Real-Time Middleware

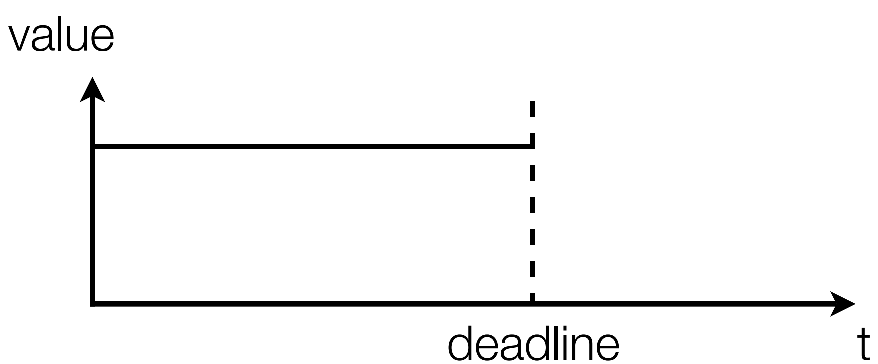
Dipl.-Inf. Andreas Rasche

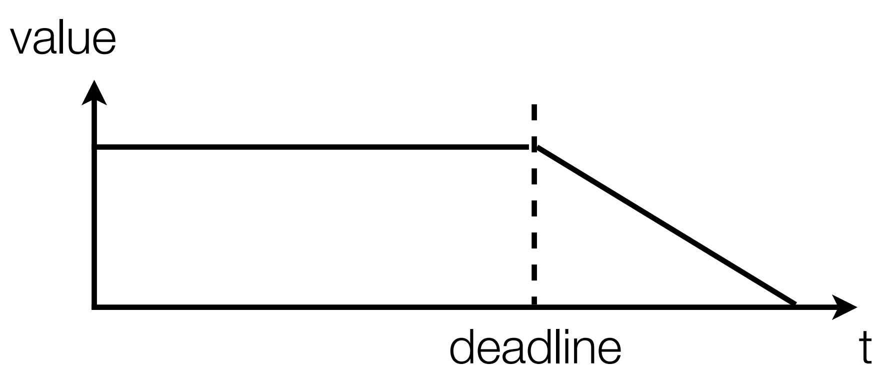
Roadmap

- Real-time Systems, Tasks, Scheduling, Priority Inversion
- Real-time CORBA Specification
- Distributed Real-time Specification for Java (D-RTSJ)
- Composite Objects
- Time-triggered Message-triggered Objects (TMO)
- OSA+

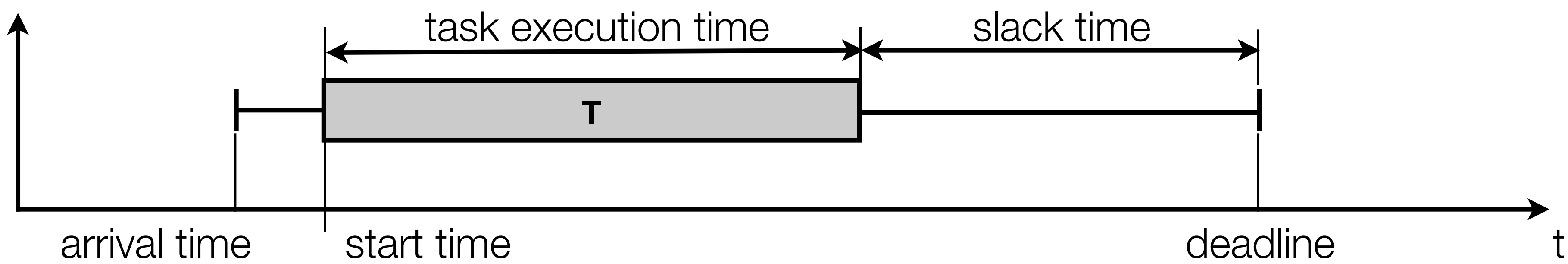
What is Real-Time ?

- “A system is a real-time system if the correctness of an operation depends not only upon the logical correctness but also upon the time at which it is performed.”
- Hard real-time: Missing a deadline could result in catastrophe
  - Flight control systems, drive-by-wire, avionics
- Soft real-time: Result arrival after deadline has still value
  - multi-media, booking systems





Tasks & Scheduling



- Scheduling: Find order for task execution so that every tasks meets its deadline
- periodic vs. aperiodic vs. sporadic tasks
- preemptive vs. non-preemptive execution
- static (priority-based) scheduling (RMS) vs. dynamic scheduling (EDF, LSF)
- task synchronisation & unbounded priority inversion / avoidance

Distributed Real-Time Embedded Systems (DRE)

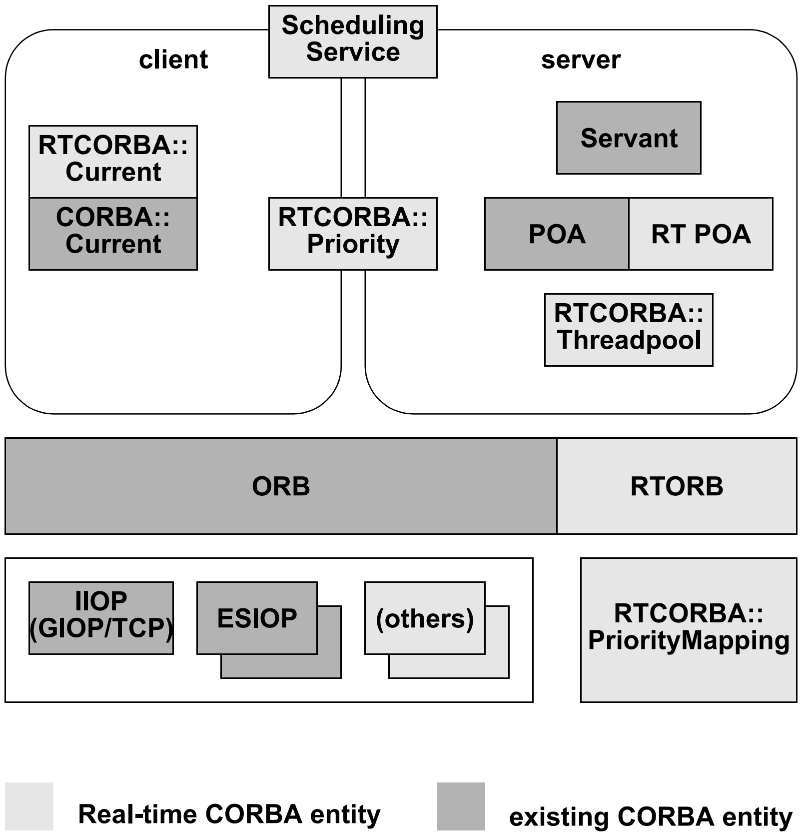
- Real-time computing is about predictability of timeliness
- Distributed real-time computing is about predictability of timeliness of multi-node (trans-node) behaviors
- Embedded systems must often deal with limited resources
- Non-functionional properties of distributed real-time systems not covered in this lecture:
  - Fault-tolerance, reliability, availability
  - Security, Quality of Service (QoS)
- Examples of DRE systems: telecommunication networks, tele-medicine, process automation, military appliciations

Real-Time CORBA Overview and Design Goals

- History: Version 1.0 Sept. 2000 - Version 2.0 Nov. 2003
- Extensions to OMG CORBA specifications
- Support of end-to-end predictability
- Definition of “Schedulable Entity” (threads) and priority control
- Avoid or bound priority inversions
- Bounding of method invocation blocking
- Extended resource management (process, storage, communication)
- Management of resource allocations (Mutex)
- Explicit set-up and configuration of bindings (connections)
- Configuration via CORBA:Policy mechnism

Real-time ORB & Real-time POA

- Real-time CORBA defines extentions to CORBA::ORB interface: RTCORBA:RTORB
- Getting RTORB: call ORB::resolve_initial_reference with ObjectId “RTORB”
- Extentions to POA defined in RTPortableServer::POA
- ORB::resolve_initial_references(“RootPOA”) returns RTPortableServer::POA



RT-CORBA Priorities & Priority Mappings

- RT-CORBA priorities are unique values ranging from 0 to 32767 (short)
- Priorities are set via RTCORBA::Current interface - resolve_i_r(“RTCurrent”)
- Mapping of CORBA priorities to native operating systems host priorities
- Upon setting the CORBA priority attribute the value is mapped to a native priority and the native priority of the current thread immediately set to that value

```
//IDL
module RT_CORBA {
    // Locality Constrained interface
     interface PriorityMapping{
         boolean to_native (in Priority       corba_priority,
                            out NativePriority native_priority);
         boolean to_CORBA  (in NativePriority native_priority,
                            out Priority corba_priority);
    };
};
```

