A Guided Tour of Real-time CORBA
Part 3b

Chris Gill
Assistant Professor
Computer Science & Engineering Dept.
Washington University, St. Louis MO
cdgill@cse.wustl.edu
www.cse.wustl.edu/~cdgill

Irfan Pyarali
President
OOMWorks, Metuchen NJ
irfan@oomworks.com
www.oomworks.com

A Guided Tour of Real-time CORBA

Part 3a
Using RT-CORBA in Dynamically Scheduled Systems
Practical implementation and Code examples

Implementing Real-Time CORBA 1.2

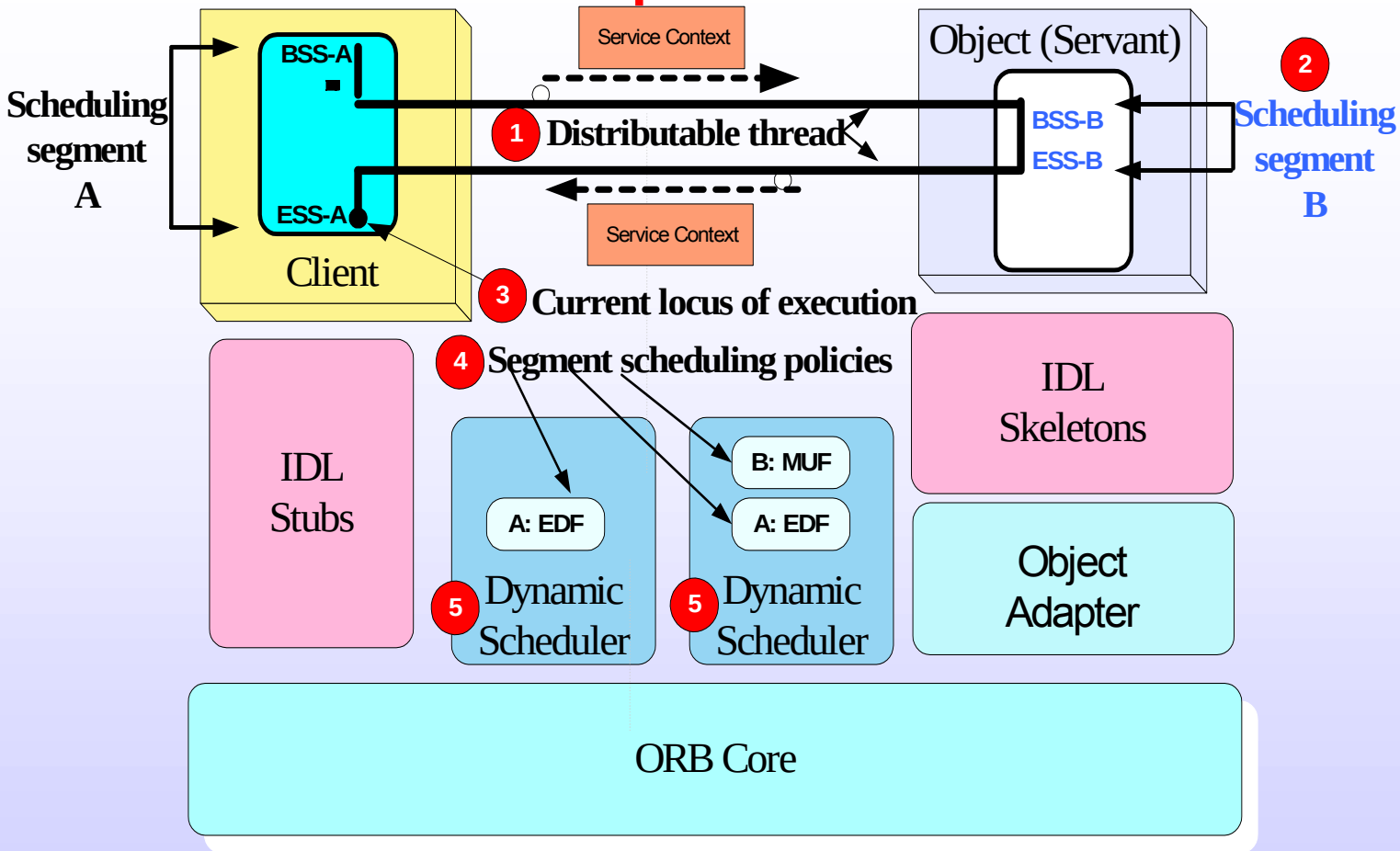
Scheduling

- ***Heterogeneous Scheduling*** is when different policies can be applied
- ***Scheduling Segment*** is a sequence of code with specific scheduling requirements
- ***Pluggable Scheduler*** facilitates the use of custom scheduling disciplines
- ***Scheduling Points*** to interact with the scheduler at application and ORB level

