# Evaluating Thread Pool Strategies for Real-time CORBA

Irfan Pyarali, Marina Spivak, and Ron Cytron

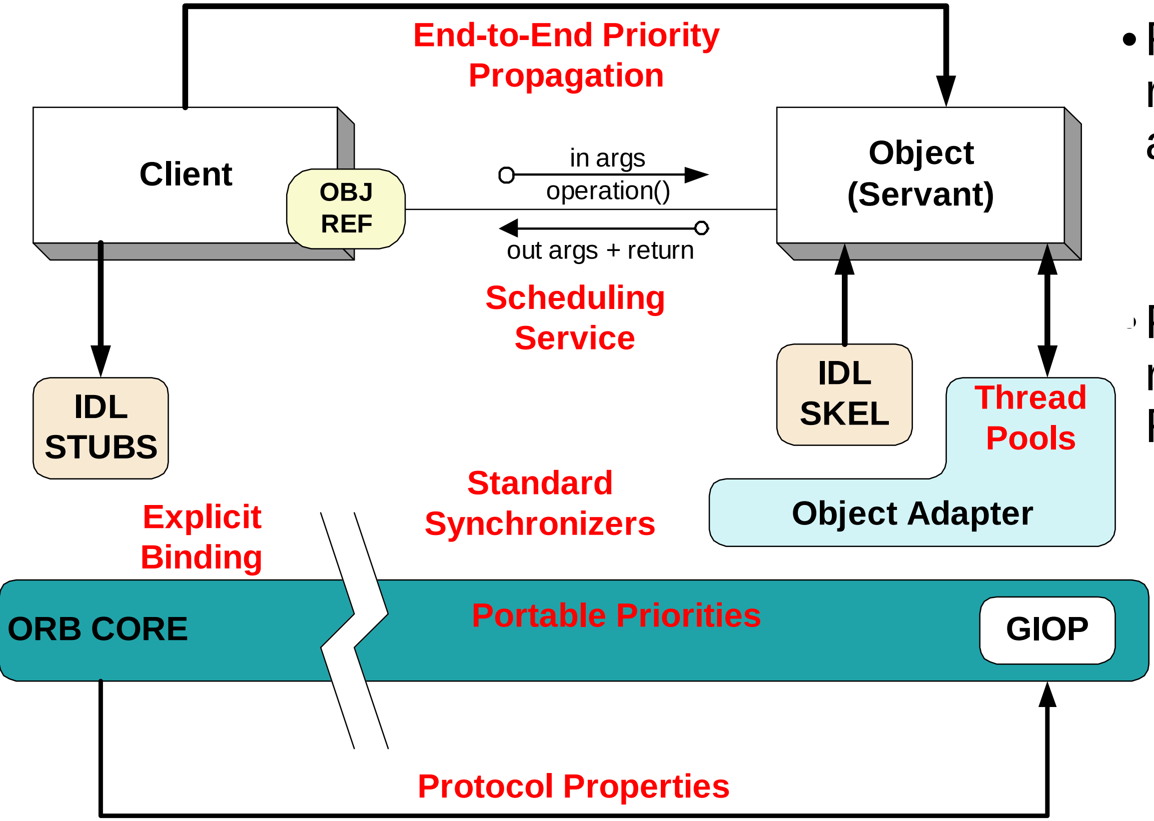
**{irfan,marina,cytron}@cs.wustl.edu**

Computer Science Dept.
Washington University,
St. Louis