RT-CORBA Priority Mappings - Example

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

[D.Schmidt et.al “Using Real-time CORBA Effectively”]

- Installation via `void install_priority_mapping(in PriorityMapping pm)`
- Only one priority mapping active at a time
- Used by the ORB for priority manipulation -> no exceptions
- Mapping function implementation must be re-entrant

Client Priority Propagation

- Configured in PriorityModelPolicy (CLIENT_PROPAGATED)
- CORBA priority is propagated in a CORBA priority service context
- During request dispatch thread priorities are adjusted
- If server code changes priority all subsequent invokations use this priority

```
module IOP {
        const ServiceId         RTCorbaPriority = 10;
};
```

Server-Set Priority Model

- Configuration via SERVER_SET_PRIORITY in PriorityModelPolicy
- Server-side thread executed with configured priority

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

policies[0] = rtorb->create_priority_model_policy
(RTCORBA::SERVER_DECLARED, LOW_PRIORITY);
// Get the ORB's policy manager

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);
```

Priority code in IOR
Used be client-side ORB to exploit e.g. priority banded conections

Client-side code in ORB should be executed with server declared priority

[D.Schmidt et.al "Using Real-time CORBA Effectively"]

Priorities - RT-CORBA 2.0 Additions

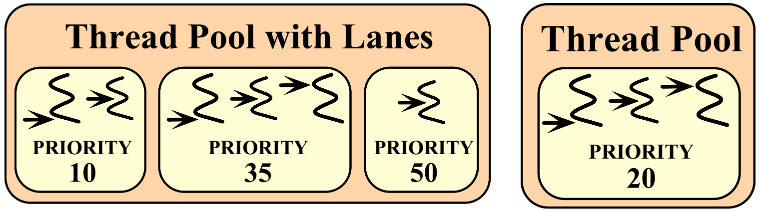
- Setting of server priority per object reference
- Overrides server declared priority

```
PortableServer::POA::ObjectId activate_object_with_priority (
        in PortableServer::Servant p_servant,
        in RTCORBA::Priority priority )

raises (PortableServer::POA::ServantAlreadyActive,
        PortableServer::POA::WrongPolicy );

void activate_object_with_id_and_priority (
        in PortableServer::ObjectId oid,
        in PortableServer::Servant p_servant,
        in RTCORBA::Priority priority )

raises (ServantAlreadyActive,
        ObjectAlreadyActive, WrongPolicy );
```

Priorities - RT-CORBA 2.0 Additions

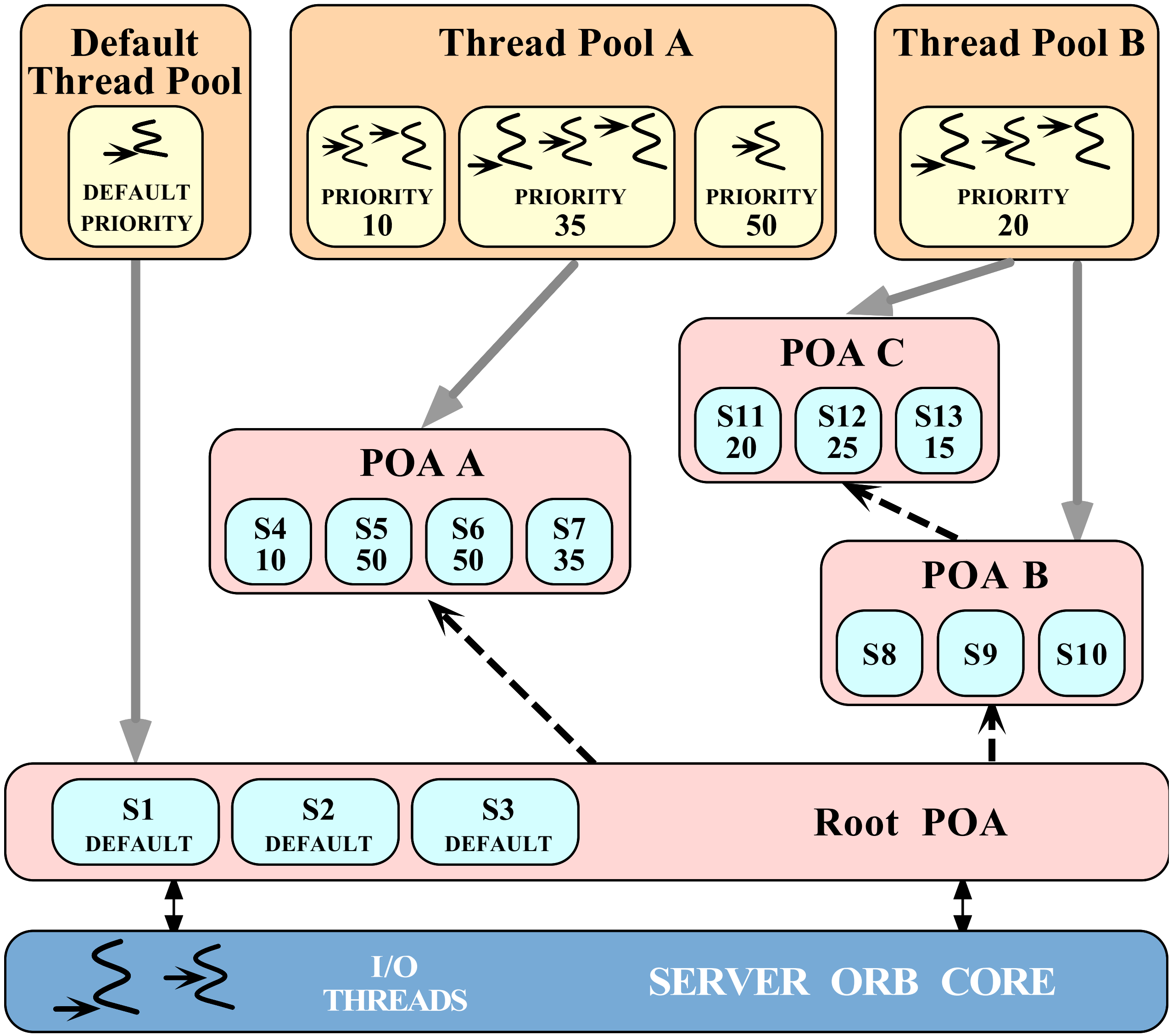
- Priority Transforms: implementation of user-defined invokation policies
  - Implementation of different priority models than server declared or client propagated
- Mapping of RTCORBA::Priority to other RTCORBA::Priority
- Can be installed:
  - During invocation upcall (after an invocation has been received at the server but before the servant code is invoked) - inbound Priority Transforms
  - When making an ‘onward’ CORBA invocation, from servant application code - outbound Priority Transforms

Threadpools & Threadpoollanes



- Lanes define different priority levels within a threadpool
  - Thread borrowing: high prio. lane may borrow threads from low prio lanes
- Preallocation of threads (static threads)
  - Reduction of priority inversion (low priority request don't block high prior ones)
  - Reduction of latency and increase of predictability by avoiding recreation and destruction of threads
- Partitioning of threads
  - isolation of system parts by association of POAs to different thread pools
- Bounding of thread usage (memory usage together with queues size)
  - Limitatation of threads a number of POAs may use
    (max. threads = static threads + dynamic threads)

Threadpools: POAs & ORB



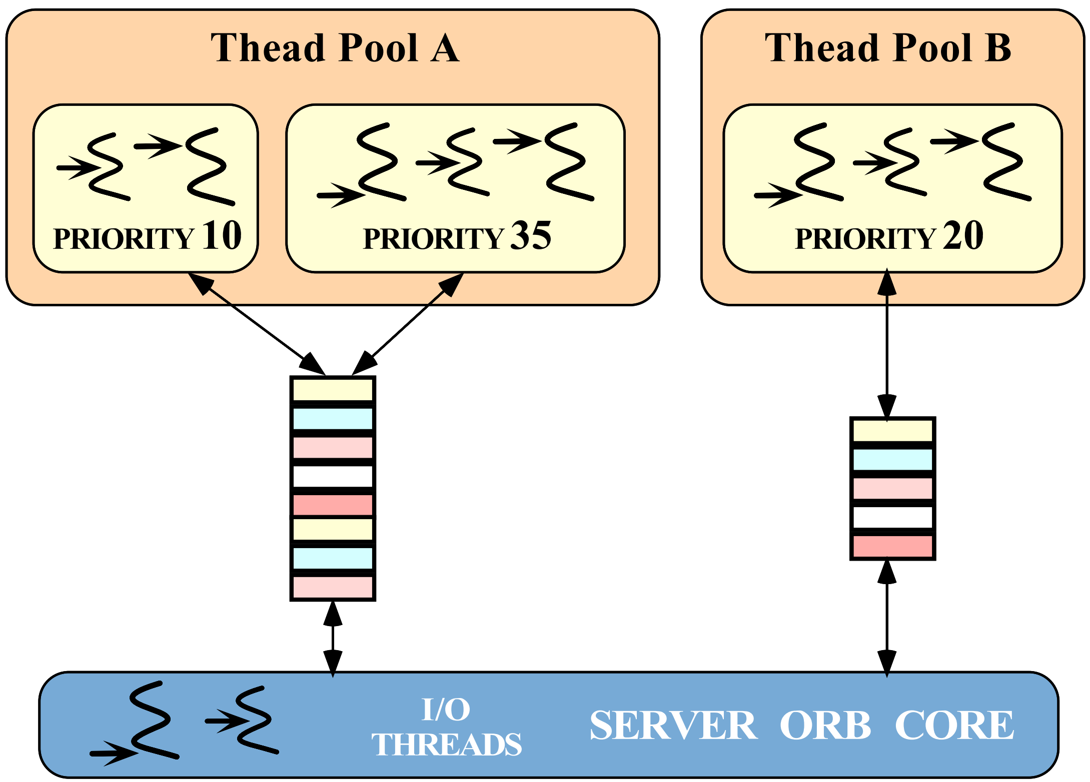
- Threadpools can be applied to POA and ORB level
- Max. one threadpool per POA

Creation and Destruction of Threadpools

