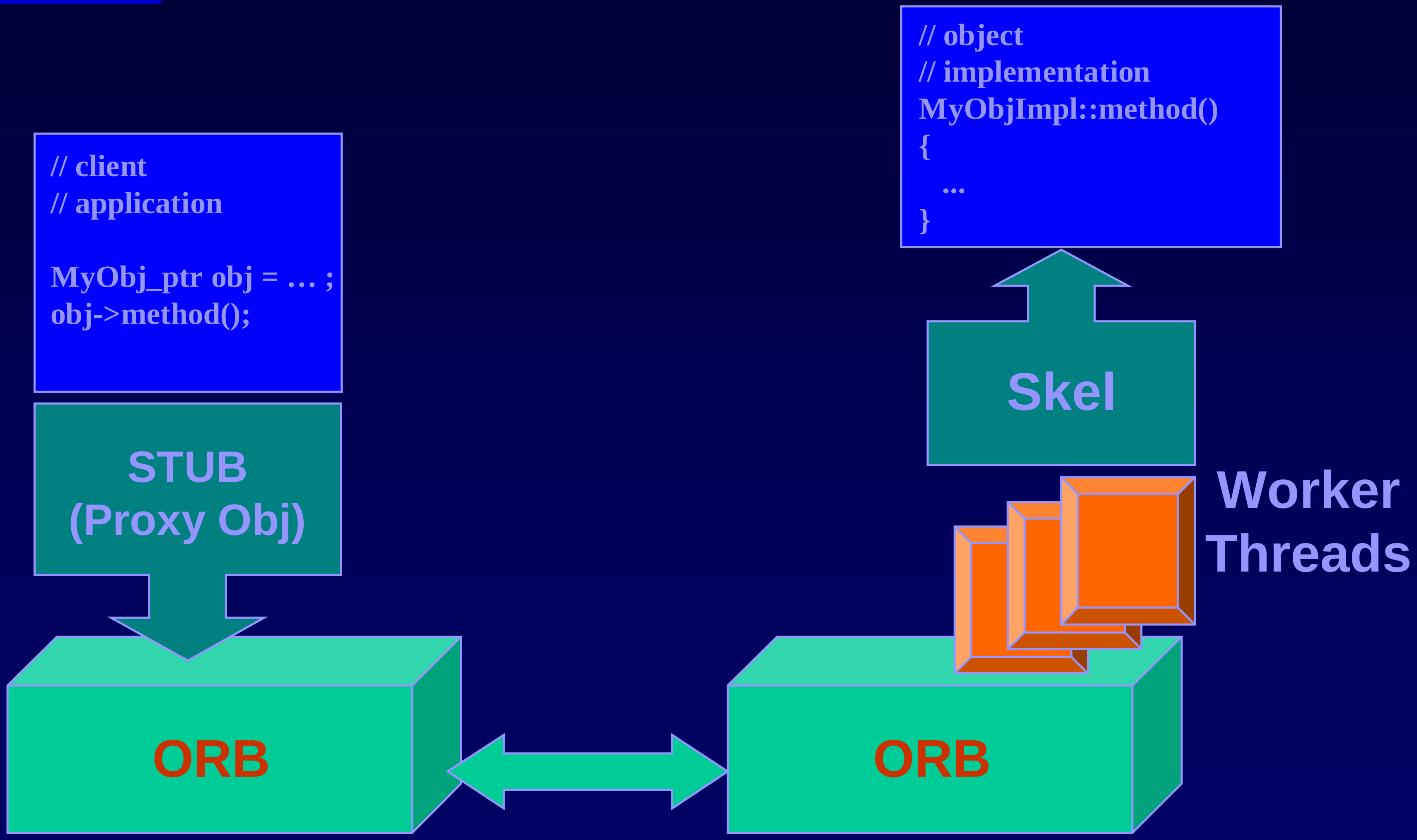
The Real-Time CORBA Specification tutorial Part-2

Contributor:
Jon Currey
Senior Systems Engineer
Highlander Engineering Inc.
jon@highlander.com
1-863-686-7767

Real-Time CORBA 1.0

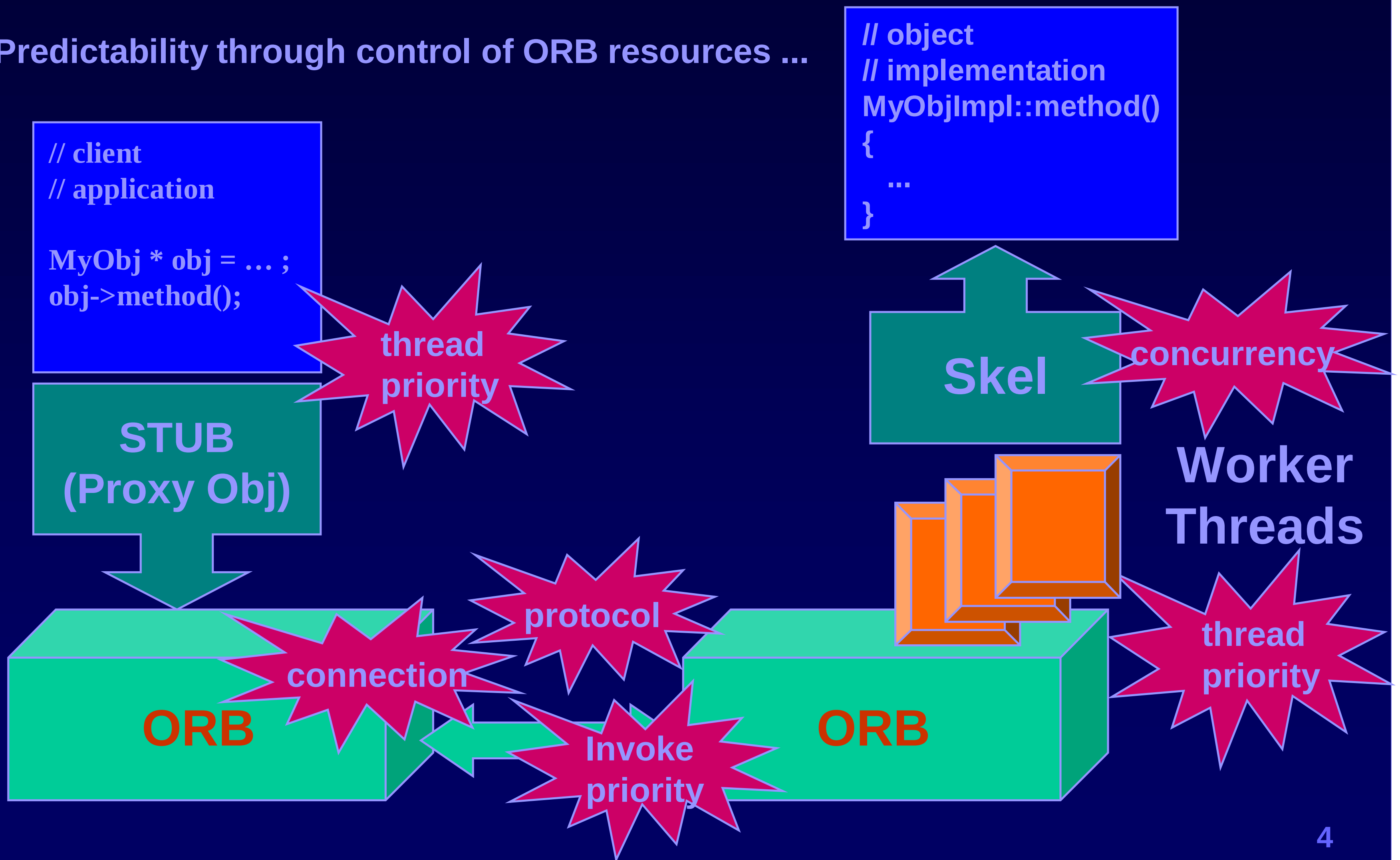
- **OMG Specification Chapter 24**
  www.omg.org/technology/documents/spec_catalog.htm
- **Now Part of CORBA 2.4.2 Specification**