Distributable Threads

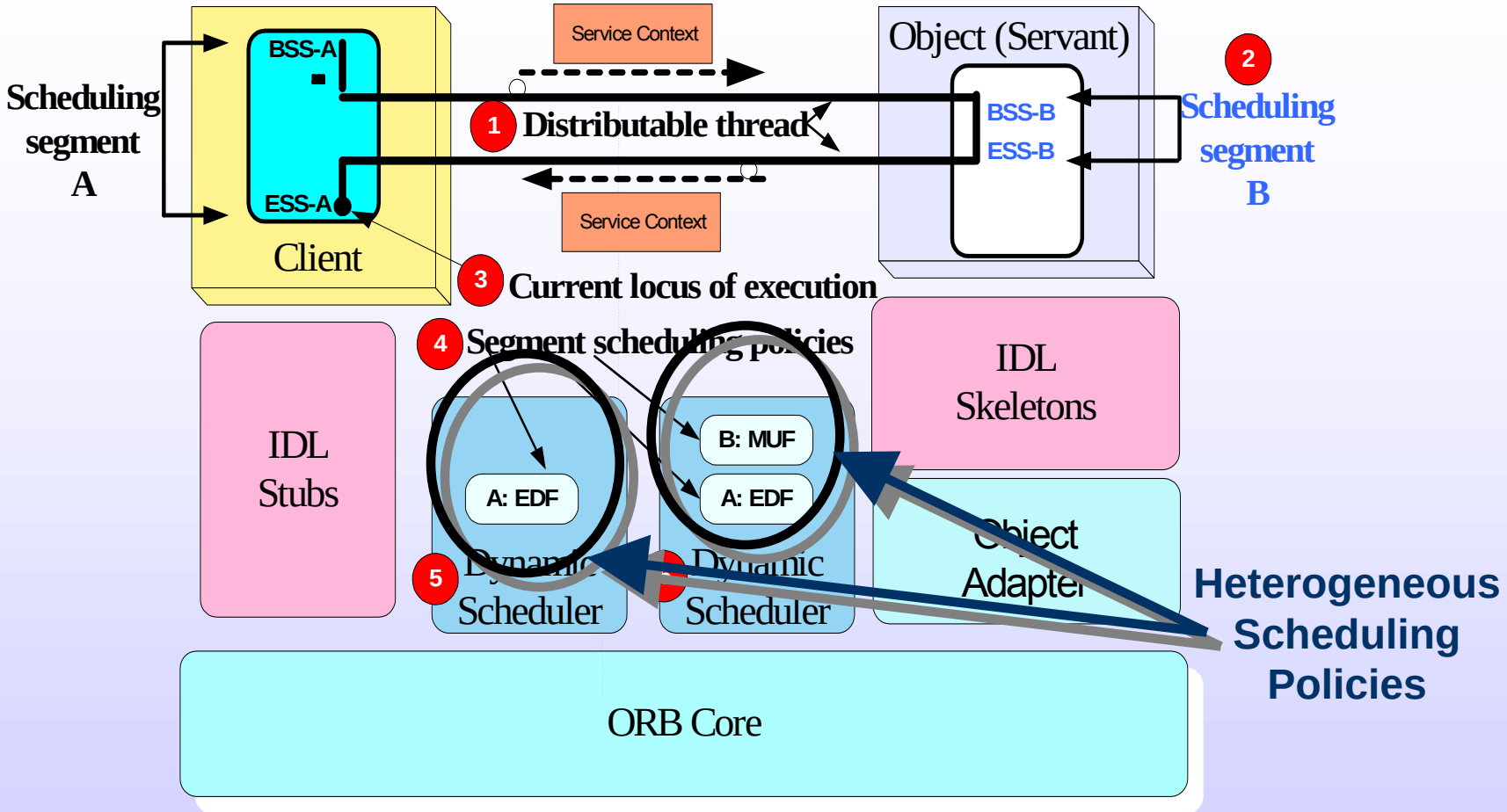
- ***Distributable Thread (DT)*** a programming model abstraction for a distributed task
  - Loci of execution span multiple nodes and scheduling segments
  - Identified by a unique system wide id GUID (Globally Unique Id)
  - DT scheduling information passed through GIOP Service Contexts
    - Across the hosts it spans

RT CORBA 1.2 Implementation Concepts



- Distributable thread – distributed concurrency abstraction
- Scheduling segment – governed by a single scheduling policy
- Locus of execution – point at which distributable thread is currently running
- Scheduling policies – determine eligibility of different distributable threads
- Dynamic schedulers – enforce distributable thread eligibility constraints

