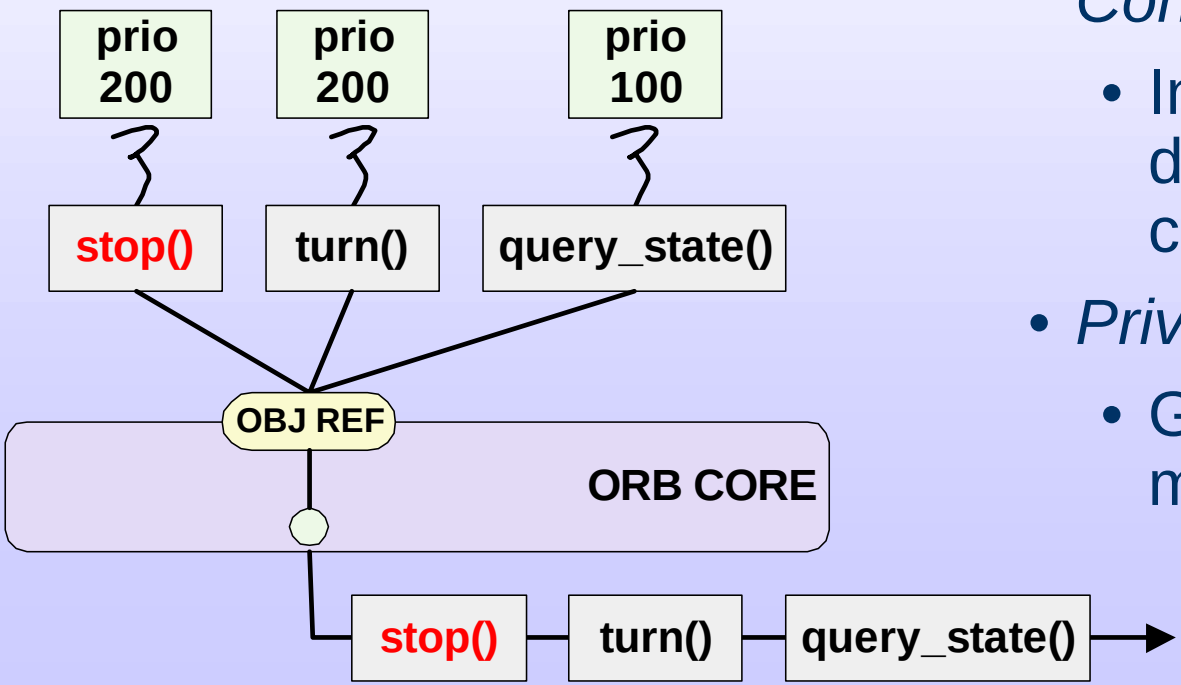
Network Resource Issues

- **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

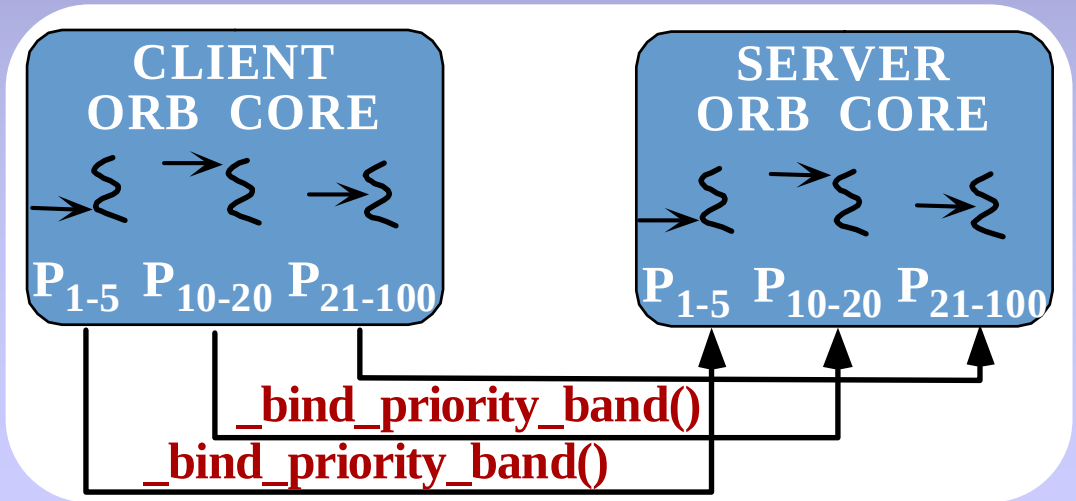
Note the priority inversion below since the **`stop()`**, **`turn()`**, and **`query_state()`** requests all share the same connection