Why Real-Time CORBA?



Why Real-Time CORBA?

Predictability through control of ORB resources ...



Scope of Real-Time CORBA

- Addresses predictability of ORB operations
- Just one component in a Real-Time System
  - application, operating system, transport protocol(s), hardware, device drivers … all affect predictability
- Real-Time CORBA 1.0 addresses Fixed Priority Real-Time Systems
  - Priority-based scheduling, rather than e.g. deadline based
  - Covers a significant portion of RTOS based development
  - Real-Time CORBA 2.0 will address Dynamic Scheduling

Control of ORB-Related System Resources

CPU Resources

Prioritized CORBA invocations

'Threadpools'

Bounding of ORB Thread Priorities

Network Resources

Protocol Selection and Configuration

Connection Management

Memory Resources

Buffering of Requests

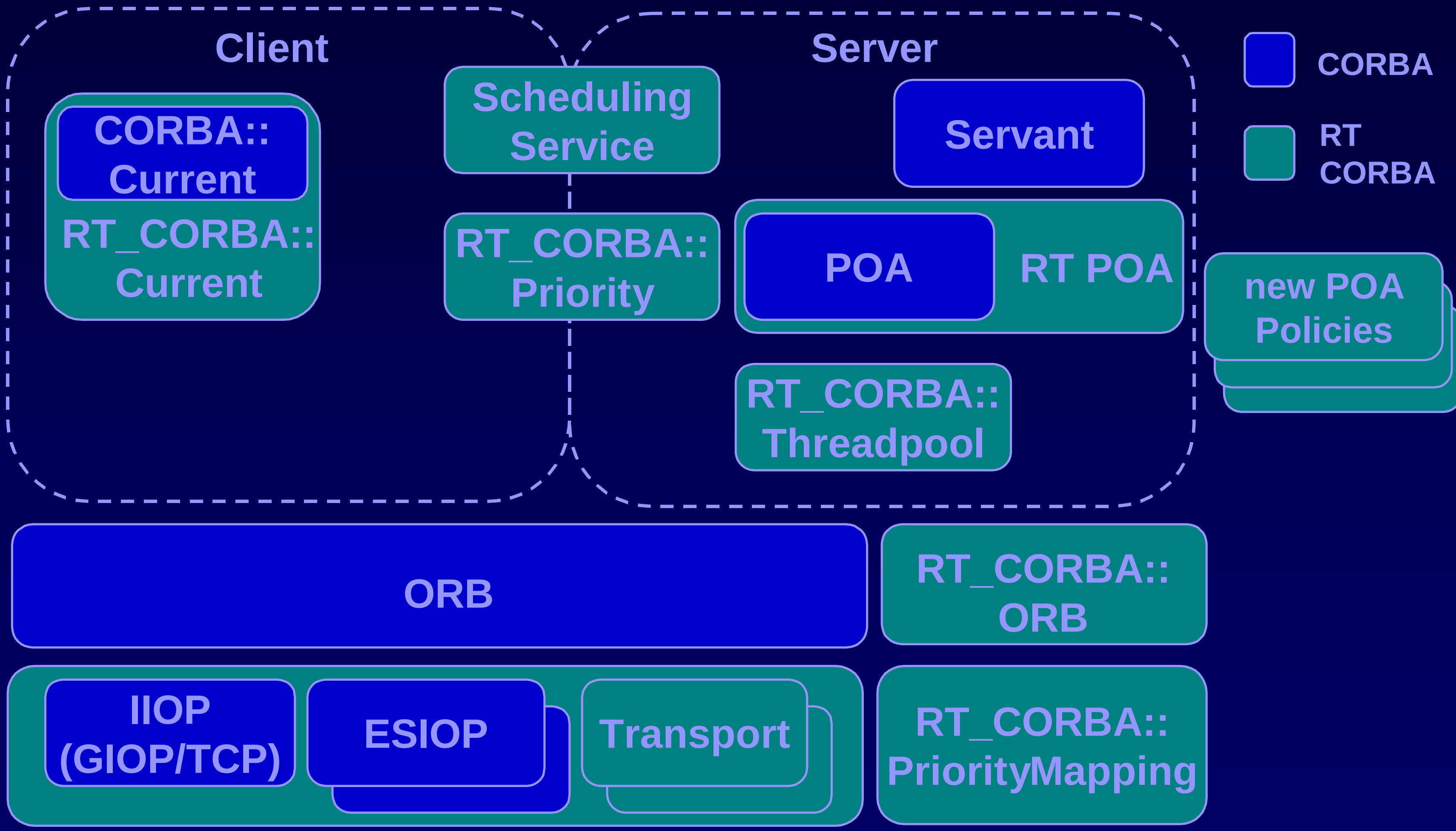
+ Thread abd Network Resource Control

Scheduling Service (optional)