**http://www.cs.wustl.edu/~doc/**

Wednesday, July 18, 2001

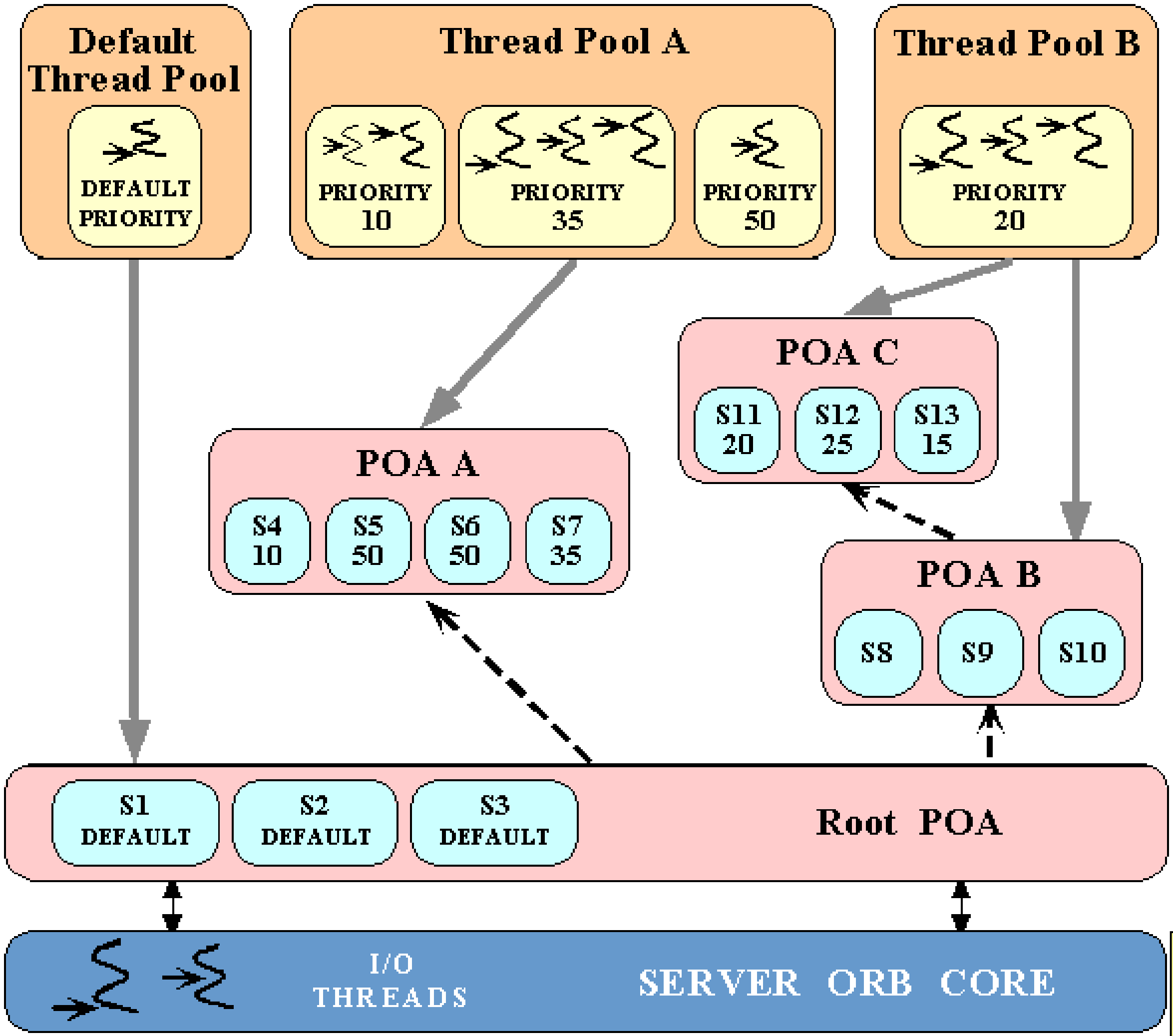
# Real-Time CORBA Overview




Real-time CORBA leverages the CORBA Messaging QoS Policy framework

- RT CORBA adds QoS control to regular CORBA 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