- *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

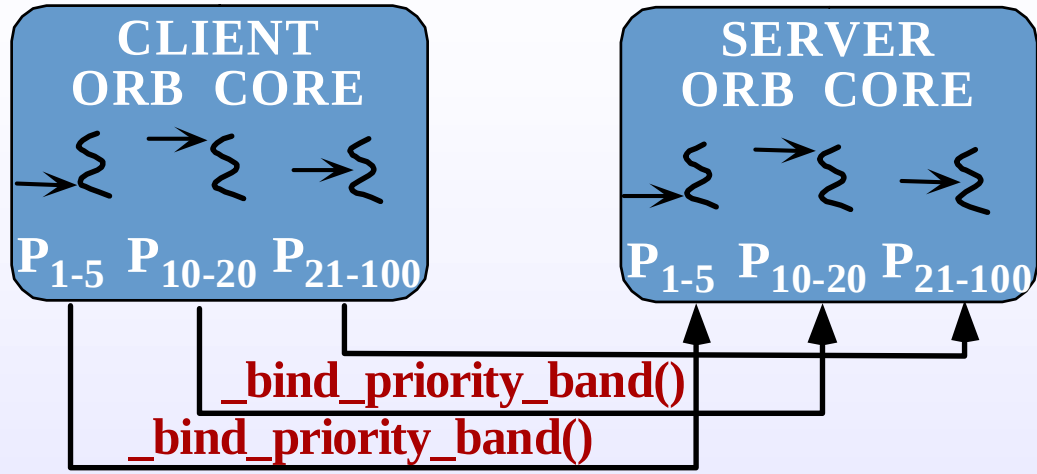
Connection Establishment

- **Problem:** How can we prevent connection establishment between the base station and the drones from resulting in unacceptable jitter?
  - Jitter is detrimental to time-critical applications
- **Solution:** Pre-allocate one or more connections using the `Object::_validate_connection()` operation



Pre-allocating Network Connections

The **_validate_connection()** operation must be invoked before making any other operation calls