API for using off-line scheduling analysis

e.g. with tools

Real-Time CORBA Extensions



Agenda

Real-Time CORBA Rationale

Real-Time CORBA Features and API

Real-Time CORBA Code Examples

Real-Time CORBA Features

- Real-Time CORBA ORB & POA
- Real-Time CORBA Priority & Priority Mappings
- Real-Time CORBA Priority Models
- Real-Time CORBA Mutex
- Threadpools
- Protocol Selection and Configuration
- Connection Management
- Bounding of ORB Thread Priorities
- Scheduling Service

RTCORBA::RTORB

- Consider as an extension of the CORBA::ORB interface
- Adds operations to create and destroy other Real-Time CORBA entities
  - Mutex, Threadpool, Real-Time Policies
- One RTORB per ORB instance
- Obtain using

```
orb->resolve_initial_references("RTORB");
```

RTPortableServer::POA

- Critical Central focus of the RTCORBA Server Side Mapping
- It is an extension to the normal POA interface

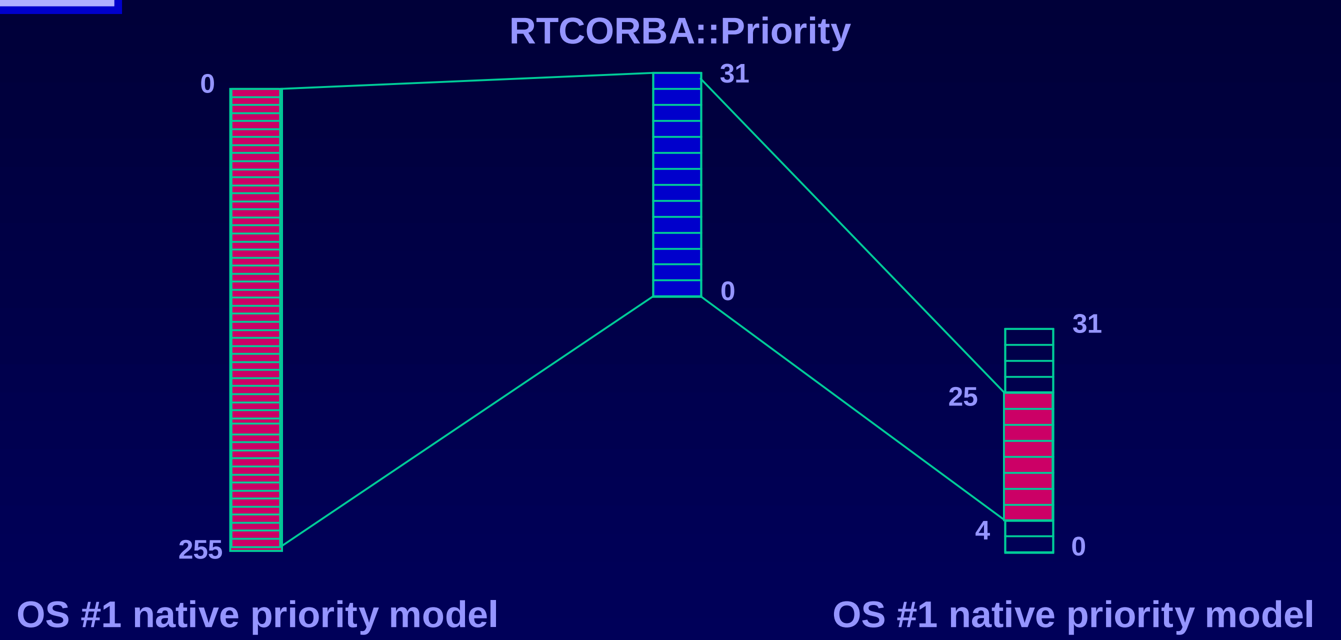
```
// IDL
   module RTPortableServer {

     interface POA : PortableServer::POA {
       // new operations here …
     };

   };
```

- Adds operations to allow setting of priority on a per-Object basis

Real-Time CORBA Priority



- An OS independent priority scheme
  - allows system design using a single, ‘global’ priority scheme, in a heterogeneous platform environment
- Priority ‘Mappings’ can be customized for each system’s needs

Types Supporting Real-Time CORBA Priority

RTCORBA::Priority

RTCORBA::NativePriority

RTCORBA::PriorityMapping

RTCORBA::Priority

```
// IDL
module RTCORBA {
   typedef short Priority;
   const Priority minPriority = 0;
   const Priority maxPriority = 32767;
};
```

Universal, platform independent priority scheme

- Allows prioritized CORBA invocations to be made in a consistent fashion between nodes with different native priority schemes

‘Global’ Priority scheme

- simplifies system design, code portability, extensibility
- use for schedulability analysis, perhaps with tools

RTCORBA::PriorityMapping

```
module RTCORBA {
   typedef short NativePriority;
   native PriorityMapping;
};
```

- NativePriority type is defined to represent OS specific native priority scheme
- A PriorityMapping defines a mapping between RTCORBA::Priority and NativePriority
  - Specified as a ‘native’ type for efficiency and simplicity

RTCORBA::PriorityMapping

- Language mappings specified for C, C++, Ada and Java
- Each specifies *to_native* and *to_CORBA* operations

```
// C++
namespace RT_CORBA {
    class PriorityMapping {
        public:
            virtual CORBA::Boolean to_native (
                RT_CORBA::Priority corba_priority,
                RT_CORBA::NativePriority &native_priority );
            virtual CORBA::Boolean to_CORBA (
                RT_CORBA::NativePriority native_priority,
                RT_CORBA::Priority &corba_priority );
    };
};
```