```
//IDL
module RTCORBA {
// Threadpool types
typedef unsigned long ThreadpoolId;
      struct ThreadpoolLane {
      Priority lane_priority;
      unsigned long  static_threads;
      unsigned long  dynamic_threads;
};
```

```
typedef sequence <ThreadpoolLane> ThreadpoolLanes;
// Threadpool Policy
const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;
local interface ThreadpoolPolicy : CORBA::Policy     {
readonly attribute ThreadpoolId threadpool;
};

local interface RTORB {
...
ThreadpoolPolicy create_threadpool_policy (in ThreadpoolId threadpool);
exception InvalidThreadpool {};
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 );
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 );
      void destroy_threadpool ( in ThreadpoolId threadpool )
      raises (InvalidThreadpool);
};
```

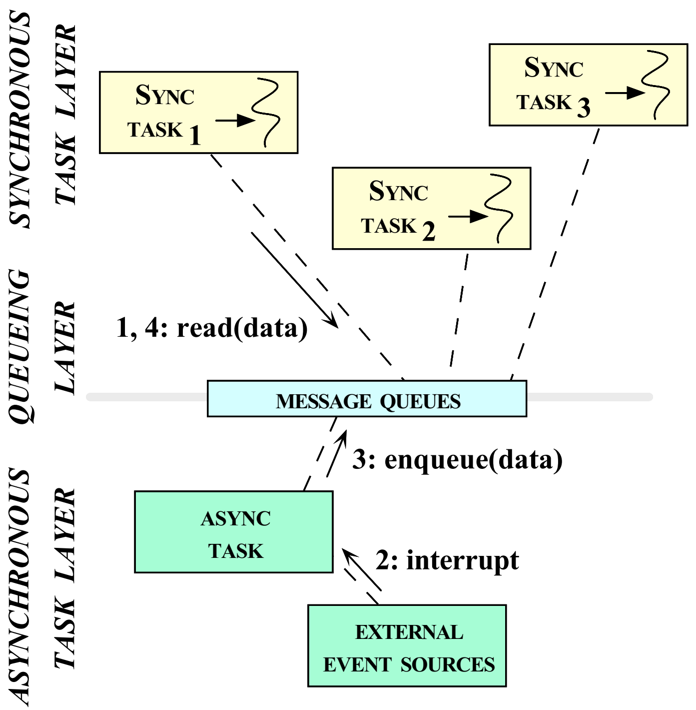
Request Buffering in RT-CORBA Threadpools

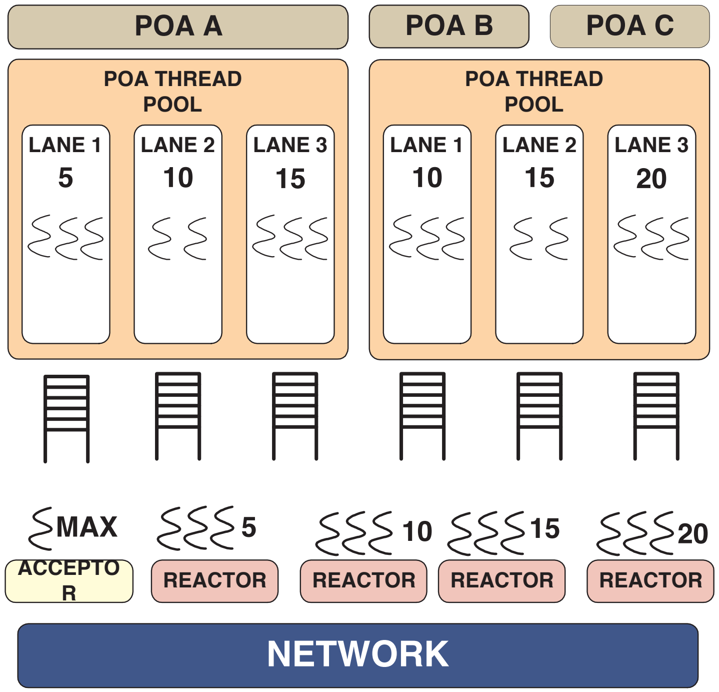


- Provides control over storage resources
- No separate thread for every request neccessary
- Used if no static or dynamic thread is available

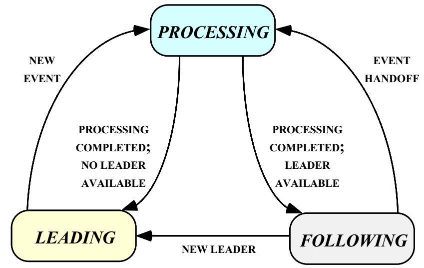
Implementing Threadpools
Half-Synch/Half-Asynch Pattern

- Buffering of requests in a queue by I/O-threads
- Worker threads within the pool process requests from queue
- Easy implementation of thread borrowing, but less efficient because of queuing



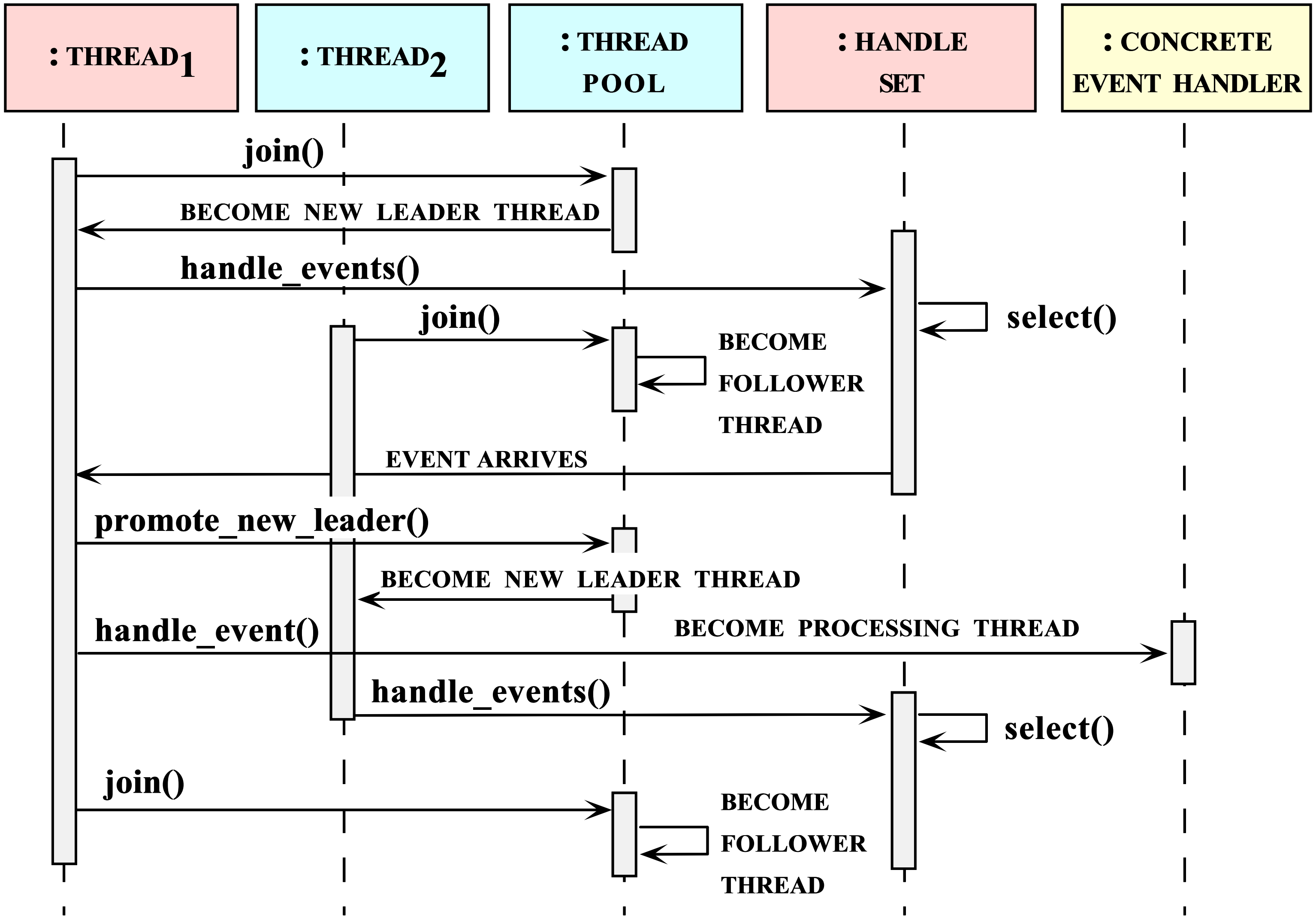


Implementing Threadpools Leader/Followers Pattern



- A number of threads (in a threadpool) is synchronized to get process external requests
- At one time one thread - the leader - waits for an event on a set of I/O-handles
- Other threads - the followers - can queue up and wait to become leader
- Current leader determines follower, after demultiplexing an event from I/O-handles
- Underlying I/O-system queues events if no threads are available
- No additional thread for request dispatch + better performance
- Request buffering & borrowing harder to implement (no explicit queue)

Leader/Followers Pattern - Example Sequence



Real-Time CORBA Mutex

- Standardized mutex implementation for all applications
- Two states: locked and unlocked
- Born in unlocked State
- Implementation of priority inheritance required
- ORB must use same mutex implementation as delivered to applications
    - Consistent priority inversion avoidance