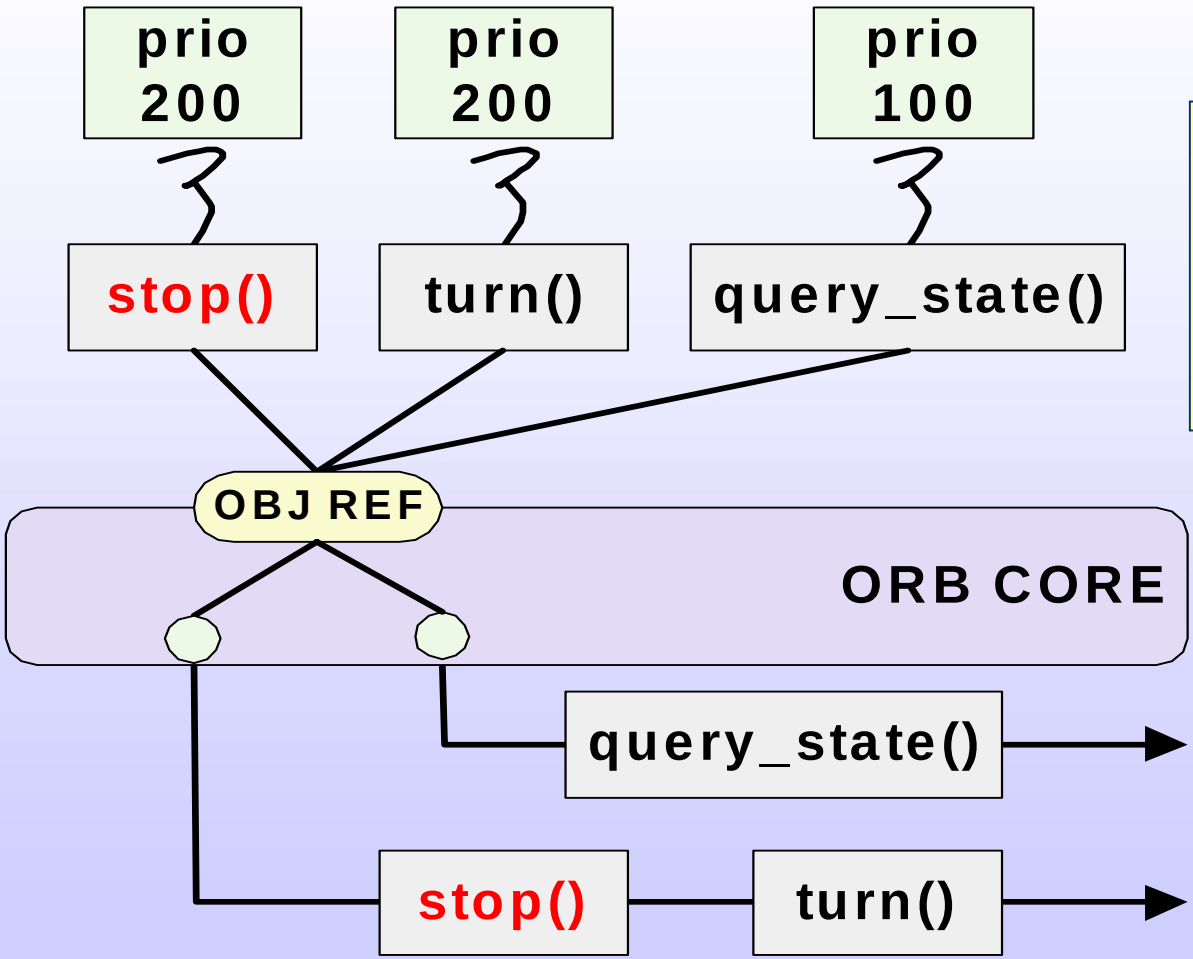

```
// Drone reference
Drone_var drone = ...;

// Pre-establish connections
// using current policies
CORBA::PolicyList_var invalid_policies;

// The following operation causes a _bind_priority_band()
// “implicit” request to be sent to the server
CORBA::Boolean success =
  drone->_validate_connection (invalid_policies);
```

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

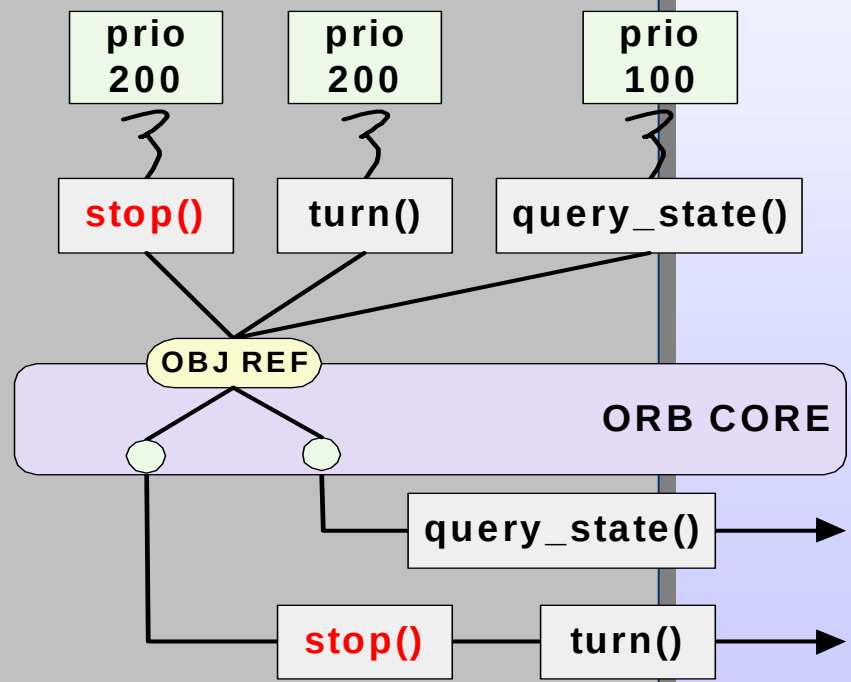


Note how the **`stop()`** and **`turn()`** requests no longer share the same connection as **`query_state()`** requests