RTCORBA::PriorityMapping

- One PriorityMapping installed at any one time per ORB instance
  - installation mechanisms are not standardized
  - left as an implementation choice. e.g.link-time and/or run-time
- The default PriorityMapping is not standardized
  - would be platform and application-domain specific
  - the default is likely to be overridden anyway
- A particular PriorityMapping may choose to map only a sub-range of native or CORBA Priorities
  - e.g. only use RTCORBA::Priority values 0 to 31 (ala POSIX) and/or only map onto a subset of the native priority range

Real-Time CORBA Priority Models

Two models for handling of RTCORBA::Priority during invocations

- Client Propagated Model
- Server Declared Model

Priority Model Policy

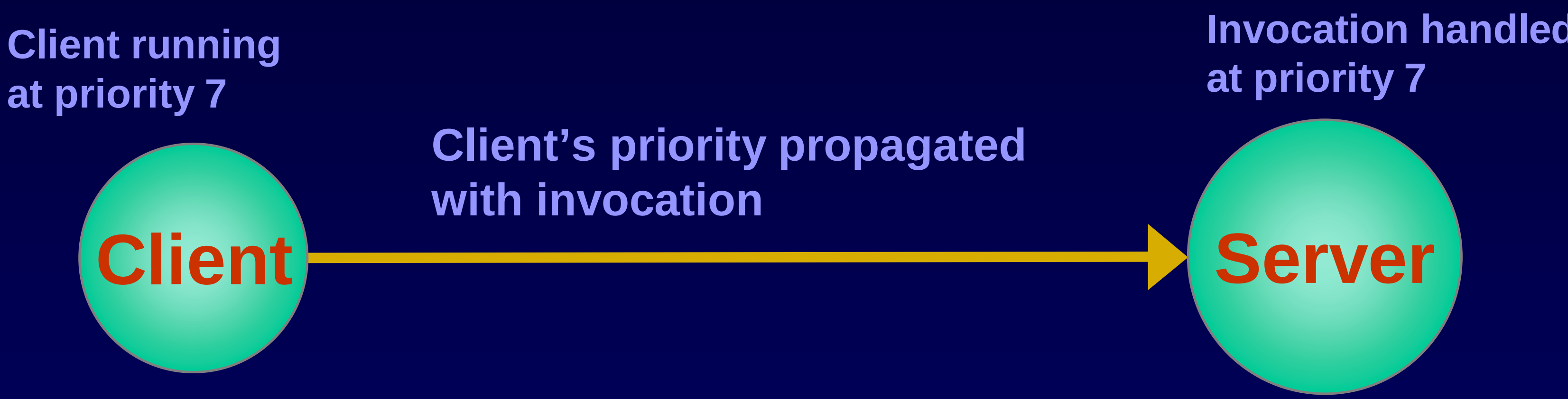
A Server-Side (POA) Policy

- configure by adding a PriorityModelPolicy to policy list parameter of POA_create operation
- all objects from a given POA support the same model

```
// IDL
enum PriorityModel {
    CLIENT_PROPAGATED,
    SERVER_DECLARED
};

interface PriorityModelPolicy : CORBA::Policy {
    readonly attribute PriorityModel priority_model;
    readonly attribute Priority server_priority;
};
```

Client Propagated Priority Model



scheduling based on priority of an activity, propagated and honored along the path of that activity through the system

RTCORBA::Current

- Used to assign a RTCORBA::Priority to the current thread of execution
  - Mapped to a change in underlying native thread priority via *to_native* operation of active PriorityMapping
  - Also determines RTCORBA::Priority value passed with invocations in the Client Propagated Priority Model

```
//IDL
module RTCORBA {
  interface Current : CORBA::Current {
    attribute RTCORBA::Priority the_priority
  };
};
```

RTCORBA::Current

- Obtained with a call to *CORBA::ORB::resolve_initial_references*, with ObjectId “RTCurrent”

- Operates in a ‘thread specific’ manner
  - so a single instance can be used by multiple threads

```
// C++
CORBA::Object_var ref = orb->resolve_initial_references(“RTCurrent”);
RTCORBA::Current_ptr rtcurrent = RTCORBA::RTCurrent::_narrow(ref);
rtcurrent->the_priority(7);
```

Priority Propagation Mechanism

The RTCORBA::Priority is passed in a RTCorbaPriority service context associated with the invocation

This allows prioritized invocations to be made between different ORB products

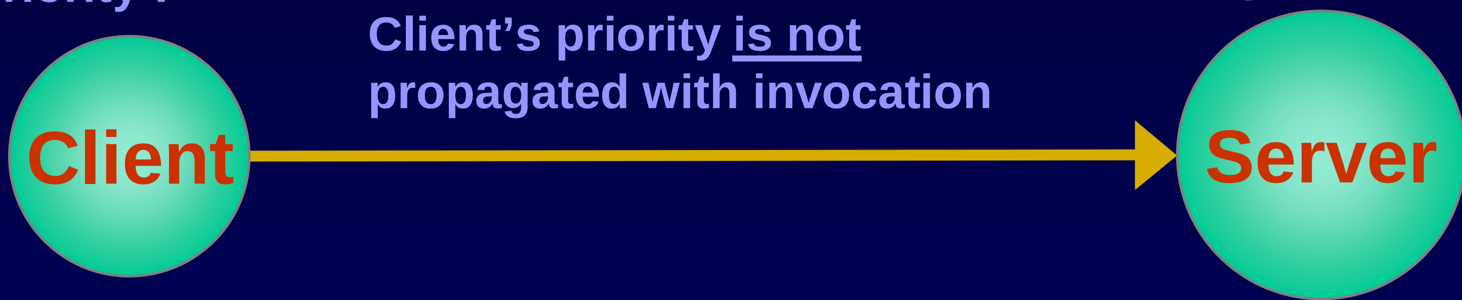
```
module IOP {
    const ServiceId RTCorbaPriority = ??;
        // value assigned by OMG
};
```

Server Declared Priority Model

Client running at priority 7

Server Priority is pre-set

Client's priority <u>is not</u> propagated with invocation



Invocation handled at the pre-set Server priority

- scheduling based on relative priorities of different objects (servers) on the same node
- a particular server or set of servers handles all invocations at a particular priority

Setting Server Priority

- PriorityModelPolicy instance that selected the Server Declared model contains a priority value
  - used as default server priority for all objects created by that POA

```
interface PriorityModelPolicy : CORBA::Policy {
    readonly attribute PriorityModel priority_model;
    readonly attribute Priority server_priority;
};
```

- Operations on RTPOA allow setting of server priority on a per Object basis ...