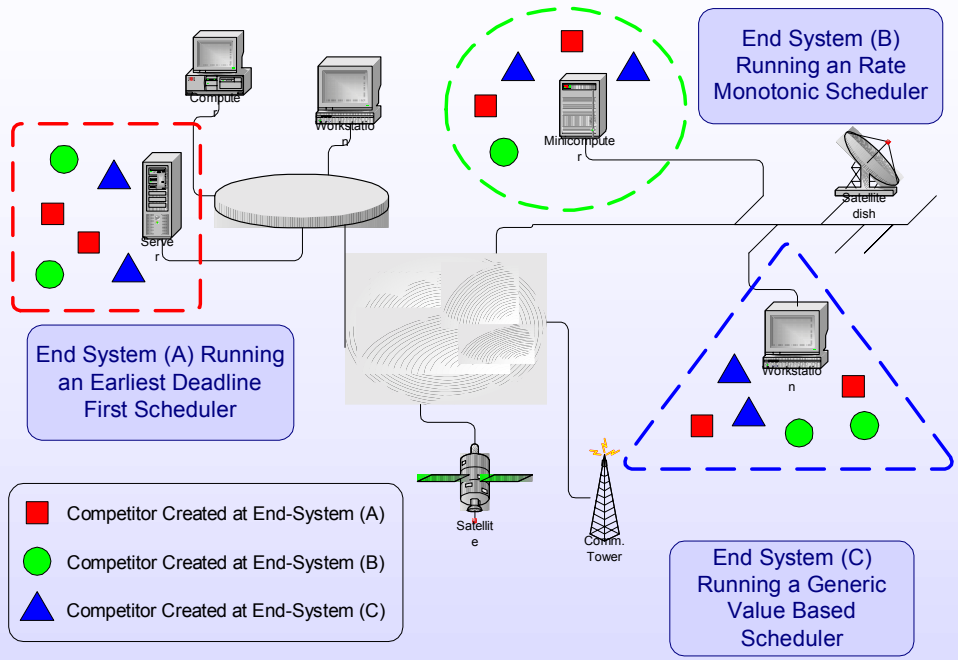
A Brief Digression and Several Caveats



- RT-CORBA 1.2 lets you install heterogeneous schedulers
  - Can be powerful in the hands of real-time experts
  - Inordinately complex and difficult otherwise
    - Even for expert systems programmers new to real-time

Early Heterogeneous Scheduling Research

- Angelo Corsaro's MS Thesis
  www.cs.wustl.edu/~corsaro/
- Examines ways to reconcile heterogeneous policies
- Defines 3 main cases
  - Restriction
  - Extension
  - General
- Provides a meta-programming architecture
  - Using specific kinds of adapters for each case



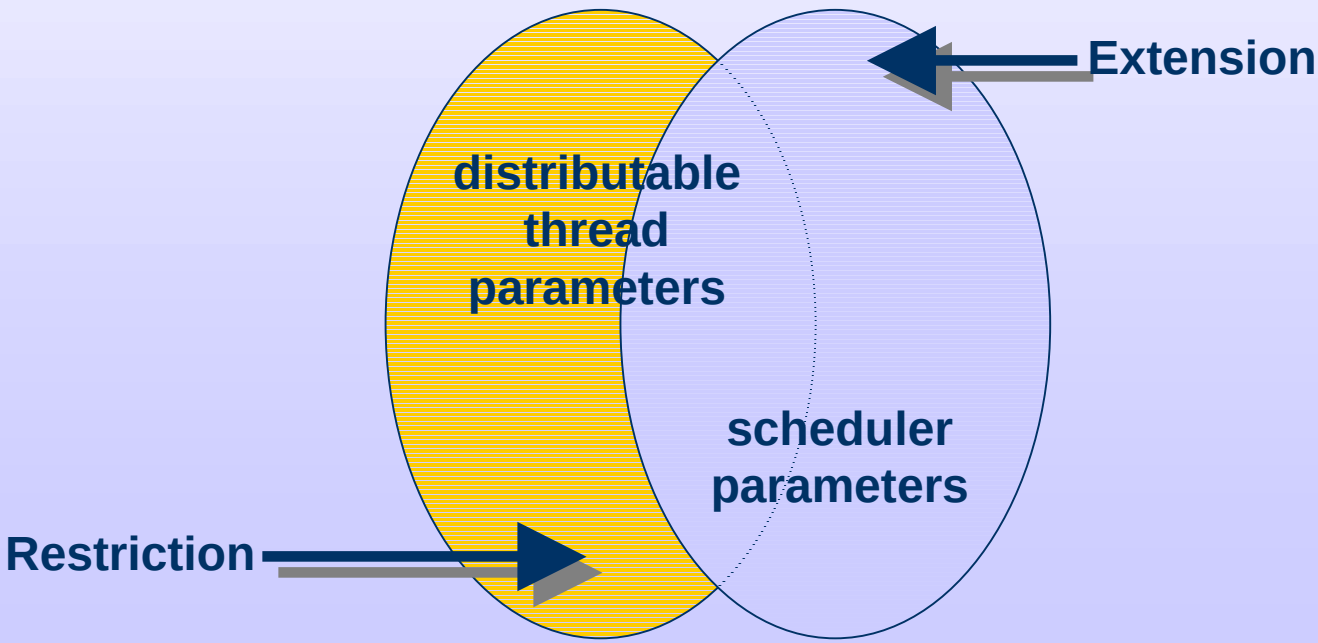
(from Corsaro, *et al.*, "Applying Meta-Programming Techniques to Reconcile Heterogeneous Scheduling Policies in Open Distributed Real-Time Systems", DOA 2001, Rome, Italy)

Restriction & Extension Adapters (Corsaro)

- Restriction Adapters
  - Used when properties provided by a distributable thread are a *superset* of those used by the scheduler (easiest case)
  - Adapter narrows the set of properties to scheduler's needs
- Extension Adapters
  - Used when distributable thread's properties are a *subset* of those used by the scheduler (a slightly harder case)
  - Extension adapter provides default values for any missing properties
  - Extension adaptation can be static (*a priori*) or dynamic (*contextual*)

General Property Adapters (Corsaro)

- Arbitrary mappings are possible
  - from parameters provided by a distributable thread and ones a scheduler uses
- Adapters may perform arbitrary computations in general
- Can be viewed as a composition of restriction & extension
  - May add constraints on parameter set interfaces/operators
  - *E.g.*, eligibility inheritance requires X(X&), =, <= on resulting set (Huang)