# Thread Pools




**Leverage hardware**
- Multi-processors machines

**Increase performance**
- Overlap computation and I/O

**Improve response-time**
- Support long durations upcalls

**Different levels of service**
- High vs low-priority tasks

**Support preemption**
- Prevent unbounded priority inversion

**Scheduling**
- Strict control over processor resources essential for many RT applications

Note that a thread pool can manage multiple POAs

# 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);
};
```

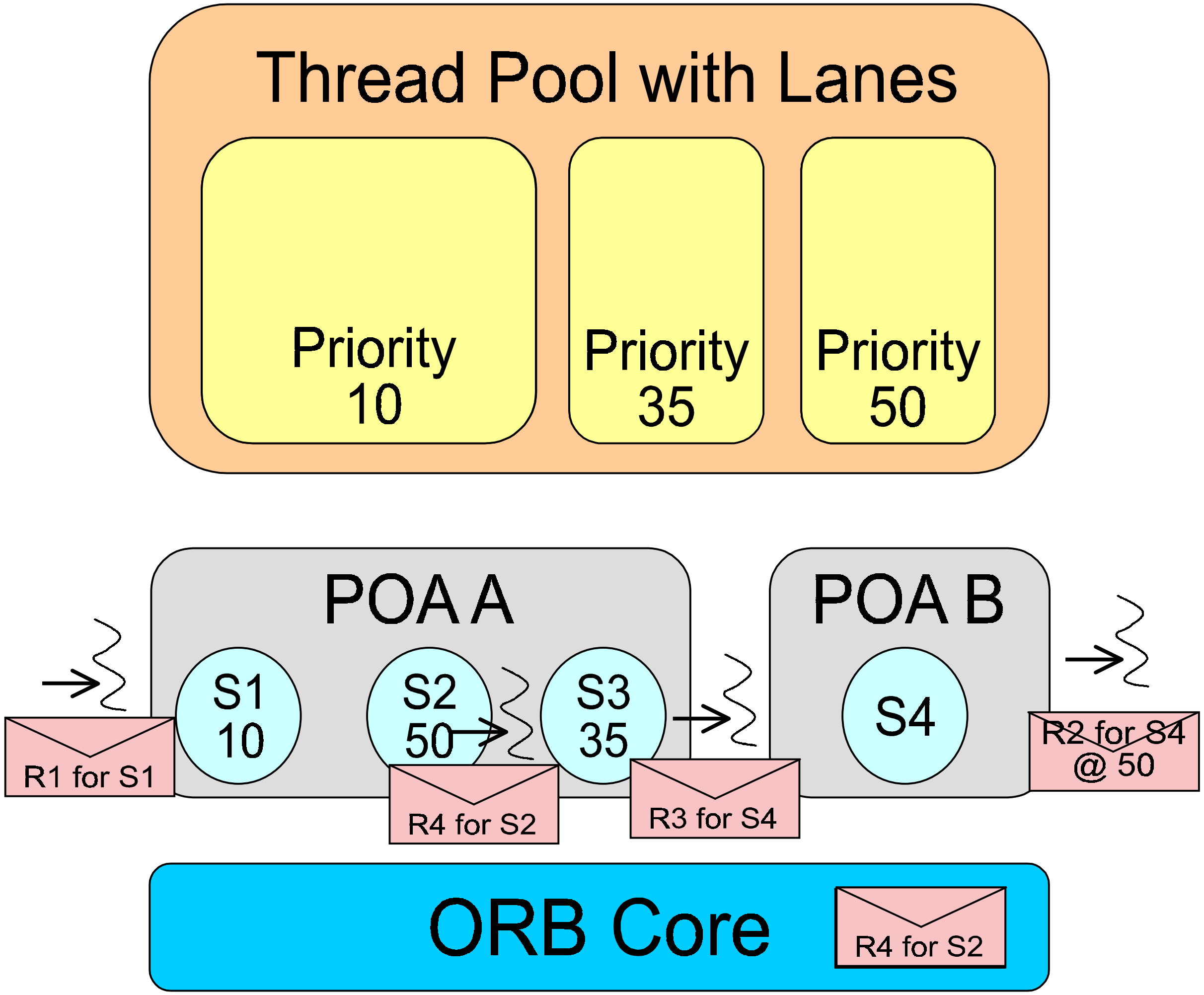
These are factory methods for controlling the life-cycle of RT-CORBA thread pools

# Creating Thread Pools with Lanes