Setting Server Priority on per-Object Basis

```
// IDL
module RTPortableServer {

  interface POA : PortableServer::POA {// locality constrained

    Object create_reference_with_priority (
                  in CORBA::RepositoryId intf,
                  in RTCORBA::Priority priority )
        raises (WrongPolicy);

    Object create_reference_with_id_and_priority (
                  in ObjectId oid,
                  in CORBA::RepositoryId intf,
                  in RTCORBA::Priority priority )
        raises (WrongPolicy);
```

Setting Server Priority on per-Object Basis

```
    ObjectId activate_object_with_priority (
                    in Servant p_servant,
                    in RTCORBA::Priority priority )
        raises (ServantAlreadyActive, WrongPolicy);

    void activate_object_with_id_and_priority (
                    in ObjectId id,
                    in Servant p_servant,
                    in RTCORBA::Priority priority )
        raises ( ServantAlreadyActive,
                 ObjectAlreadyActive, WrongPolicy);
  };
};
```

Real-Time CORBA Mutex

- API that gives the application access to the same Mutex implementation that the ORB is using
  - important for consistency in using a Priority Protocol e.g. Priority Inheritance or Priority Ceiling Protocol
- The implementation must offer (at least one) Priority Protocol
  - No particular protocol is mandated though
  - application domain and RTOS specific

RTCORBA::Mutex

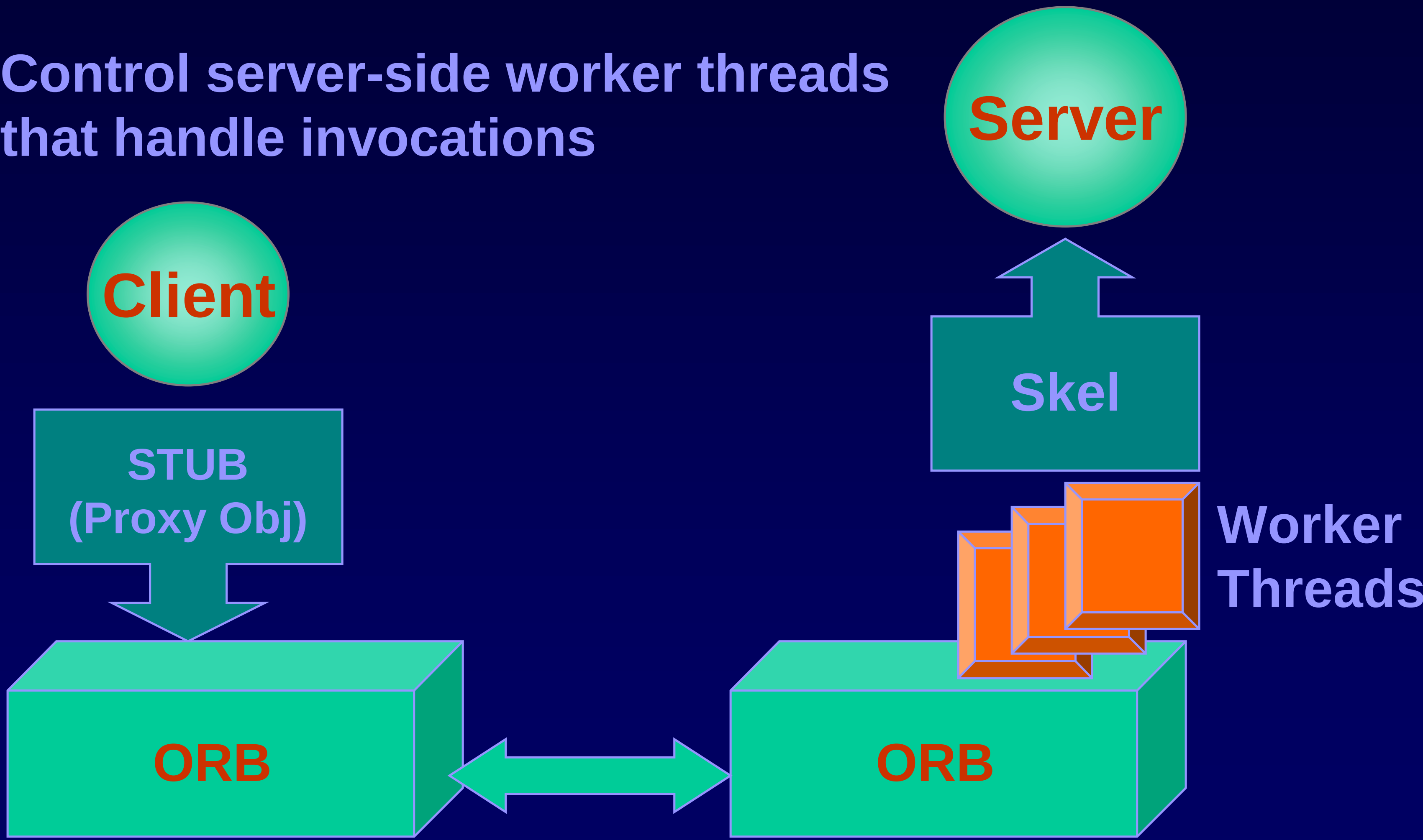
```
module RTCORBA {
 interface Mutex {
     void lock();
     void unlock();
     boolean try_lock( in TimeBase::TimeT max_wait);
 };

 interface RTORB {
  Mutex create_mutex();
  void destroy_mutex( in Mutex the_mutex );
 };
};
```