Other Things to Keep in Mind

- Real-time systems have a “weakest-link-in-the-chain” property
  - Insufficiently bounded timing anywhere in the system can cause real-time failure of the entire system
- Some examples are fairly obvious
  - **`while(1)`** considered harmful
- Other examples are less obvious
  - When is **`while(k <= n)`** OK?
  - Dynamic binding: object substitution and virtual methods
- Middleware-based real-time systems are further nuanced
  - Direct manipulation of OS threads, priorities, mutexes, etc.
  - If the middleware assumes it has control, these “end-runs” can again lead to real-time failure
- Important tension between control by application programmer and the need to manage complexity through encapsulation

Implementation Challenges I: Dynamic Scheduling

- Tasks span multiple heterogeneous hosts
- Tasks and hosts they span are not known a priori
- Propagate scheduling requirements of a task across hosts it spans
- Allow custom scheduling discipline plug-ins
- Fixed priorities for tasks insufficient for scheduling analysis
- Cancel a distributed task and reschedule accordingly on hosts it spans
- Local schedulers need to reschedule when tasks return from a remote invocation
- Collaboration of schedulers in the system to ensure global scheduling

Motivation

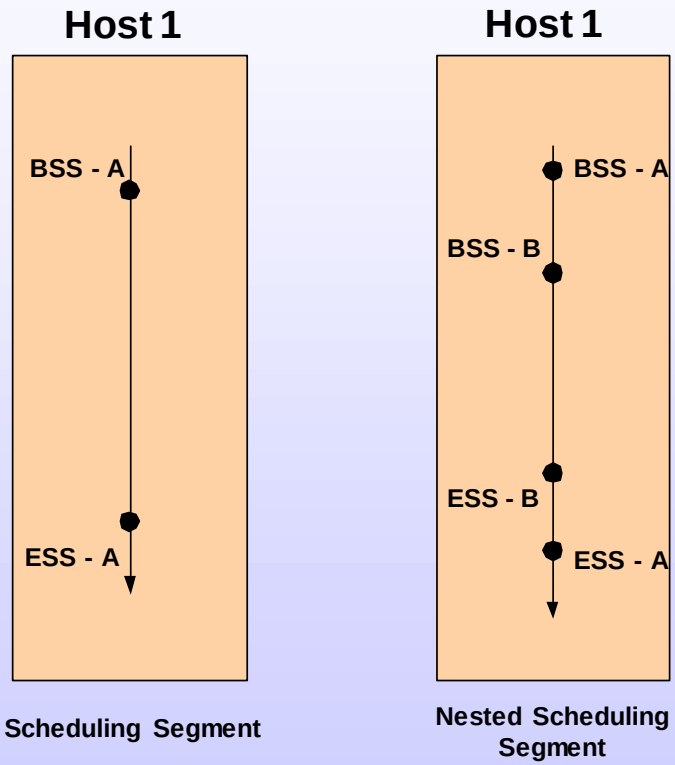
- Dynamic scheduling allows late binding of ordering decisions among competing tasks
  - Allows greater customization w/ behavioral variation
- Especially useful if run-time factors influence decisions
- Many dynamic scheduling approaches are well-known
  - Time-utility functions (utility accrual)
  - Earliest deadline first
  - Minimum laxity first (deadline – remaining work)
  - Maximum urgency first (adds criticality bias)

Implementation Differences for Static and Dynamic Scheduling

Static Scheduling	Dynamic Scheduling
Priorities can be mapped to OS thread priorities (per mode)	If thread priorities are used, they may need to be changed dynamically
Schedulability can be assessed once, assured before run-time	Dynamic changes in scheduling parameters (including passage of time!) represent on-line admission control / feasibility check points
Only dispatching done on-line	Scheduling and dispatching done on-line

Scheduling Segments

BSS -*begin_scheduling_segment*
ESS -*end_scheduling_segment*



- `RTScheduling::Current` has methods to begin, end and update scheduling segments and spawn a new DT
- Each segment's scheduling characteristics is defined by its *Scheduling Parameters*
- Nested scheduling segments allow association of different scheduling parameters
- `begin` and `end` of a scheduling segment should be on the same host