Priority Banded Connection Policy

```
RTCORBA::RTORB_var rtorb =
  RTCORBA::RTORB::_narrow (
    orb->resolve_initial_references
      (“RTORB”));
CORBA::PolicyManager_var orb_pol_mgr =
  CORBA::PolicyManager::_narrow (
    orb->resolve_initial_references
      (“ORBPolicyManager”);
// 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::PolicyList policy_list;
policy_list.length (1);
policy_list[0] = rtorb->
  create_priority_banded_connection_policy (bands);
orb_pol_mgr->_set_policy_overrides
  (policy_list, CORBA::ADD_OVERRIDE);
```

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



Overriding IOR with PriorityBands

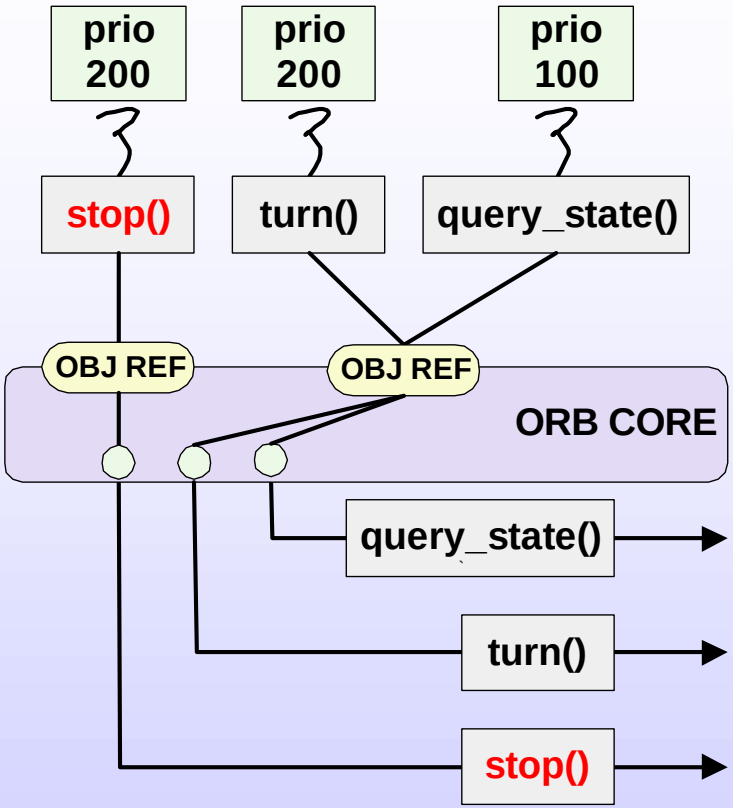
```
Controller_var controller = // get from naming service, etc.

// Override the object reference with banding policies.
CORBA::Object_var temp =
  controller->_set_policy_overrides (policy_list,
                                     CORBA::ADD_OVERRIDE);

Controller_var rt_controller = Controller::_narrow (temp);

// Real-time invocation using priority banding
rt_controller->edge_alarm ();

// Normal invocation without priority banding.
controller->edge_alarm ();
```

- The servant invoked via the **`rt_controller`** object reference runs at the priority of the client thread's priority
- The servant invoked via the **`controller`** object reference runs at an undefined priority in the server
  - This behavior is clearly undesirable in a real-time application

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

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

Scheduling Activities

- **Problem:** How can RT-CORBA give developers control over system resources while avoiding the following two deficiencies:
  - It can be tedious to configure all the CORBA client/server policies
  - Application developers 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

Client-side Scheduling

- **The client-side programming model is simple**

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

// Schedule the ‘edge_alarm’ activity
scheduler->schedule_activity (“edge_alarm”);

controller->edge_alarm ();
```

- Note the Scheduling Service is an optional part of RT-CORBA 1.0

Server-side Scheduling

- Servers can also be configured using the Scheduling Service

```
// 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*****:** The Joint Submission (orbos/01-06-09) has been accepted

Controlling Request Timeouts

- **Problem:** How can we keep our `Controller` objects from blocking indefinitely when trying to stop a drone that's about to fall off an edge?!
- **Solution:** Override the timeout policy in the `Drone` object reference