Instances are obtained through *create_mutex* operation on RTCORBA::RTORB

Threadpools

Control server-side worker threads that handle invocations



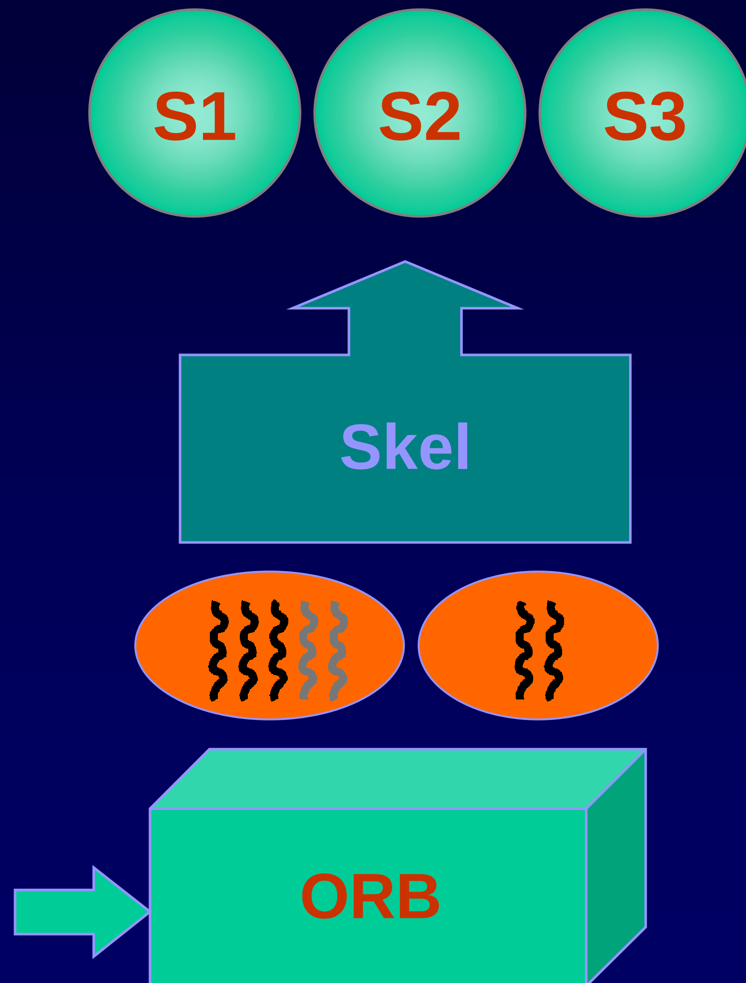
Threadpools

Threadpool Benefits

- Control invocation concurrency
- Thread pre-creation and reuse
- Configure idle thread priorities

Multiple Threadpools ⋗

- System partitioning
- Protect independent sub-systems
- Integrate different systems more predictably



Threadpools

- Threadpool abstraction is used to manage threads on server-side of Real-Time CORBA ORB
  - pre-allocation, partitioning, bounding usage: predictability

- Threadpool parameters
  - number of static threads
  - dynamic thread limit
    - 0 = no limit. same value as static = no dynamic threads
  - thread stacksize
  - default thread priority
    - thread priority will change as required

Threadpool IDL

```
module RTCORBA {
  typedef unsigned long ThreadpoolId;

  interface RTORB {
    ThreadpoolId create_threadpool (
                in unsigned long stacksize,
                in unsigned long static_threads,
                in unsigned long max_threads,
                in Priority default_priority,
                in boolean allow_request_buffering,
                in unsigned long max_buffered_requests,
                in unsigned long max_request_buffer_size );

    ThreadpoolId create_threadpool_with_lanes ( … );

    void destroy_threadpool (
                in ThreadpoolId threadpool )
  };
};
```

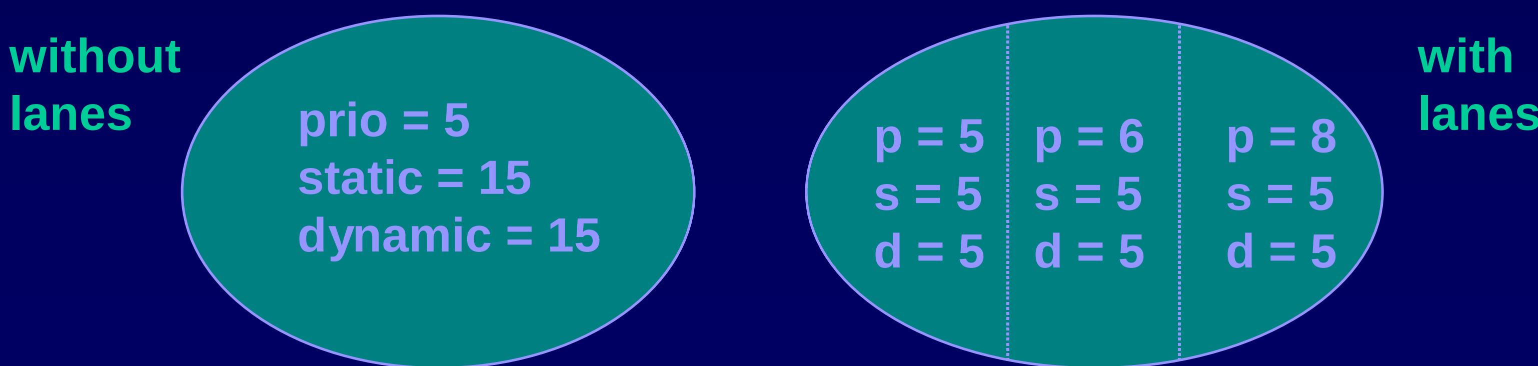
Threadpool Policy

- Server-side (POA) policy, used to associate a POA with a particular Threadpool
- ThreadpoolId allows same pool to be shared by multiple POAs

```
module RTCORBA {
  interface ThreadpoolPolicy : CORBA::Policy {
      readonly attribute ThreadpoolId threadpool;
  };
};
```

Laned Threadpools

- Alternate way of configuring a Threadpool
  - for applications with detailed knowledge of priority utilization
  - preconfigure ‘lanes’ of threads with different priorities
  - ‘borrowing’ from lower priority lanes can be permitted