RTScheduling::Current Interface

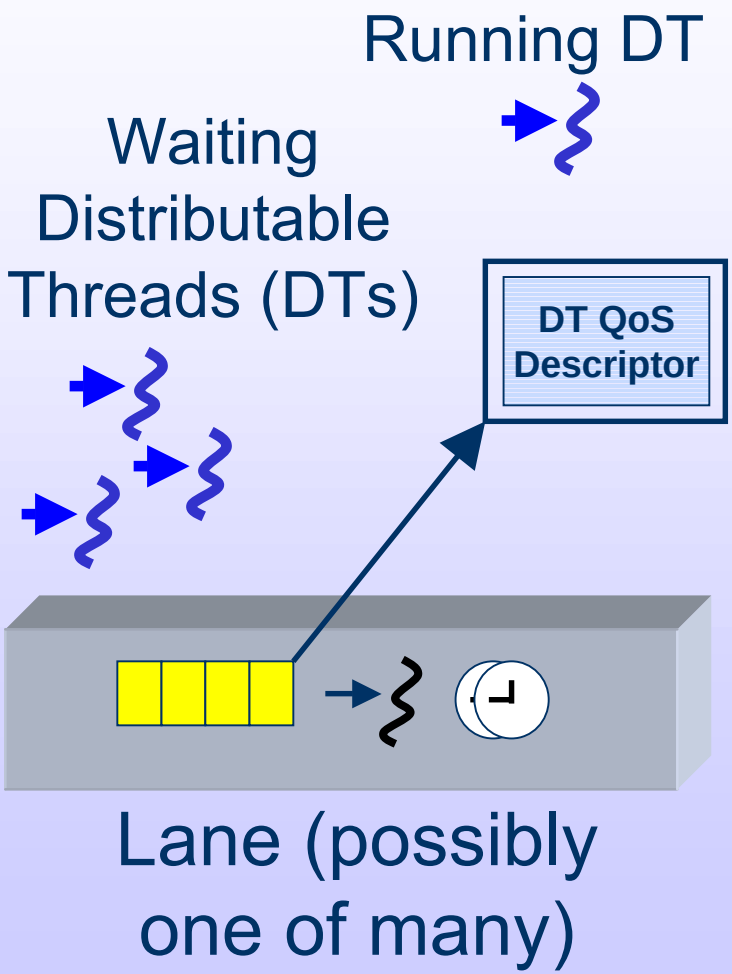
```
local interface RTScheduling::Current :
                RTCORBA::Current
{
  void begin_scheduling_segment
    (in string scheduling_segment_name,
     in CORBA::Policy sched_param,
     in CORBA::Policy
         implicit_sched_param)
  raises
    (UNSUPPORTED_SCHEDULING_DISCIPLINE);
  void update_scheduling_segment
    (in string scheduling_segment_name,
     in CORBA::Policy sched_param,
     in CORBA::Policy
         implicit_sched_param)
  raises
    (UNSUPPORTED_SCHEDULING_DISCIPLINE);
  void end_scheduling_segment
    (in string scheduling_segment_name);
```

```
DistributableThread
    spawn (in ThreadAction start,
     in CORBA::VoidData data,
     in string name,
     in CORBA::Policy sched_param,
     in CORBA::Policy
         implicit_sched_param,
     in unsigned long stack_size,
     in RTCORBA::Priority base_priority);
};


module RTScheduling {
  local interface ThreadAction {
     void do(in CORBA::VoidData data);
  };
};
```

- Derived from **`RTCORBA::Current`** interface
- Encapsulates scheduling information of a scheduling segment
- Each DT is associated with a corresponding **`Current`**
- Stored in thread specific storage (key TSS vs. DTSS issues)
- **`Current`** objects of nested scheduling segments are chained

Pluggable Scheduling Policies/Mechanisms



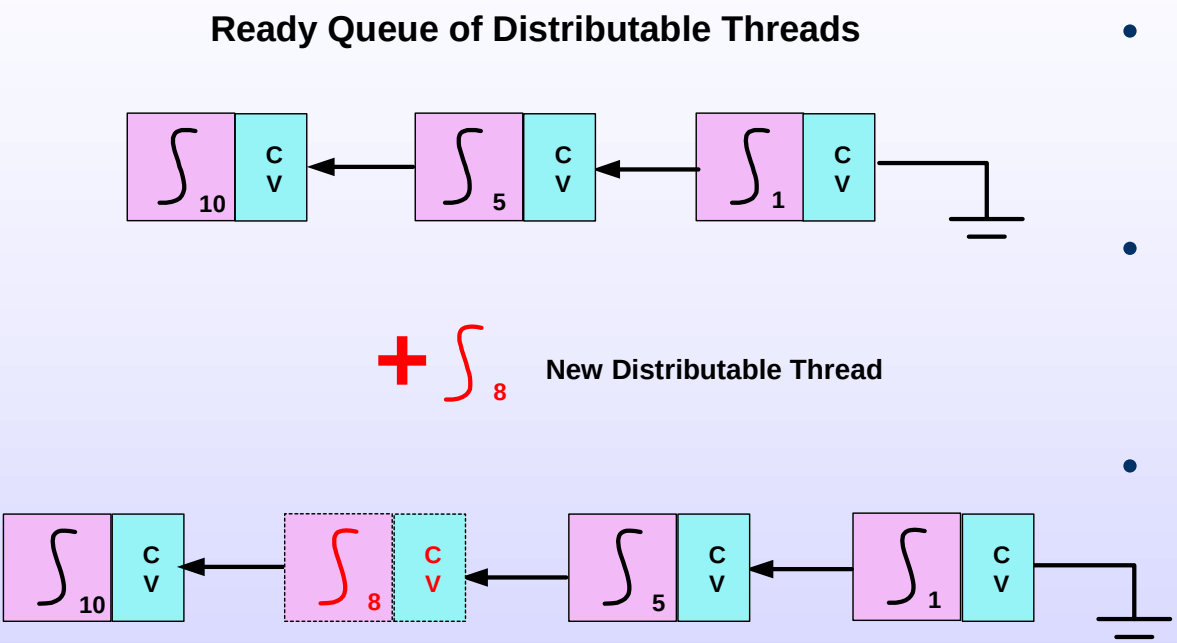
- Scheduling/Dispatching
  - Enforce predictable behavior in DRE systems
- Alternative disciplines
  - RMS, MUF, EDF, LLF
  - Custom scheduling disciplines dictated by system requirements
  - Queried via `resolve_initial_references (RTScheduler)`
- `RTScheduling::Scheduler` interface
  - From which implementations are derived
  - Interactions with application and ORB
- Scheduler Managers
  - Install/manage schedulers in the ORB