```
//IDL
module RT_CORBA {
  // locality constrained interface
   interface Mutex    {
      void lock();
      void unlock();
      boolean try_lock(in TimeBase::TimeT max_wait);
         // if max_wait = 0 then return immediately
   };
  interface ORB : CORBA::ORB    {
   ...
     Mutex create_mutex();
   ...
  };
};
```

Server-Side Configuration - ProtocolPolicy

- Configuration and selection of communication protocols
- Definition of multiple protocols and order configuration possible
- Protocol defined as pair of ORB protocol (GIOP) and transport protocol (TCP)
- ProtocolProperties for protocol specific configuration (message length, buffer size)

```
/ IDL module RT_CORBA {
    // Locality Constrained interface
    interface ProtocolProperties {};
    struct Protocol {
          IOP::ProfileId      protocol_type;
          ProtocolProperties orb_protocol_properties;
          ProtocolProperties transport_protocol_properties;
      };
    typedef sequence <Protocol> ProtocolList;
    // Protocol Policy
     const CORBA::PolicyType PROTOCOL_POLICY_TYPE = ??;
     // Locality Constrained interface
     interface ProtocolPolicy : CORBA::Policy     {
          readonly attribute ProtocolList protocols;
      };
  };
```

ProtocolPolicy Example

[D.Schmidt et.al "Using Real-time CORBA Effectively"]

- Creation of 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 */);
```

- Configuration of protocol list

```
RTCORBA::ProtocolList plist; plist.length (2);
plist[0].protocol_type = MY_PROTOCOL_TAG;  // Custom protocol
plist[0].trans_protocol_props =
/* Use ORB proprietary interface */
plist[1].protocol_type = IOP::TAG_INTERNET_IOP; // IIOP
plist[1].trans_protocol_props = tcp_properties;
RTCORBA::ClientProtocolPolicy_ptr policy =
rtorb->create_client_protocol_policy (plist);
```

Client-side configuration - Banded Connections

- Configured via PriorityBandedConnectionsPolicy
- Reduction of priority inversion caused by using non-priority transport protocols
- Facility for clients to communicate with a server via multiple connections
  - Each connections handles separate invokation priority level (range)
  - Connection selection transparent to the application
  - Applied at client-side during object binding or server-side and propagated via IOR

```
//IDL
module RT_CORBA {
     struct PriorityBand     {
          Priority low;
          Priority high;
    }
    typedef sequence <PriorityBand> PriorityBands;
    // PriorityBandedConnectionPolicy
    const CORBA::PolicyType
                     PRIORITY_BANDED_CONNECTIONS_POLICY_TYPE = 45;
    interface PriorityBandedConnectionPolicy : CORBA::Policy     {
          readonly attribute PriorityBands priority_bands;
     };
};
```

Priority Bands - Example

```
// Create the priority bands
RTCORBA::PriorityBands bands (2); bands.length (2);
bands[0].low = LOW_PRIO;       // We can have bands with
bands[0].high = MEDIUM_PRIO;   // a range of priorities or
bands[1].low = HIGH_PRIO;      // just a "range" of 1!
bands[1].high = HIGH_PRIO;
// Now create the policy...
CORBA::PolicyList policies (1); policies.length (1);
policies[0] =
rtorb->create_priority_banded_connection_policy (bands);
// Use just like any other policies...
```

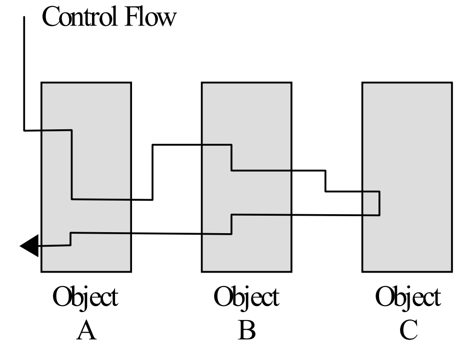
- Priority Bands can also be used on client-side to pre-allocate connections
- If priority bands are installed and an invokation with a priority triggered without a configured (range): a “no resource” system exception is thrown

More Connection Policies

- Client-Side Configuration - Private Connections
    - Configured via PrivateConnectionPolicy
    - Private for connection for one object binding
    - not multiplexed with other invokations
- Invokation Timeouts
    - Configured via RelativeRoundtripTimeoutPolicy
    - Allows for definition of timeout for invokations
    - Server is not informed about expiration of a timeout
    - Defined in original CORBA specification

RT-CORBA v2.0 Dynamic Scheduling

- Static priority scheduling not sufficient for dynamic workloads
- Integration of other (dynamic) scheduling algorithms (EDF,LSF,LLF,...)
  - Plugin schedulers

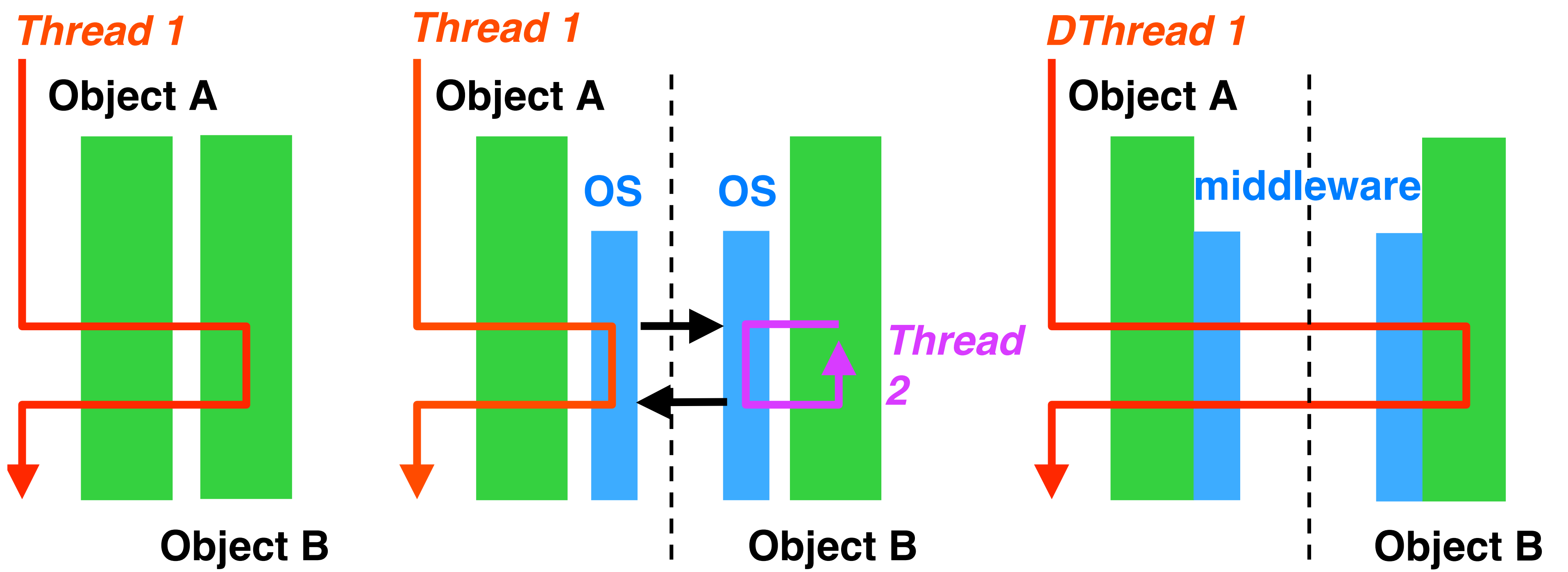


- Distributable Thread (DT) replaces activity definition
  - Each DT has system-wide unique identifier
  - DT has one or more execution scheduling parameter elements (priority, time constraints (deadlines, utility functions, importance)
  - Semantics of acceptability of end-to-end timelininess defined by the application in context of used scheduling discipline
  - Execution of DTs governed by scheduling parameter elements at each visited node