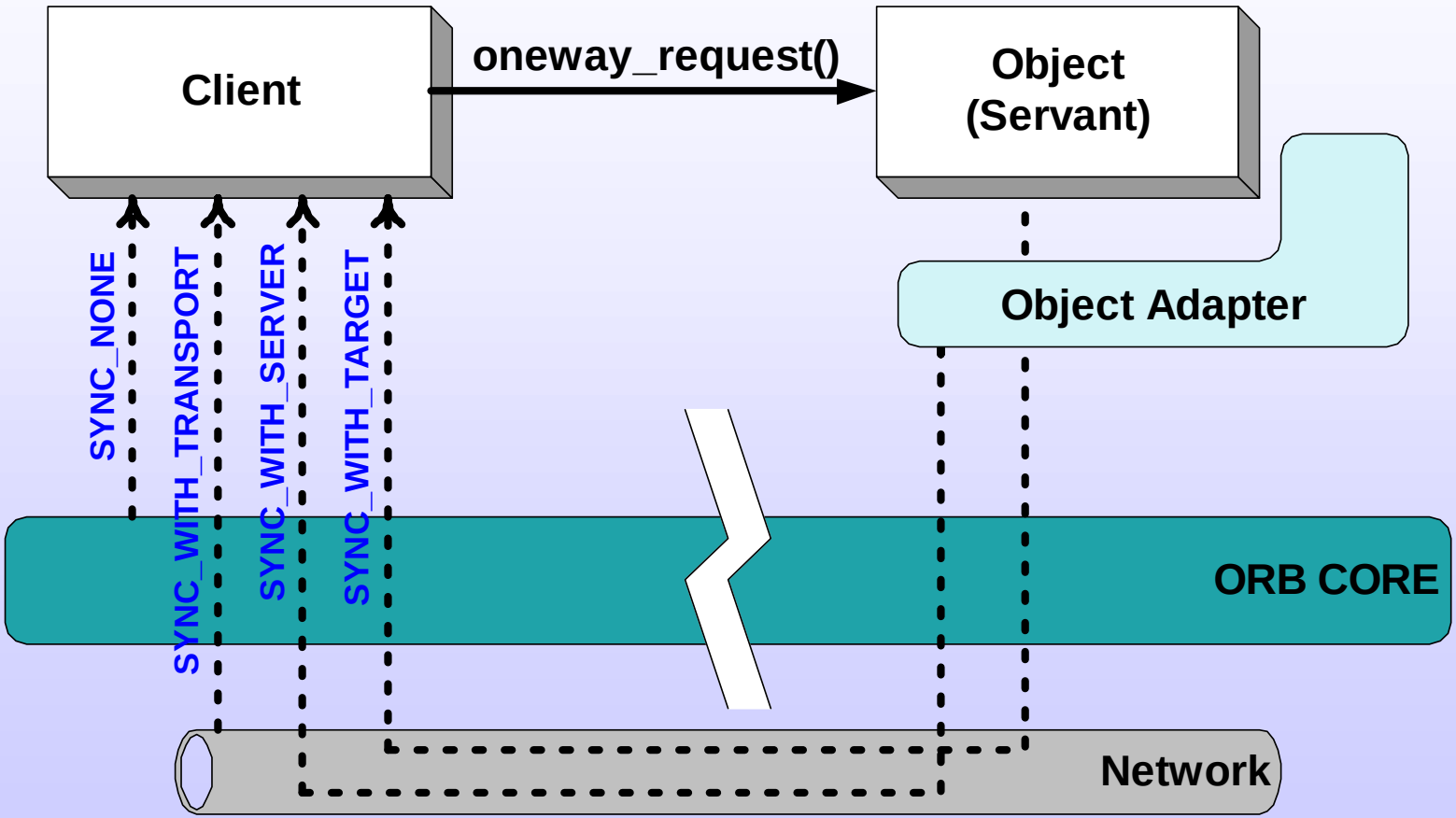
Applying Request Timeouts

```
// 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. */ }
```

Oneway Calls

- **Problem:** How can we handle the fact that CORBA one-way operation semantics aren't precise enough for real-time applications?
- **Solution:** Use the **SyncScope** policy to control one-way semantics

