Evaluating and Optimizing Thread Pool Implementations for RT-CORBA

Irfan Pyarali, Marina Spivak, and Ron Cytron
{irfan,marina,cytron}@cs.wustl.edu
Computer Science Dept.
Washington University,
St. Louis, MO

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering Dept.
University of Irvine,
Irvine, CA

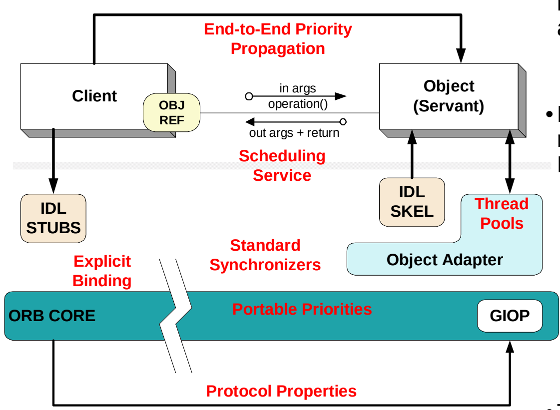
`http://www.cs.wustl.edu/~doc/`

Wednesday, June 13, 2001

Presentation Outline

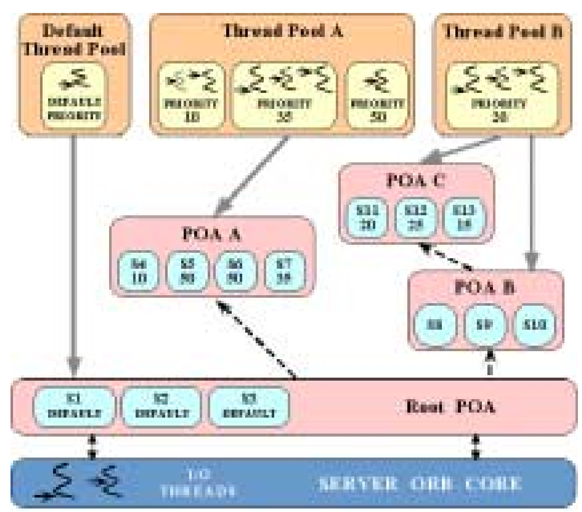
- Real-Time CORBA specification
- Thread Pools in Real-Time CORBA
- Requirements and features of Thread Pools
- Two strategies for implementing Thread Pools
- Evaluation of the strategies
- Conclusions and future work

Real-Time CORBA Overview



- 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 in RT-CORBA



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

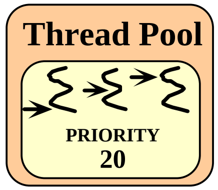
Scheduling
- Strict control over processor resources essential for many RT applications

Creating & Destroying Thread Pools

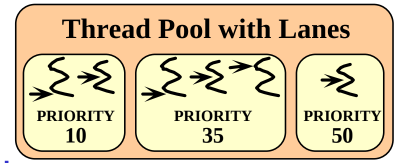
```
interface RTCORBA::RTORB {
   typedef unsigned long ThreadpoolId;

   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);
};
```

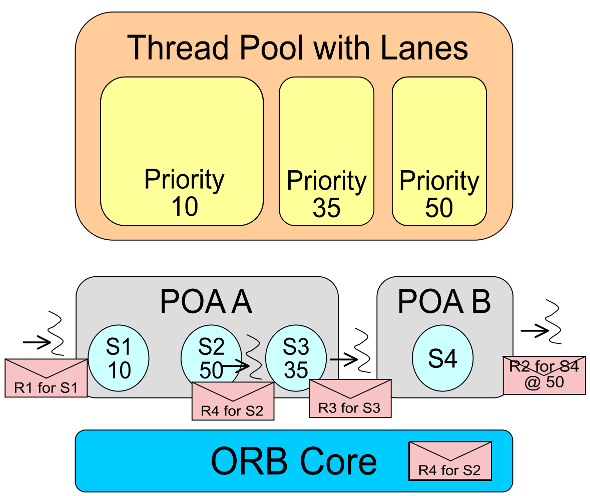


Creating Thread Pools with Lanes




```
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 );
};
```


Thread Borrowing



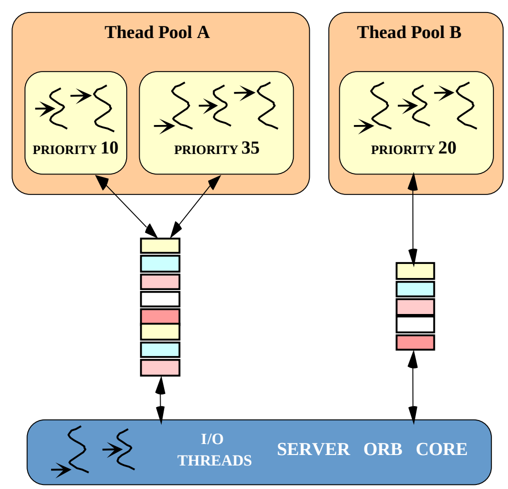
Borrowing

- 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 requests

Evaluating Thread Pools Implementations

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

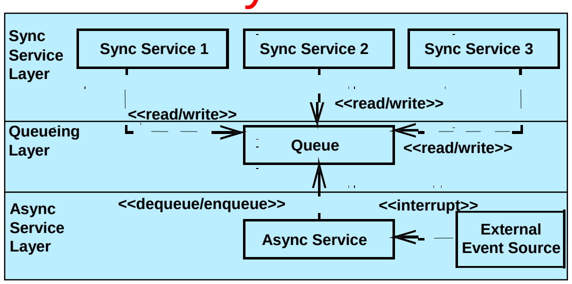
Capability	Description
Feature support	Request buffering and thread borrowing
Scalability	Endpoints and event demultiplexers required
Efficiency	Data movement, context switches, memory allocations, & synchronizations required
Optimizations	Stack & thread specific storage memory allocations
Priority inversion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