Scheduler Bootstrapping Interface

```
interface RTScheduling::Manager
{
   Scheduler scheduler ();
   void scheduler (Scheduler);
};
```

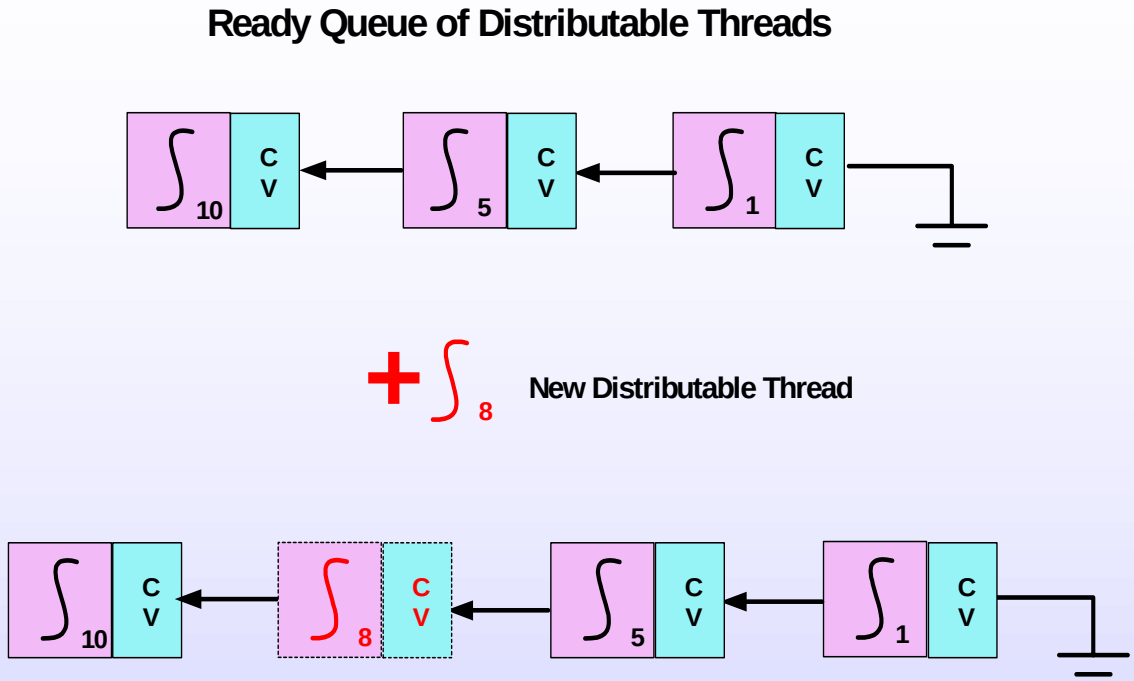
- RTScheduling::Manager interface provides set/get methods
- To obtain manager, call **`ORB:resolve_initial_reference` with `"RTScheduler_Manager"`**
- Can then get or replace the scheduler implementation

Pluggable Scheduling: Most Important First (MIF) Scheduler Implementation



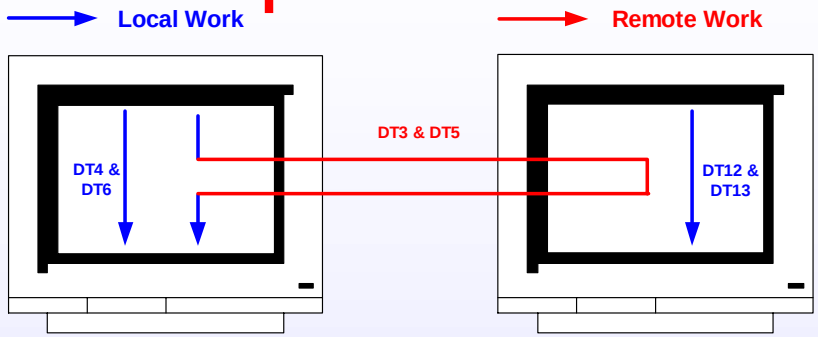
- MIF Scheduler maintains a ready queue of DTs
- DTs are queued in order of their importance
- DTs in a queue are suspended on a Condition Variable
- Each executing DT yields to the scheduler at regular intervals

Middleware Based MIF Scheduling



- Benefits: scalable in # of distributable threads per OS thread
- Drawbacks: queue management costs for some sched policies
- Alternatives: OS thread per distributable thread, lanes