Distributed System Scheduling

- Scheduling in distributed system can be devided into 4 classes
  - Scheduling independently on each node and there is no trans-node end-to-end timeliness requirement (non-realtime systems)
  - Scheduling independently on each node but there is a mechanism such as priority propagation (RT-CORBA specification 1.*)
  - Scheduling on each node is global: there is a logical singular system-wide scheduling algorithm instatiated on each node (implementable in RT-CORBA 2.0)
  - Multi-level scheduling: at least one level of meta-scheduling - global optimization by adaptive adjustment of local policies

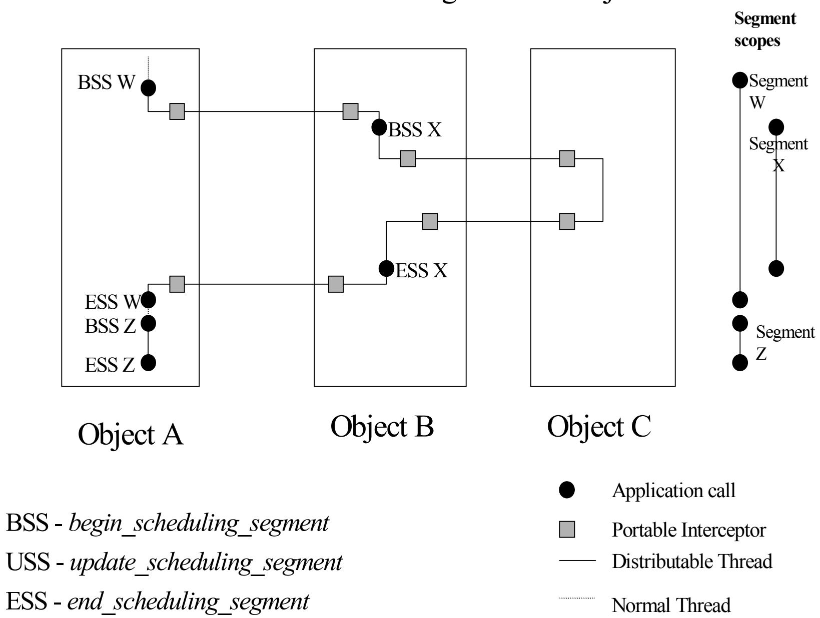
Distributable Thread Abstraction



Distributable Threads - Scheduling Segments

- Distributable threads consist of one or more (potentially nested) scheduling segments (nesting creates scheduling scopes)
- Each segment represents a sequence of control flow with associated scheduling parameter elements
- Declaration of segments within code through: begin_scheduling_segment and end_scheduling_segment
- Update of scheduling parameters within segment using update_scheduling_segment
- Segments may span processor bounddaries

Distributable Thread Traversing CORBA Objects



Dynamic Scheduling Interfaces

- DT entry points defined by overriding ThreadAction::do method
- DT creation: RTCORBA::Current::spawn
- segment specific functions (begin,end,update)
- Distributable thread id specific functions
  - IdType get_current_id();
  - DistributableThread lookup(in IdType id);
- DT cancelation (RTCORBA::Current::cancel(id)
- Readonly access to scheduling parameters
- Getting current segment names (list)

```
module RTScheduling {
    ...
    local interface Current : RTCORBA::Current {
        ...
        DistributableThread spawn (
            in ThreadAction      start,
            in unsigned long     stack_size,
             // zero means use the O/S default
             in RTCORBA::Priority base_priority);
        ...
    };
    ...
};
```

(Distributed) Real-Time Specification for Java

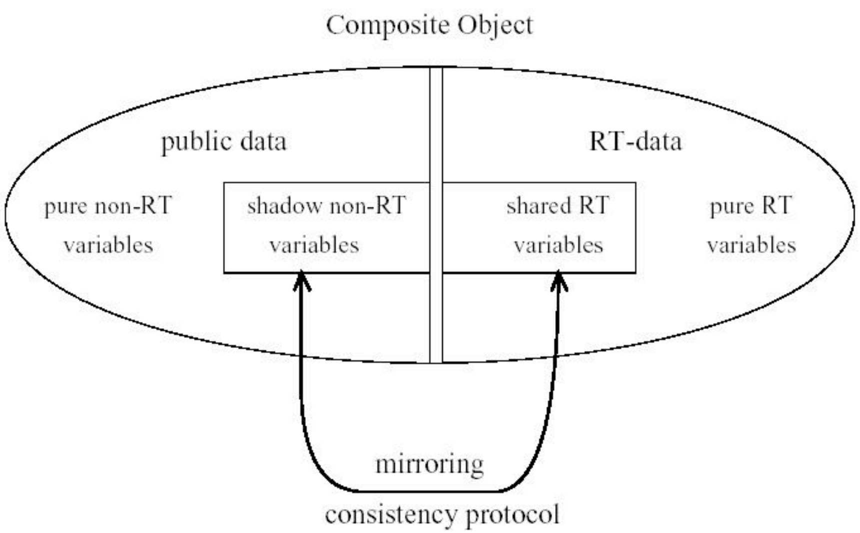
- Extended thread & synchronization model
  - RealtimeThread and NoHeapRealtimeThread
  - Static priority scheduler with > 28 priorities
- Support for user-defined schedulers
- Extended Memory Model - GC-free memory regions
  - Scoped Memory
  - Immortal Memory
- Asynchronous Transfer of Control
- Direct memory access and interrupt handling

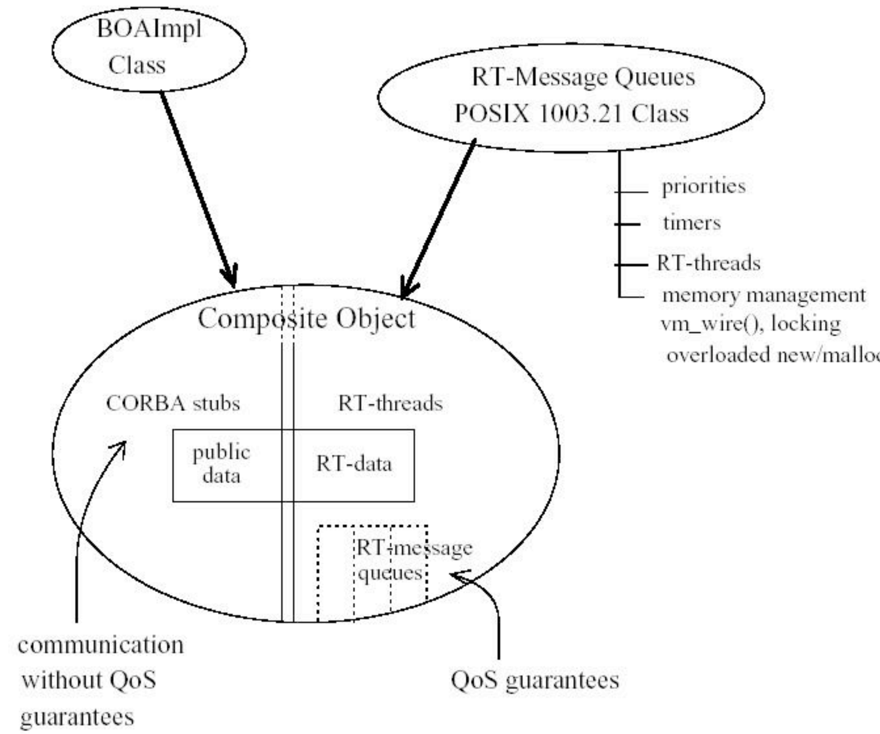
Distributed Real-Time Specification for Java (JSR-50)