Thread Pools Implementation Strategies

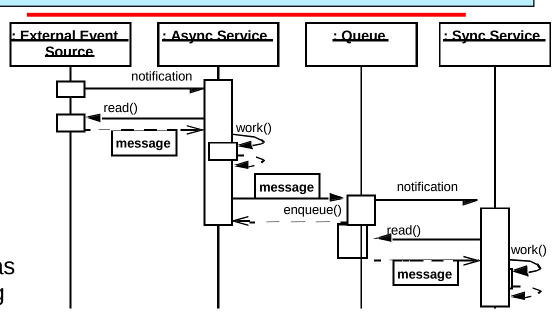
- There are two general strategies to implement RT CORBA thread pools:
  1. Use the *Half-Sync/Half-Async* pattern to have I/O thread(s) buffer client requests in a queue & then have worker threads in the pool process the requests
  2. Use the *Leader/Followers* pattern to demultiplex I/O events into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

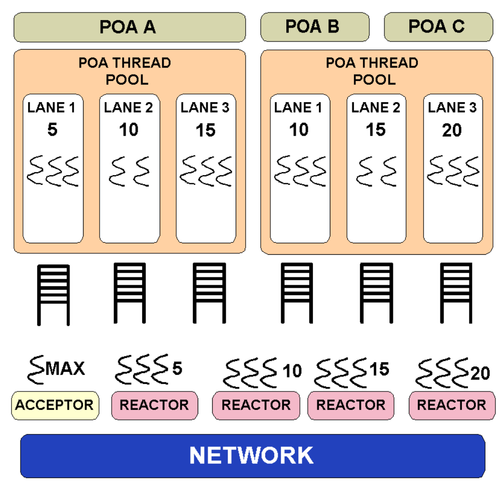
Intent
The *Half-Sync/Half-Async* architectural pattern decouples async & sync service processing in concurrent systems, to simplify programming without unduly reducing performance



- 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 = Smaller IORs
  - Fewer reactors
- Easier piece-by-piece integration into the ORB

Cons
- User has no control over I/O threads
- Queuing adds to overhead
- Predictability reduced without _bind_priority_band() implicit operation

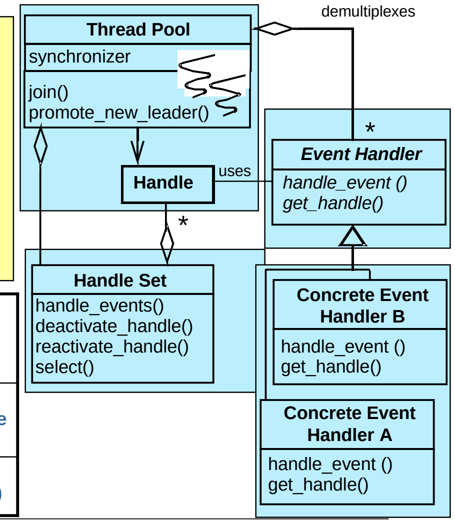
Evaluation of Half-Sync/Half-Async Thread Pools

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

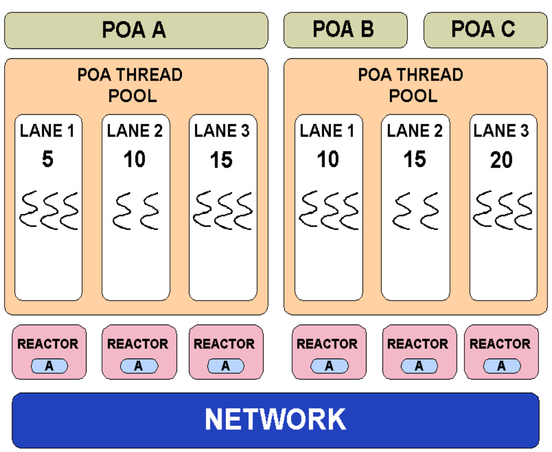
The Leader/Followers Pattern

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

Handles / Handle Sets	Concurrent Handles	Iterative Handles
Concurrent Handle Sets	UDP Sockets + `WaitForMultipleObjects()`	TCP Sockets + `WaitForMultipleObjects()`
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, etc.
- I/O & application-level request processing are done in the same thread

Pros
- 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 and TSS memory supported
Priority Inversion	***Good:*** little or no priority inversion

Concluding Remarks & Future Work

- RT CORBA 1.0 specifies thread pool creation & management
  - Only thread pools are specified
  - Thread-per-connection & thread-per-request not specified
  - Multi-threading previously done in CORBA through proprietary mechanisms
- RT Thread pools can be used to:
  - Leverage multi-processors hardware
  - Increase performance by overlapping I/O & computation
  - Supports different levels of service: differentiate between high & low-priority tasks
  - Supports preemption & prevent unbounded priority inversion
  - Supports scheduling by controlling processor resources
- Spec compliance of different thread pool implementations
  - Multiple endpoints used as hints
  - Connections for ORBs that don’t use endpoint hint can be moved to correct priority during the binding or first request
- Portions of spec are under-specified
  - Developers must be familiar with the implementation decisions made by their RT ORB because it effects schedulability, scalability, & predictability of their application
- Future work
  - Complete Leader/Followers Thread Pool implementation
  - Carefully instrument code to make sure there are no cases of unbounded priority inversion