Fixed Priority vs. MIF Scheduler Experiments



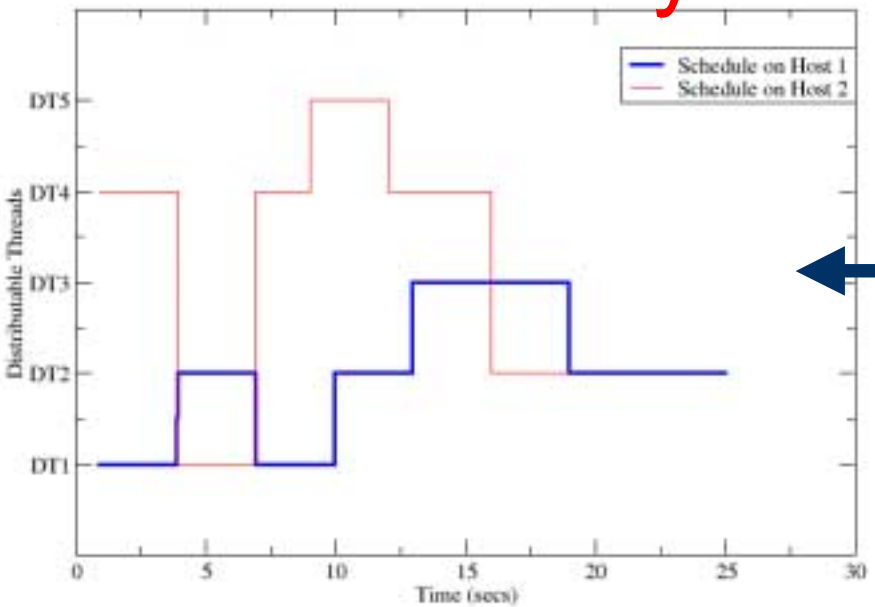
DT GUID on Host2	Start Time T (secs)	Importance	Execution Time (secs)	
			Local	Dist
12	0	15	10	-
13	5	2	15	-

DT GUID on Host2	Start Time T (secs)	Importance	Execution Time (secs)	
			Local	Dist
12	1	2	15	-
13	1	15	10	-

DT GUID on Host1	Start Time T (secs)	Importance	Execution Time (secs)	
			Local	Dist
3	4	10	5	5
4	7	8	10	-
5	4	5	5	5
6	7	4	5	-

DT GUID on Host1	Start Time T (secs)	Importance	Execution Time (secs)	
			Local	Dist
3	1	5	6	5
4	1	8	3	-
5	1	4	6	5
6	5	10	3	-

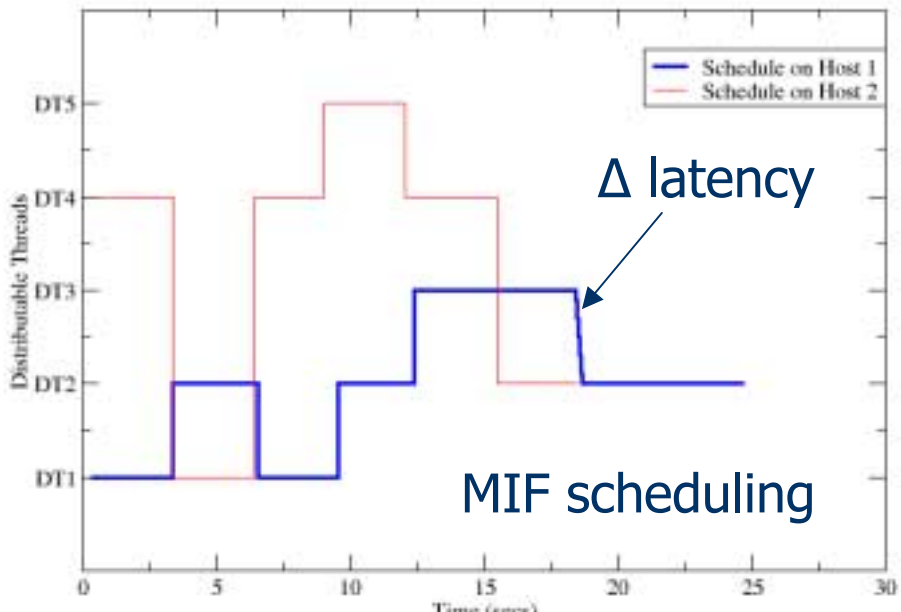
Fixed Priority vs. MIF Scheduler, Cont.