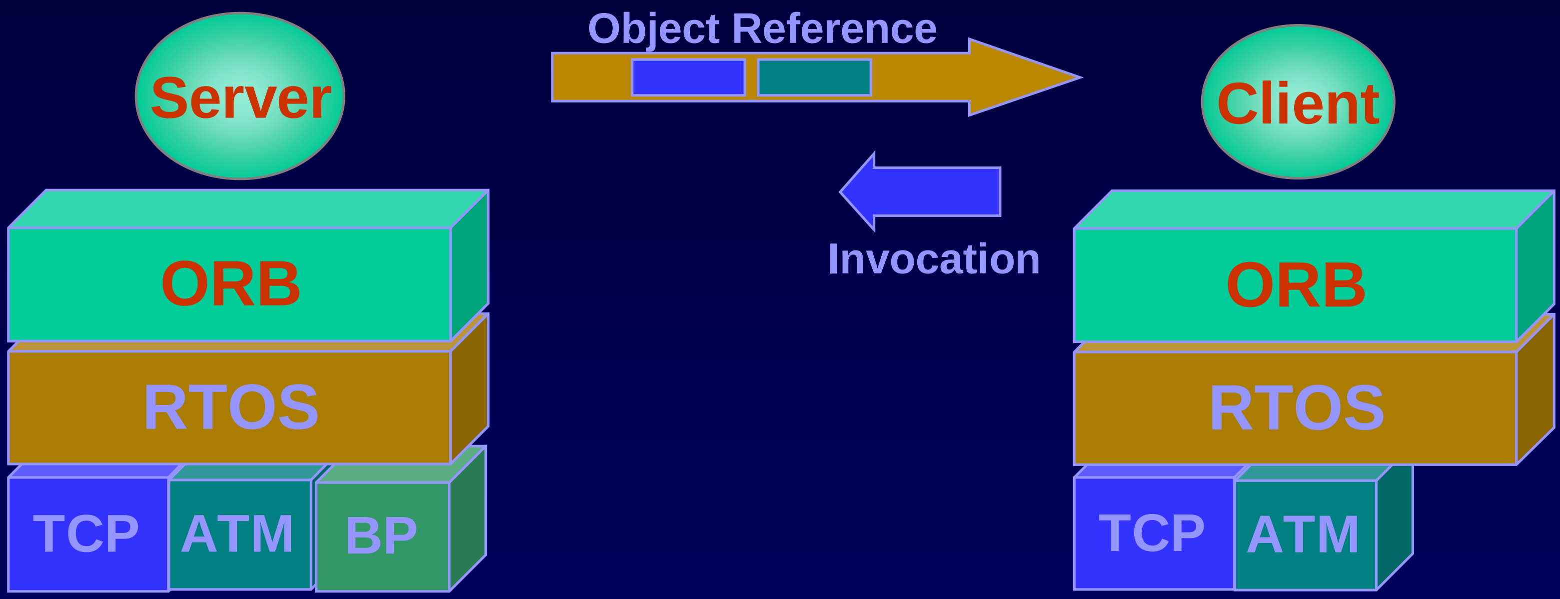


Laned Threadpool IDL

```
module RTCORBA {
    struct ThreadpoolLane {
        Priority lane_priority;
        unsigned long static_threads;
        unsigned long dynamic_threads;
     };
    typedef sequence <ThreadpoolLane> ThreadpoolLanes;

    interface RTORB {
      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 );
};
```

Protocol Selection and Configuration



Server-side

Which protocol(s) to publish in Object Reference

Protocol configuration

Client-side

Which protocol to connect to Object via

Protocol configuration

ServerProtocolPolicy

- Enables selection and configuration of communication protocols on a per-POA basis
- Protocols are represented by RTCORBA::Protocol type
  - Protocols defined as ORB/Transport level protocol pairs
- RTCORBA::ProtocolList allows multiple protocols to be supported by one POA
  - Order of protocols in list indicates order of preference

ServerProtocolPolicy

```
module RTCORBA {

  struct Protocol {
      IOP::ProfileId protocol_type;
      ProtocolProperties orb_protocol_props;
      ProtocolProperties trans_protocol_props;
    };
  typedef sequence <Protocol> ProtocolList;

    interface ServerProtocolPolicy : CORBA::Policy {
      readonly attribute ProtocolList protocols;
    };

};
```

ProtocolProperties

- A ProtocolProperties interface to be provided for each configurable protocol supported
  - allows support for proprietary and future standardized protocols
- Interfaces are derived from a base interface type

```
interface ProtocolProperties {};
```

- Real-Time CORBA only specifies ProtocolProperties for TCP

TCPProtocolProperties

```
module {
  interface TCPProtocolProperties : ProtocolProperties {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
  };
};
```

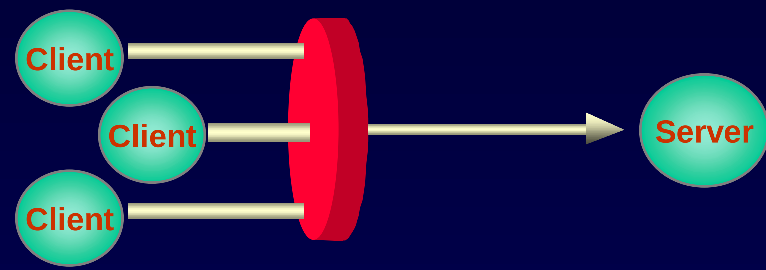
ClientProtocolPolicy

- Same syntax as server-side
  - RTCORBA::Protocol, ProtocolProperties, ProtocolList
- On client, ProtocolList specifies protocols that may be used to make a connection
  - order indicates order of preference
- If ProtocolPolicy not set, order of protocols in target object's IOR used as order of preference