- Extention of RTSJ in a natural and familiar way
- Real-time RMI (Modification of JSR-78 RMI - Custom Remote Interfaces)
  - Support for propagating resource management specific data
  - Configuration of underlying transport infratructure
- Lexically scoped timing constraints (BeginTimeContraint{}, BeginTimeContraint{})
- Distributable Thread Integrity Framework
  - Integration of application-specific policies for maintaining the health and integrity of Distributable Threads in presence of failures
- Scheduling Framework
  - Plug-in architecture for integration of appropriate user space policies for

Composite Objects - Real-Time with CORBA [Polze98]

- Integration of real-time into non-realtime CORBA
- Decoupling of real-time and non-real-time part via shared buffer and consistency protocol (weak consistency for shared variables)

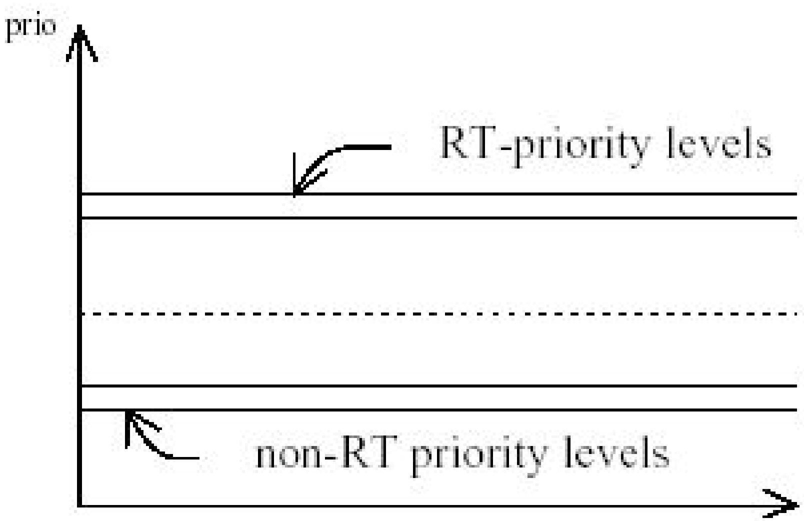




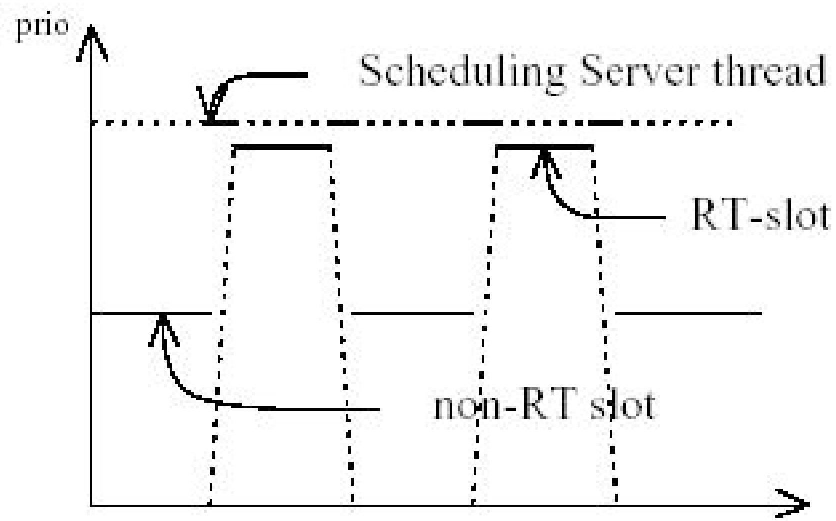
Problems: Scheduling of RT-threads
Firewall/mirroring between RT & public data

Composite Objects - Timing Firewalls

- Non-real-time parts must not violate real-time scheduling rules
- Usage of scheduling server approach for CPU partitioning