```
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

# Thread Borrowing



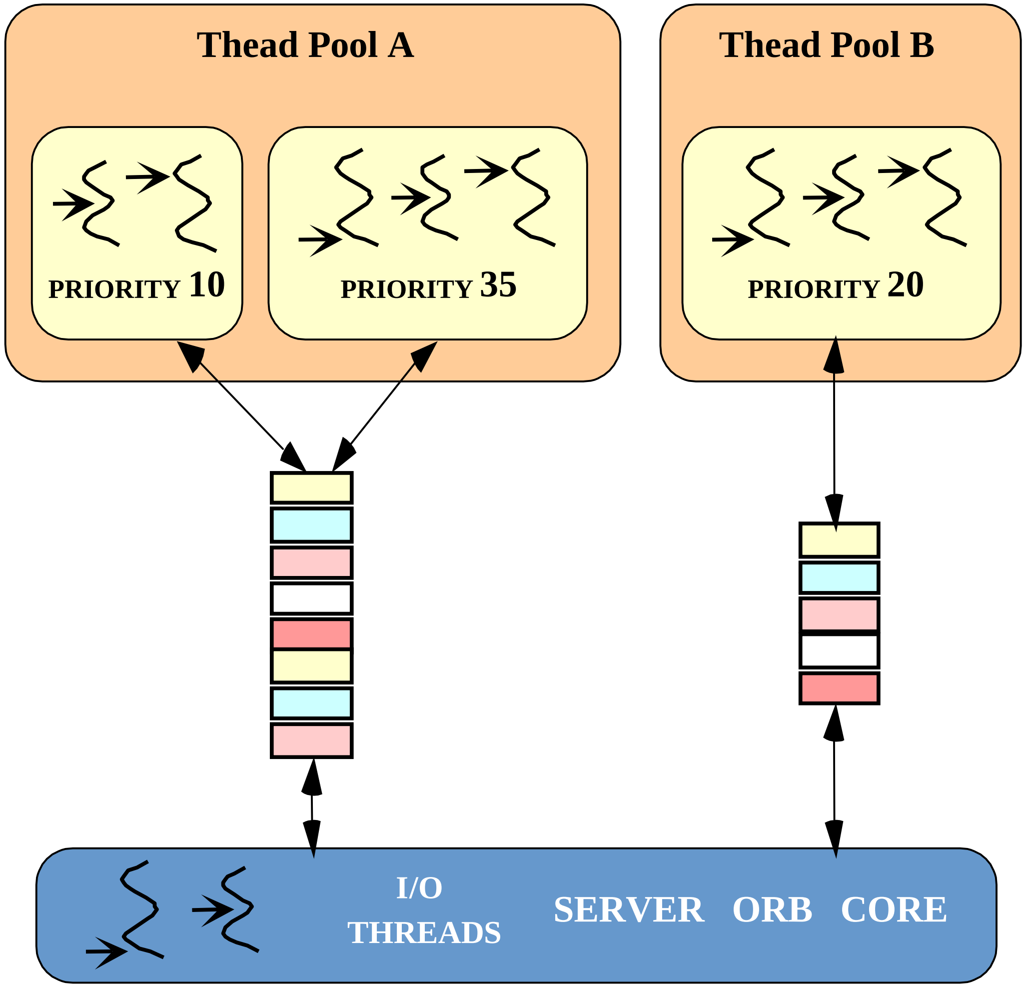

**Borrowing**

- Lane borrows thread from a lower priority lane when it exhausts its maximum number of static and dynamic threads

**Restoring**

- Priority is raised when thread is borrowed
- When there are no more requests, borrowed thread is returned and priority is restored

# Buffering Client Requests




**Handle "bursty" client traffic**

- Some applications need more buffering than is provided by the OS I/O subsystem

**Flexible configuration**

- Buffer capacities can be configured according to:
  1. Maximum number of bytes and/or
  2. Maximum number of requestsi