Connection Management

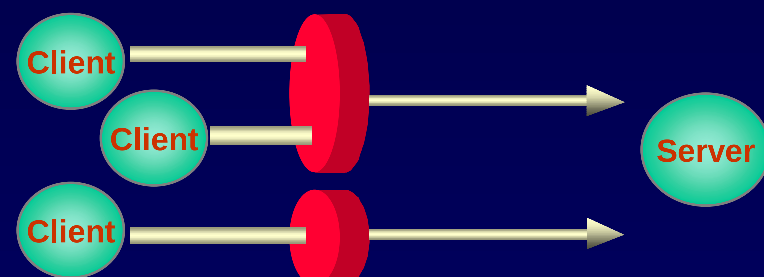
Connection Multiplexing

- Offered by most ORBs for resource conservation



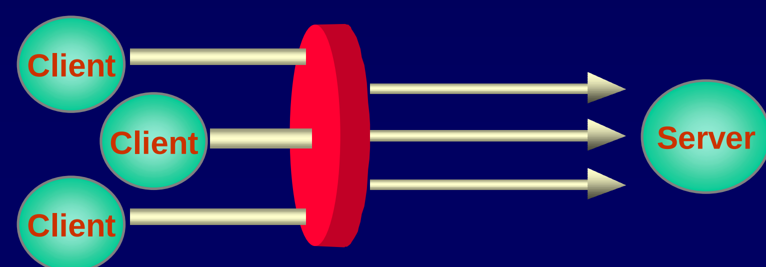
Private Connection Policy

- Guarantees separate connection for that client



Priority Banded Connections

- Several connections between nodes
- Invocation priority determines which connection used



Private Connection Policy

- **Allows a client to demand a private transport connection to the target object**
  - **no multiplexing with requests for other target objects within protocol resources controllable by ORB**

- **A client-side policy, applied through CORBA *set_policy_overrides* operation**

```
// C++
CORBA::Object_ptr ref = // Object reference for target object
CORBA::PolicyList policy_list(1);
policy_list[0] = private_connection_policy;

CORBA::Object_var new_ref = ref->set_policy_overrides( policy_list,
                                                        CORBA::ADD_OVERRIDE);
```

Priority Banding

Multiple connections, to reduce priority inversion

– each connection handling different priority invocations

Banding

– each connection may represent a range of priorities, to allow resources to be traded off against limited inversion

– may have different ranges in each band, including range of 1

PriorityBandedConnectionPolicy

```
module RTCORBA {
  struct PriorityBand {
    Priority low;
    Priority high;
  };

  typedef sequence <PriorityBand> PriorityBands;

  interface PriorityBandedConnectionPolicy : CORBA::Policy {
      readonly attribute PriorityBands priority_bands;
  };
};
```

- Applied on *server-side or client-side*
- Used on client-side, to establish connections at bind time

Bounding of ORB Thread Priorities

- Application may specify a range of CORBA Priorities that are available for ORB internal threads
  - standardizes some level of control over the ORB's use of priorities
  - affects all 'other' ORB threads, apart from Threadpool threads

- Specified at ORB initialization, via an ORB_init parameter

-ORBRTpriorityrange <min priority>,<max priority>

Next –
The Real-Time CORBA 1.0 Scheduling Service

Real-Time CORBA Scheduling Service

Contributor:
Tom Cox
Senior Systems Engineer
Tri-Pacific Software
tomc@tripac.com
1-972-620-2520

Why a Scheduling Service?

Effective Real-Time scheduling is complicated

To ensure a uniform scheduling policy, such as global Rate Monotonic Scheduling, requires:

- the Real-Time CORBA primitives must be used properly, and
- their parameters must be set properly in all parts of the CORBA system

Why a Scheduling Service?

The problem is made more acute by things like

- large system size
- changes to the system design
- porting the system

The Scheduling Service API abstracts away from the low-level Real-Time constructs

- simplifies the building and maintenance of schedulable systems
- allows use of scheduling analysis tools, that support the specified API

Real-Time CORBA Scheduling Service

A Scheduling Service implementation will choose:

- Real-Time CORBA Priorities,
- POA policies, and
- Priority Mappings

in such a way as to realize a uniform Real-Time scheduling policy

Different implementations can provide different Real-Time scheduling policies

The Scheduling Service Abstraction

Abstraction of scheduling parameters (such as Real-Time CORBA Priorities) is through the use of "names" (strings)

The system designer identifies:

- a static set of CORBA Activities,
- CORBA objects that the Activities use,
- scheduling parameters, such as Real-Time CORBA Priorities, for those Activities and objects,
- names that are uniquely assigned to those Activities and Objects

The Scheduling Service internally associates the names with the scheduling parameters and policies for the corresponding Activities and CORBA objects

Scheduling Service IDL

```
module RTCosScheduling {

    interface ClientScheduler {
            void schedule_activity(in string name)
                            raises(UnknownName);
    };

    interface ServerScheduler {
            PortableServer::POA create_POA (
                    in PortableServer::POA parent,
                    in string adapter_name,
                    in PortableServer::POAManager a_POAManager,
                    in CORBA::PolicyList policies)
                            raises ( PortableServer::POA::AdapterAlreadyExists,
                                    PortableServer::POA::InvalidPolicy );

            void schedule_object(in Object obj, in string name)
                            raises(UnknownName);
    };
};
```

Client-side Semantics

A CORBA client obtains a local reference to a *ClientScheduler* object

Whenever the client begins a region of code with a new deadline or priority (indicating a new CORBA Activity), it invokes *schedule_activity* with the name of the new activity

The Scheduling Service associates a Real-Time CORBA priority with this name and it invokes appropriate RT ORB and RTOS primitives to schedule this activity

Server-side Semantics

- A CORBA server obtains a local reference to a *ServerScheduler* object
- The *create_POA* method accepts parameters allowing it to create a POA
- This POA will enforce all of the non-Real-Time policies in the Policy List input parameter
- All Real-Time policies for the returned POA will be set internally by this scheduling service method

Server-side Semantics

- *schedule_object* is provided to allow the Scheduling Service to achieve object-level control over scheduling of the object
- RT POA policies in the RT ORB allow some control over the scheduling of object invocations, but must do so for all objects managed by each POA
- Some Real-Time scheduling policies, such as priority ceiling concurrency control, requires object-level scheduling

Next –
The Real-Time CORBA Code Examples