Reliable Oneways

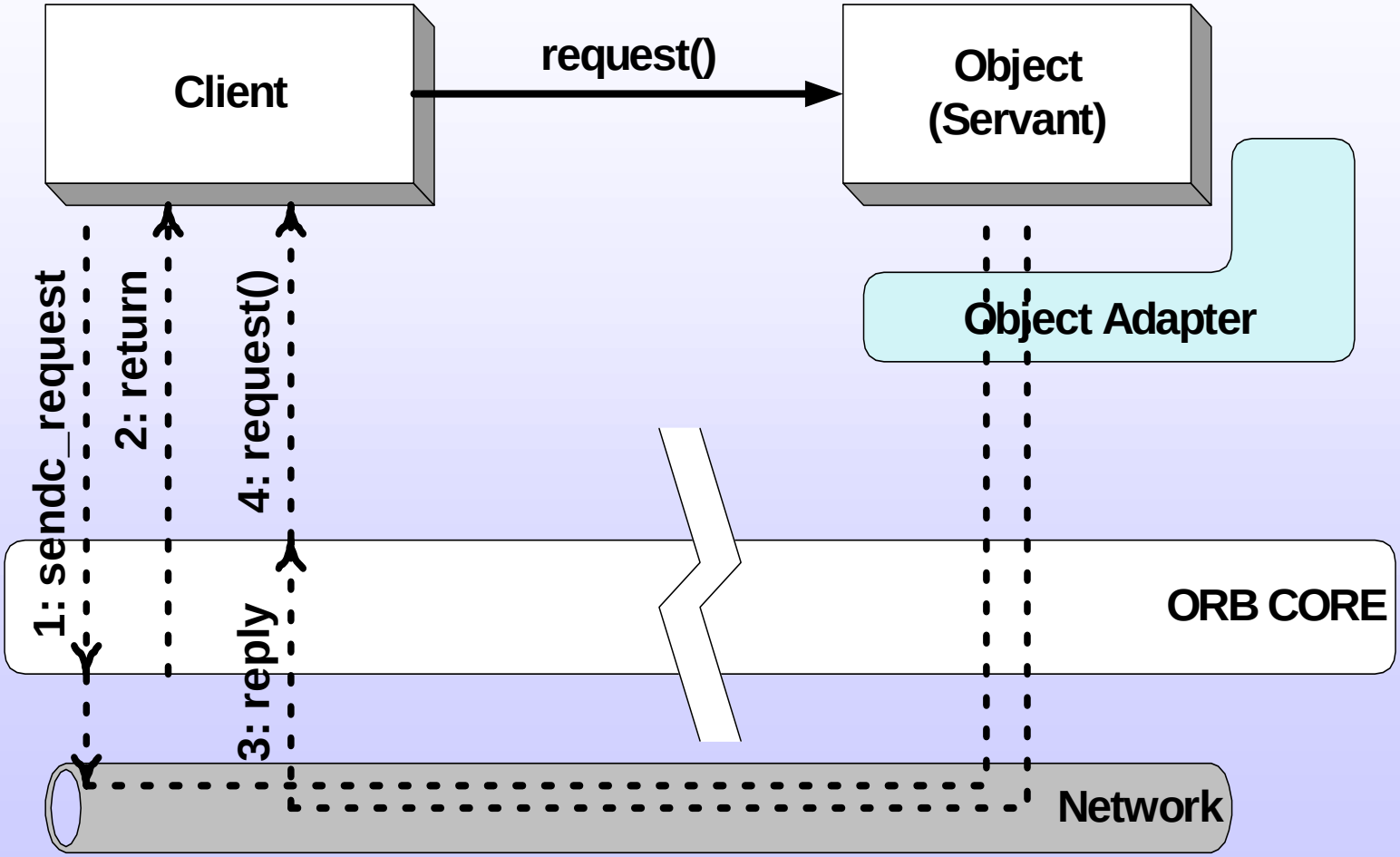


Asynchronous Method Invocation

- **Problem:** How can we simultaneously
  1. Prevent clients from blocking while long-duration requests complete &
  2. Allow many requests to be issued concurrently