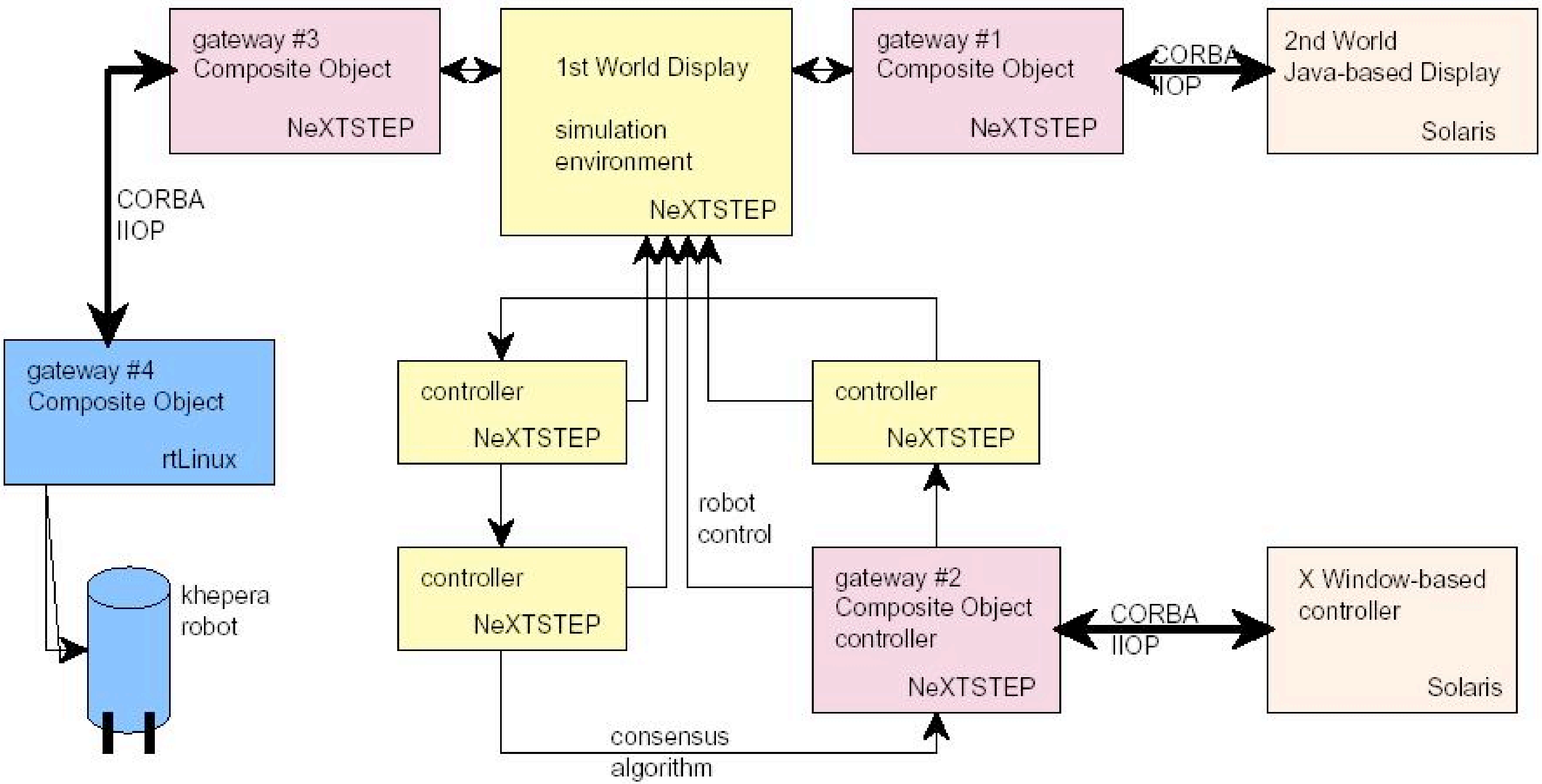


horizontal firewall



vertical firewall

Composite Objects in Action - Unstoppable Robots



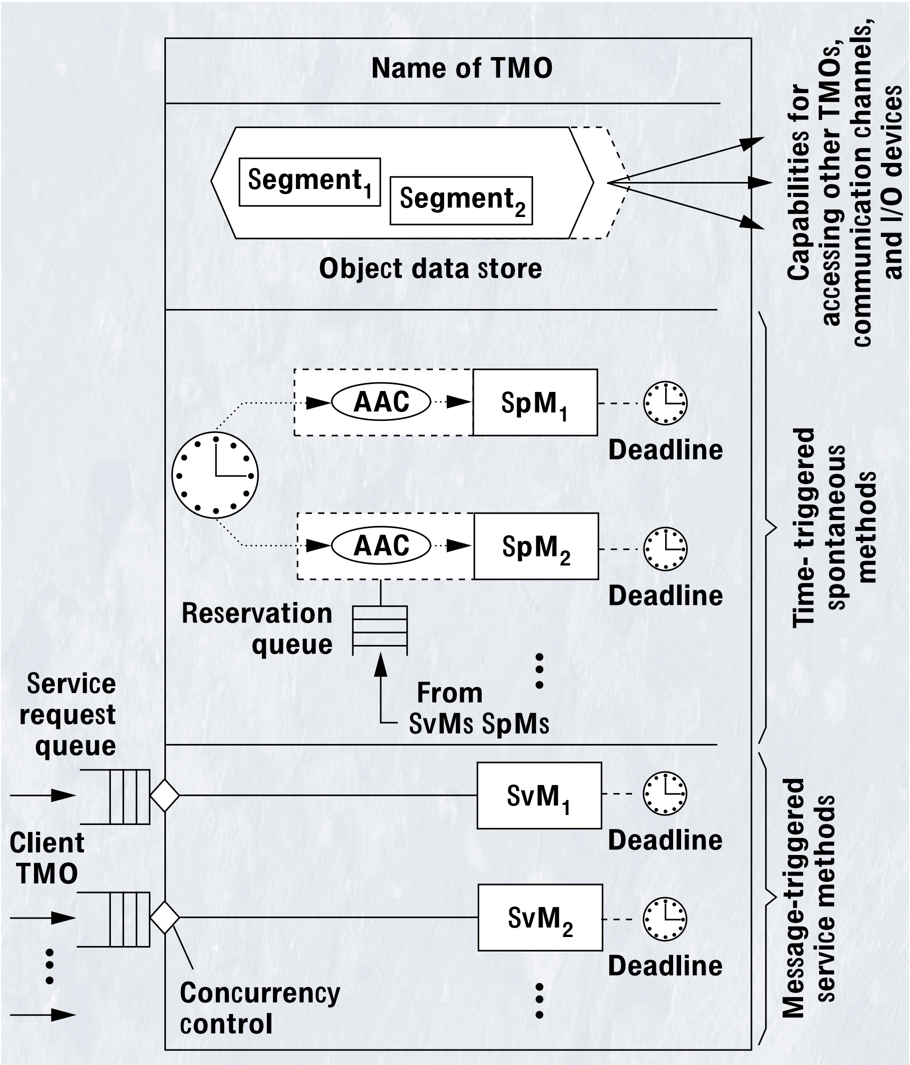
Time-Triggered Message-Triggered Object (TMO)

- Early ’90s by Kane Kim at Dreamlabs University of California Irvine
- Component structering scheme supporting real-time and non-real-time objects
- A TMOs are distributed computing components interacting via remote method calls
- TMOs can contain two types of methods
  - Time-triggered methods (also called spontaneous methods or SpMs)
  - Conventional service methods (SvMs)
- Basic concurrency constraint: activation of an SvM triggered by a message from an external client is allowed only when conflicting SpM executions are not in place
- Triggering times for SpMs must be specified as constants during design time

```
"for t = from 10am to 10:50am
    every  30min
    start-during (t, t+5min)
    finish-by t+10min"
```

TMO structure

- Object data store: lockable segments containing data members
- Service methods: triggerd by messages to provide services requested by client objects (TMO designer guarantees deadlines for output production)
- SpMs are invoked when the real-time clock reaches the specified time
- Candidate times: set of times actual triggering time will be choosen from
- TMO designer guarantees timely service to all potential clients by indicating the deadline for every output produced in response to a service method request



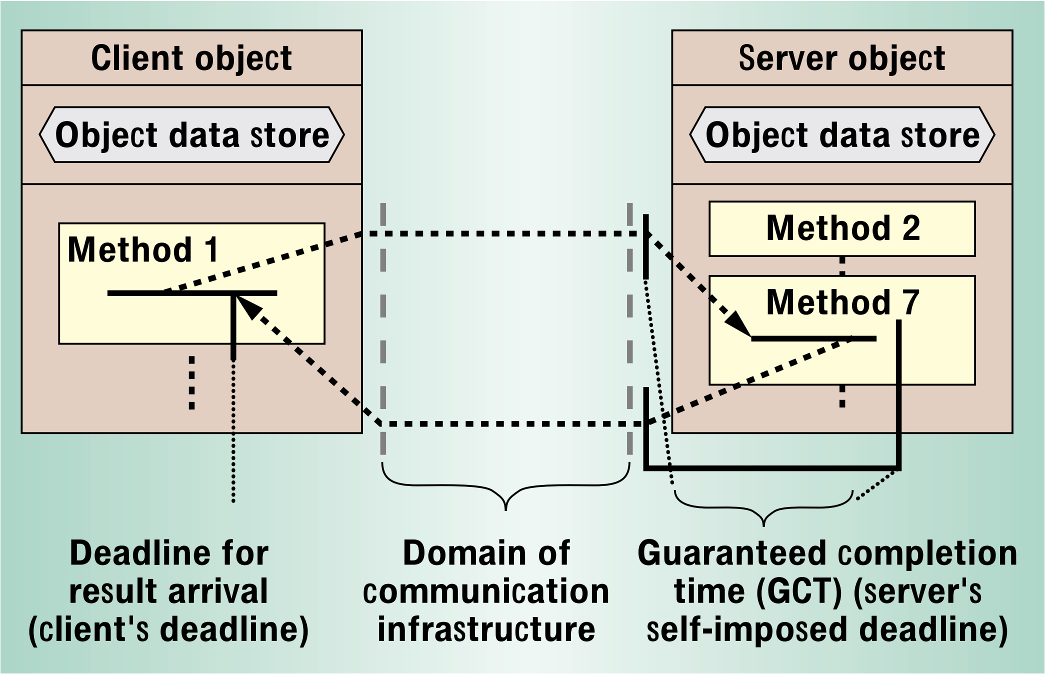
TMO - Guaranteed Deadlines

- Client's deadline for result arrival is set by the programmer with knowledge of the server's GCT and the transmission times consumed by the communication infrastructure

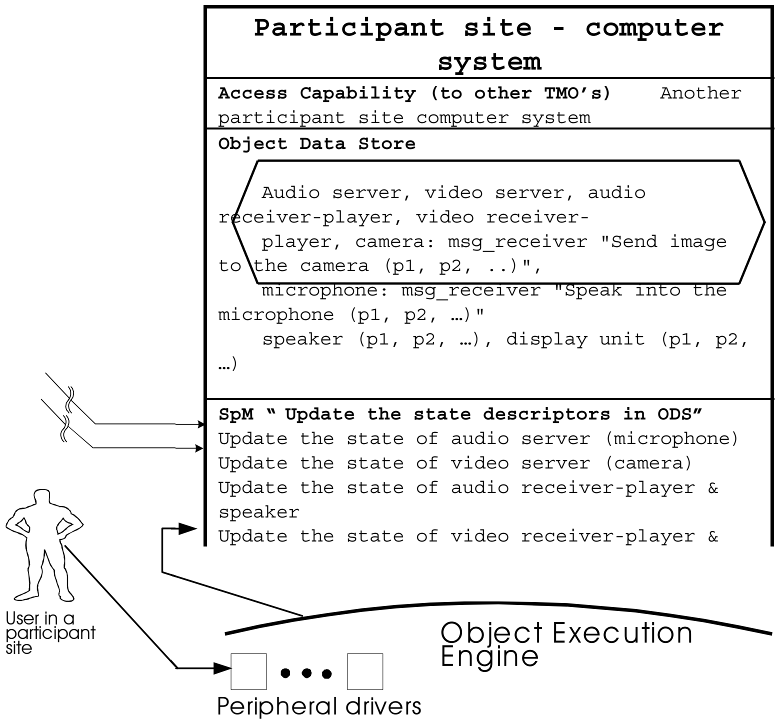
- Client's execution engine ensures that client's deadline is kept under a GCT advertised by a server

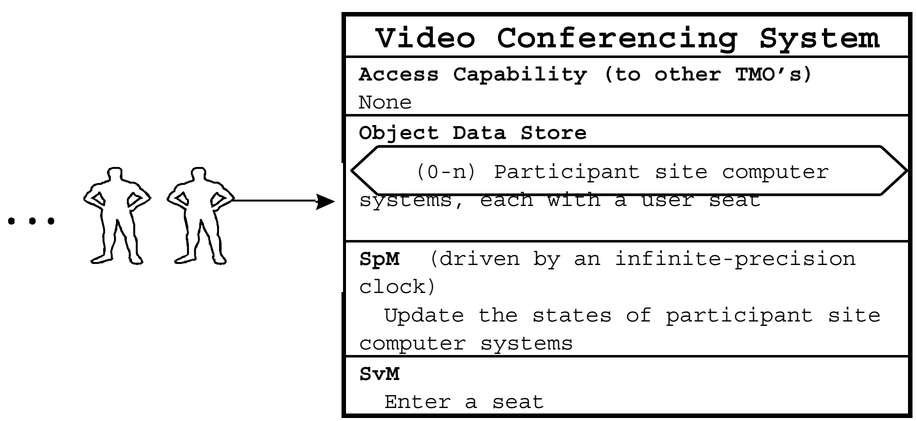
- Maximum invokation rates (MIR) are specified during SvM creation