# Evaluating Thread Pools Implementations

- RT-CORBA spec underspecifies many quality of implementation issues
  - Thread pools, memory, & connection management
  - Maximizes freedom of RT-CORBA developers
  - Requires application developers to understand ORB implementation
  - Affects schedulability, scalability, & predictability of their application
- Examine patterns underlying common strategies for implementing thread pools
- Evaluate each thread pool strategy in terms of:

**1. Feature support**

Request buffering and thread borrowing

**2. Scalability**

Endpoints and event demultiplexers required

**3. Efficiency**

Data movement, context switches, memory allocations, & synchronizations required

**4. Optimizations**

Stack & thread specific storage memory allocations

**5. Priority inversion**

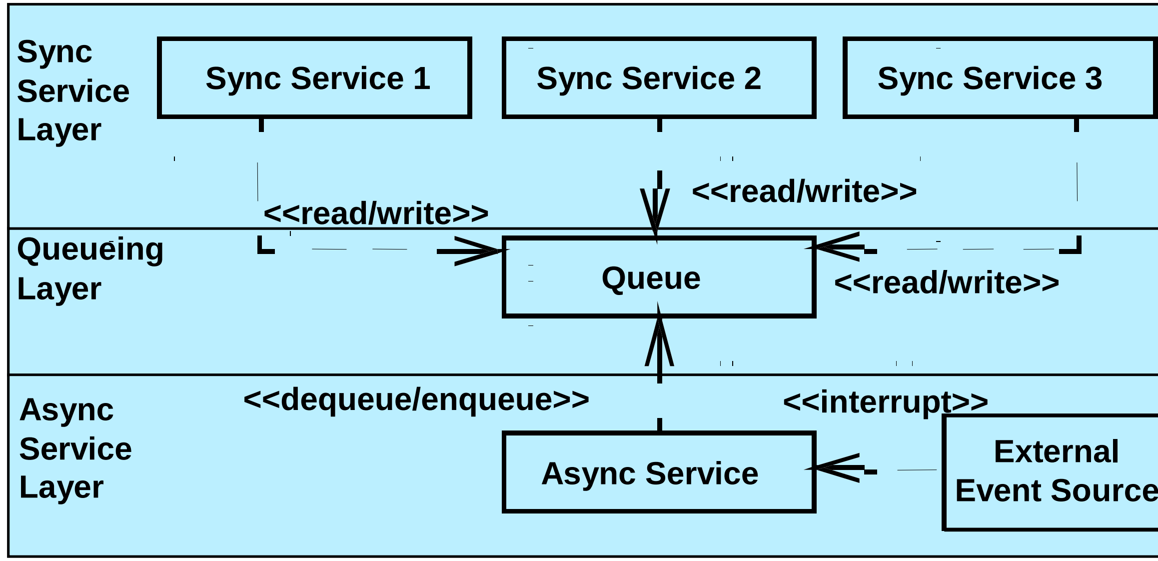
Bounded & unbounded priority inversion incurred in each implementation