- **Solution:** Use the CORBA Asynchronous Method Invocation (AMI) interfaces to separate (in time & space) the thread issuing the request from the thread processing the reply

Asynchronous Method Invocation (AMI)



Open Issues with the Real-Time CORBA Specification and some advisory notes

1. No standard APIs for setting & getting priority mappings & priority transforms

2. Few 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 overly dependent on implementation details

5. Use of RT-ORBs today

- **Today RT and embedded ORB user community are the 'power users'**
  - **measures their ORB extensively in many ways – what should they look for to get the best**
  - **How to use orb in 'fast' mode**
    - **Use simple primitive IDL types (union/any)**
    - **Define clean simple IDL (short method names)**
    - **Stay away from recursive IDL**
    - **Be aware of the structure of a GIOP request**
    - **Exercise linear section of the ORBs performance envelope.**
    - **When you think the edges of the linear envelope are not giving you the performance –' then' and only then think about a new fabric/transport.**
  - **How to use orb in 'deterministic' mode**
    - **Set thread priorities according to well defined and simulated a-priori schedule- there is no substitute for a scheduling analysis.**
    - **Write simple servant code – avoid recursion…..**
    - **Use simple well understood POA semantics**
    - **Make sure you understand your priority mapping and transformation for a particular RTOS . . . e.g. posix calls**
    - **Be aware of the structure of a GIOP request**
    - **Use Messaging Roundtrip request timeout facility and**
    - **Messaging sync scoping to clearly identify the semantic importance of your calls**

Use of RTORBs today

- **CORBA IDL interface**

```
interface I
{
   void f( );
};
```

RTORB for data transport

- **CORBA Request - wire**

```
0x47, 0x49, 0x4F, 0x50,                 // 'G' 'I' 'O' 'P'
0x01, 0x00,                             // GIOP version '1' '0'
0x01,                                   // flgs
0x00,                                   // msg type  (one of 8 )
0x28, 0x00, 0x00, 0x00,                 // msg size

/* RequestHeader */
0x00, 0x00, 0x00, 0x00,                 // svc context
0x00, 0x00, 0x00, 0x00,                 // request id
0x01,                                   // response_expected
0x00, 0x00, 0x00,                       // padding
0x09, 0x00, 0x00, 0x00,                 // object key size
0x00, 0x00, 0x00, 0x00, 0x00,           // object key body
        0x00, 0x00, 0x00, 0x00,
0x04, 0x31, 0x00,                       // padding
0x02, 0x00, 0x00, 0x00,                 // operation name size
0x66, 0x00,                             // operation name ('f' '0')
0x00, 0x00,                             // padding
0x00, 0x00, 0x00, 0x00                  // principal
```

Additional Information

- **CORBA 3.0 specification (includes RT CORBA)**
  - **www.omg.org/technology/documents/formal/corbaiiop.htm**
- **Patterns for concurrent & networked OO middleware**
  - **www.posa.uci.edu**
- **Real-time & Embedded CORBA ORBs**
  - **TAO <www.theaceorb.com>**
  - **e*ORB <www.prismtechnologies.com>**
  - **HighComm <www.highcomm.com>**
  - **ORBacus/E <www.ooc.com>**
- **CORBA research papers & tutorials etc**
  - **www.real-time.org/ ***
  - **www.cs.wustl.edu/~schmidt/ ***

Additional Information

Recommended texts on real-time systems scheduling theory

1. ***Real-time Systems,***
   Jane Liu, Prentice Hall.
2. ***Hard Real-time Computing Systems,***
   G.C. Buttazzo, Kluwer
3. ***Scheduling in real-time systems,***
   F. Cottet, Delacroix, et al., Wiley
4. ***Practitioners Handbook for Real-time Analysis,***
   Klein, Ralya, et al., Kluwer
5. ***Real-time Systems,***
   Nimal Nissanke, Prentice Hall.
6. ***Synchronization in Real-time Systems,***
   R. Rajkumar, Kluwer.
7. ***Deterministic and Stochastic Scheduling,***
   Dempster, M.A.H., J.K. Lenstra, and A.H.G. Rinnooy Kan (Eds.), D. Reidel.
8. ***Scheduling: Theory, Algorithms, and Systems,***
   Pinedo, M., 2nd edition, Prentice Hall.