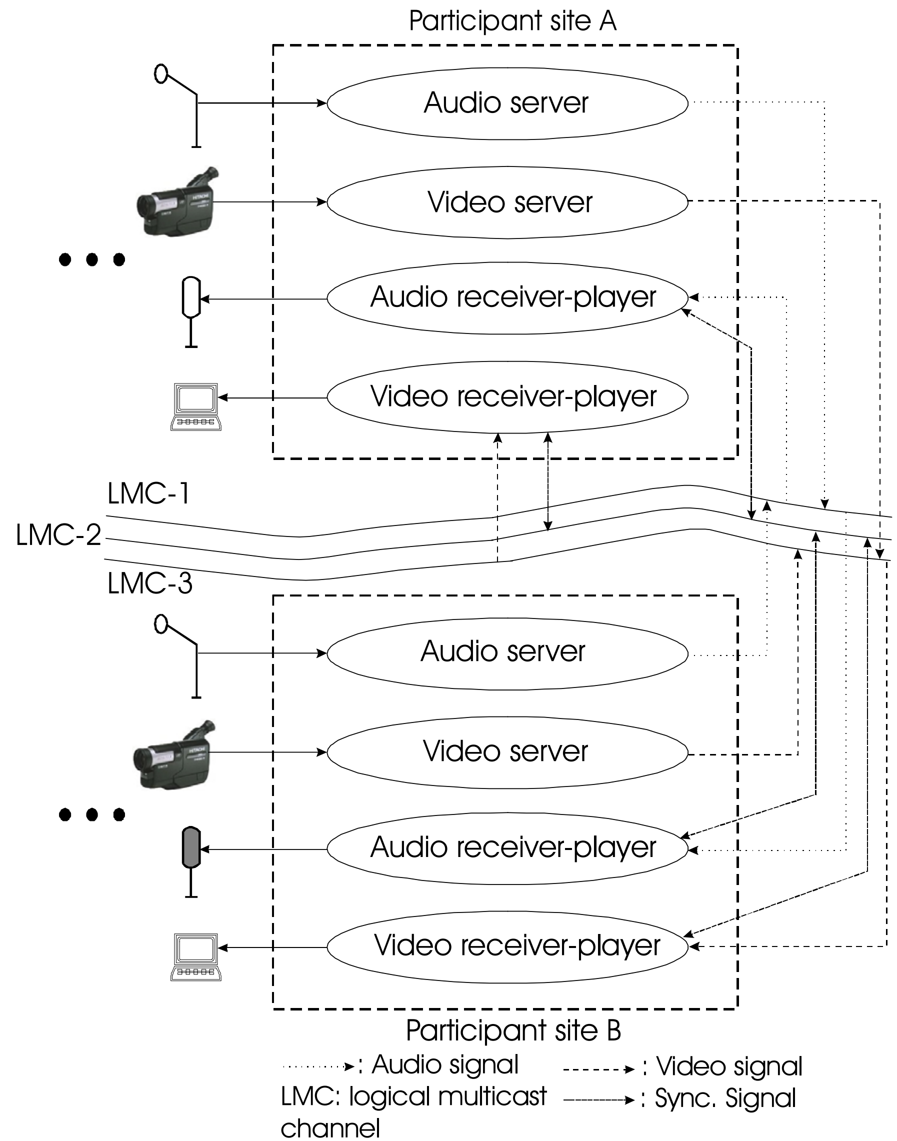
- If a client can't hold its deadline it can trigger an alternative action or choose another TMO with better timings (comm. infrastructure, GCT, MIR (load situation)



TMO-based Video Conferencing System

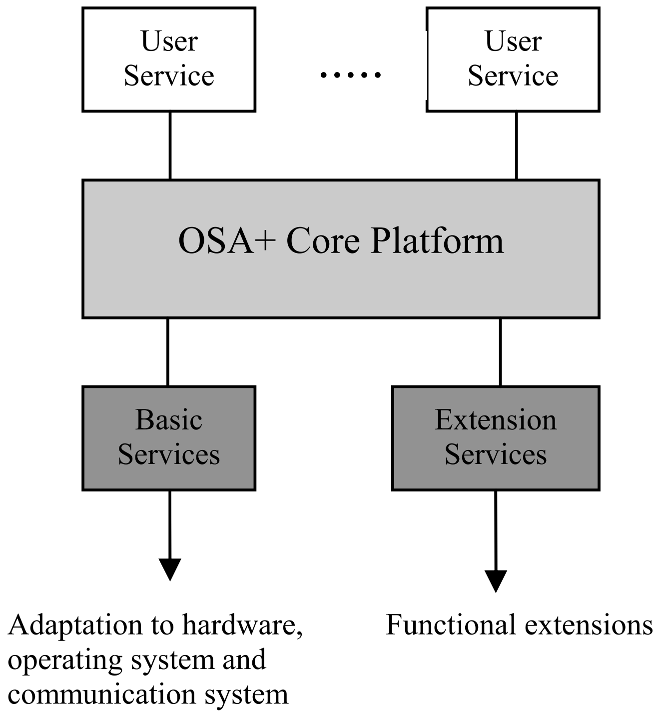






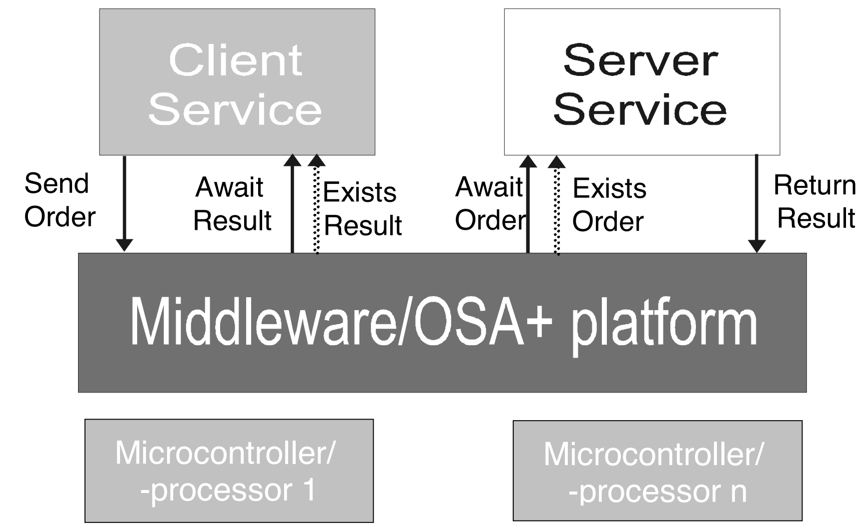
Open Systems Architecture - OSA+

- Developed at University of Karlsruhe (Prof. Brinkschulte)
- Real-time middleware using microkernel concepts targeting small low power devices
- Active entities in OSA+ are services - they communicate via jobs
    - A job consist of order and result
- Services can be plugged into a platform
- Multiple platforms in a distributed environment form a virtual platform hiding heterogenous infrastructure of underlying systems



OSA+ Jobs

- Jobs are used for:
  - Communication - by exchaning order and result
  - Synchronisation - by creating a specific order of orders
  - Parallel execution - by parallel creation or orders
  - Real-time execution - using time contraints within orders



OSA+ Base Services

- Task Service - Connection between micro kernel and underlying operting system. Implements scheduling, synchronization, parallel execution
- Memory Service - Connection between micro kernel and memory management of underlying operting system. Implements dynamic allocation and management of memory
- Event Service - Time-triggered execution of jobs and copling of job delivery to internal and external events
- Communication Service - Connection to communication sub-system. Delivery of jobs to distributed services
- Addressing Service - Localization of services. Clients can query locations of distributed services
- Reconfiguration Service - Dynamic Rekonfiguration of services during runtime

Further Reading

- J. Lui. “Real-Time Systems”, Prentice Hall
- RealTime-CORBA Specification 2.0, OMG, November 2003
- Carlos O. Rain, D. Schmidt, “Using Real-time CORBA effectivly”, www.cs.wustl.edu/~schmidt/tutorials-corba.html/
- J. Anderson, D. Jensen, “Distributed Real-time Specification for Java - A Status Report”
- K.H. (Kane) Kim, “Object Structures for Real-time Systems and Simulators”, IEEE Computer 1997
- F. Picioroaga et. al. “OSA+ Real-Time Middleware, Results and Perspectives”, ISORC ‘04