# Thread Pools Implementation Strategies

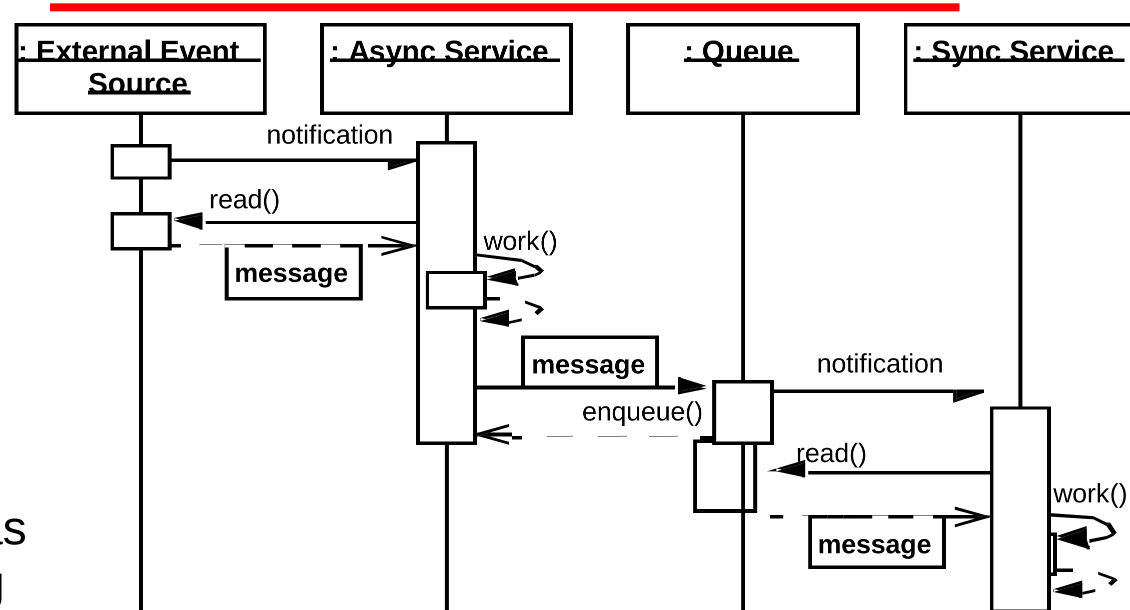
- There are two general strategies to implement RT CORBA thread pools:
  1. Use the *Half-Sync/Half-Async* pattern to have I/O thread(s) buffer client requests in a queue & then have worker threads in the pool process the requests
  2. Use the *Leader/Followers* pattern to demultiplex I/O events into 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

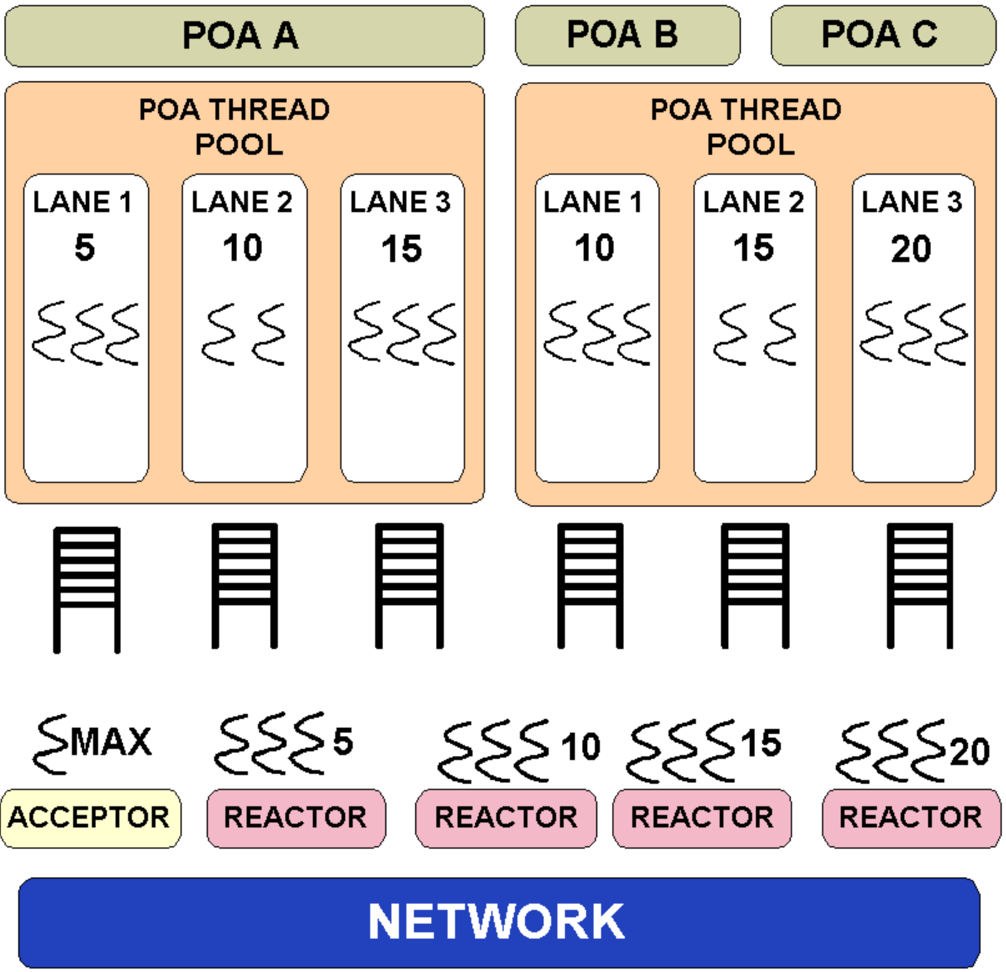
**Intent**
The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance




- 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**

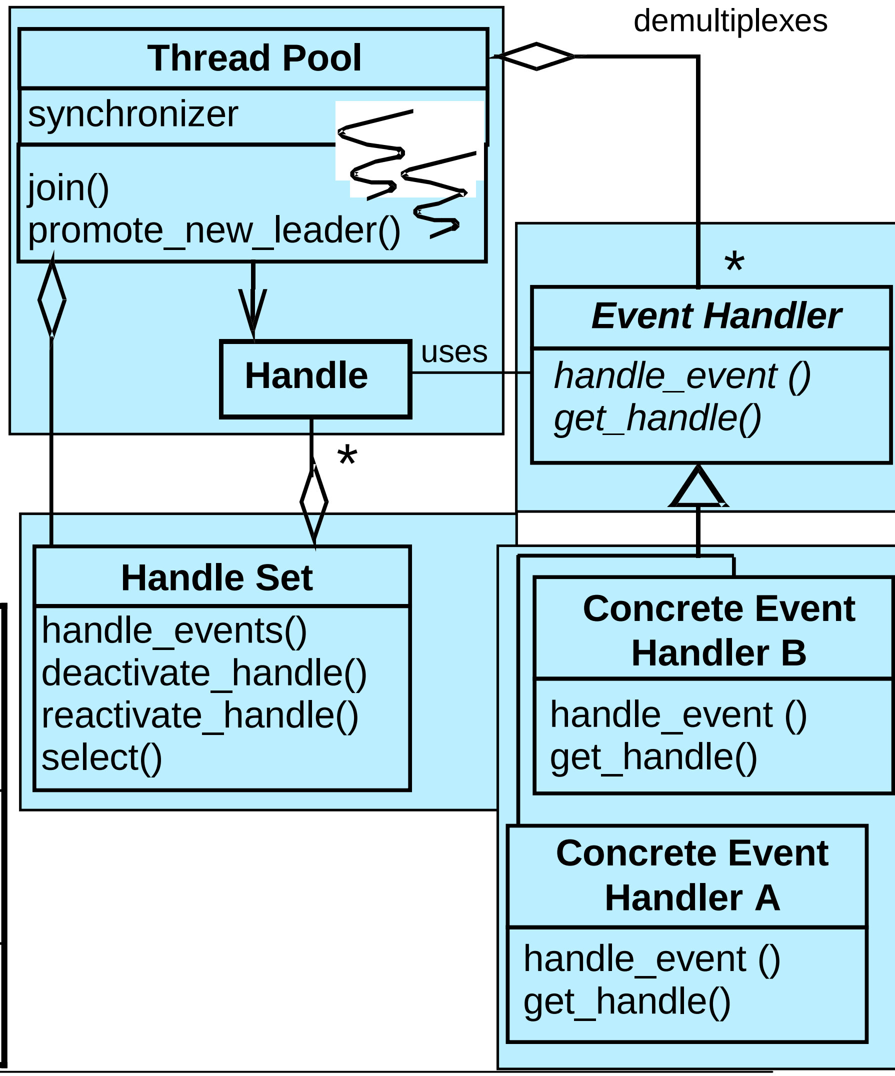
- Less efficient because of queuing
- Predictability reduced without _bind_priority_band() implicit operation

# The Leader/Followers Pattern

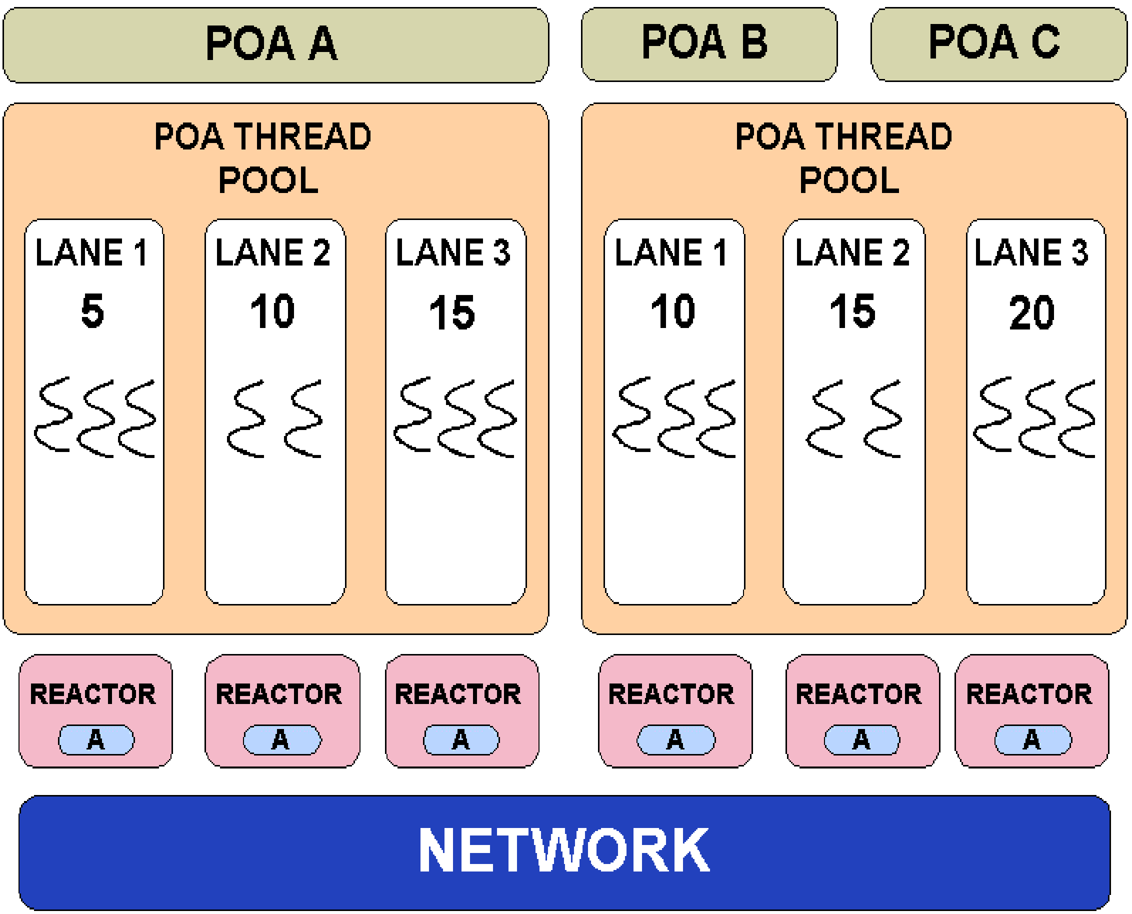
## Intent

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

| Handles / Handle Sets | Concurrent Handles | Iterative Handles |
|---|---|---|
| Concurrent Handle Sets | UDP Sockets + **`WaitForMultipleObjects()`** | TCP Sockets + **`WaitForMultpleObjects()`** |
| Iterative Handle Sets | UDP Sockets + **`select()/poll()`** | 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 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

# 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