OS-level mechanism

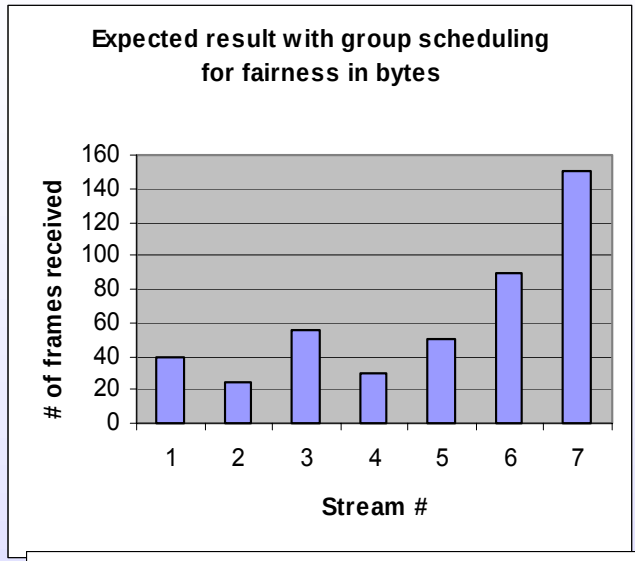
Fixed priority scheduling

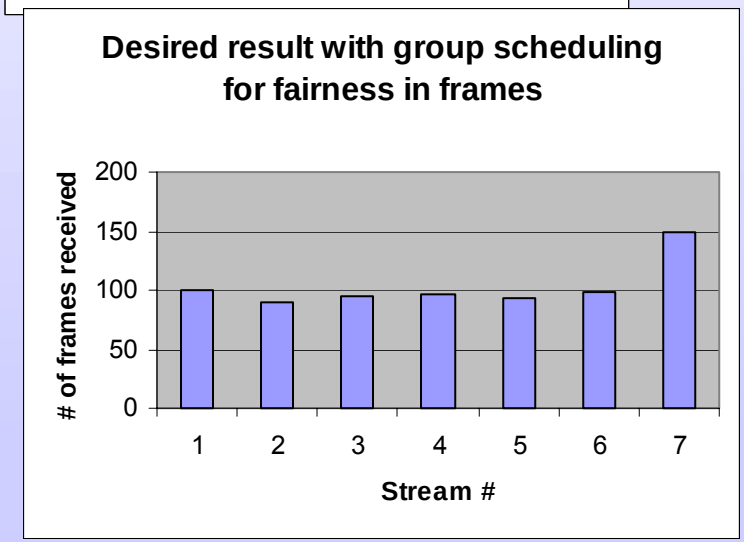
Middleware-level mechanism



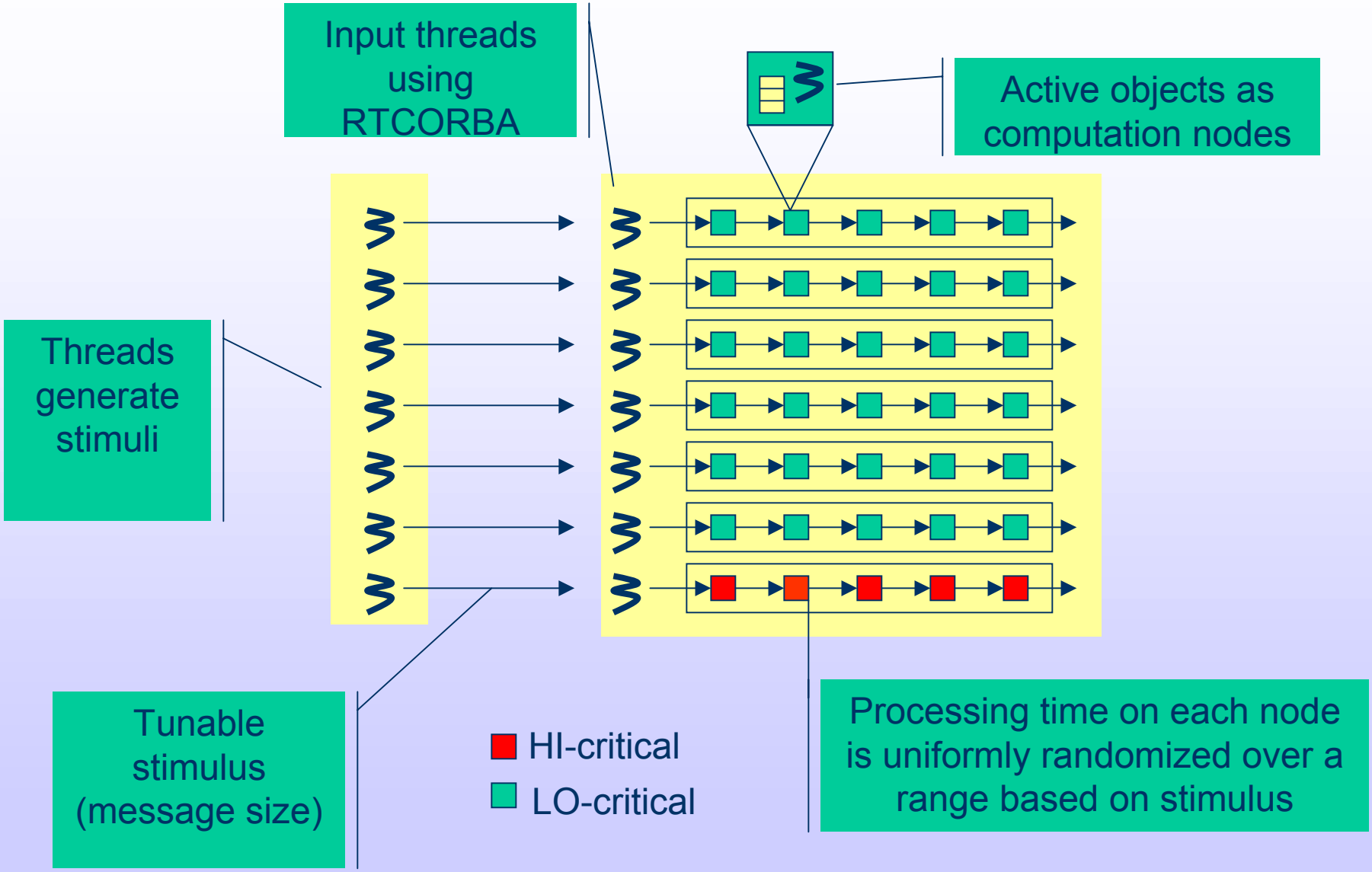
More Middleware/OS Dynamic Scheduling

- Share-based resource allocation may not give sufficient control
  - Varying image sizes, contents →
  - Dynamic communication & computation requirements
- Need an application-based definition of ***progress***
  - E.g., # of frames examined for a particular kind of TCT
    - # of bytes per frame could vary
    - Computation time could also vary
    - **Scheduling algorithm adapts to these dynamic changes**
    - Need to convey progress
      - Published by application/MW
      - Used by Group Scheduling



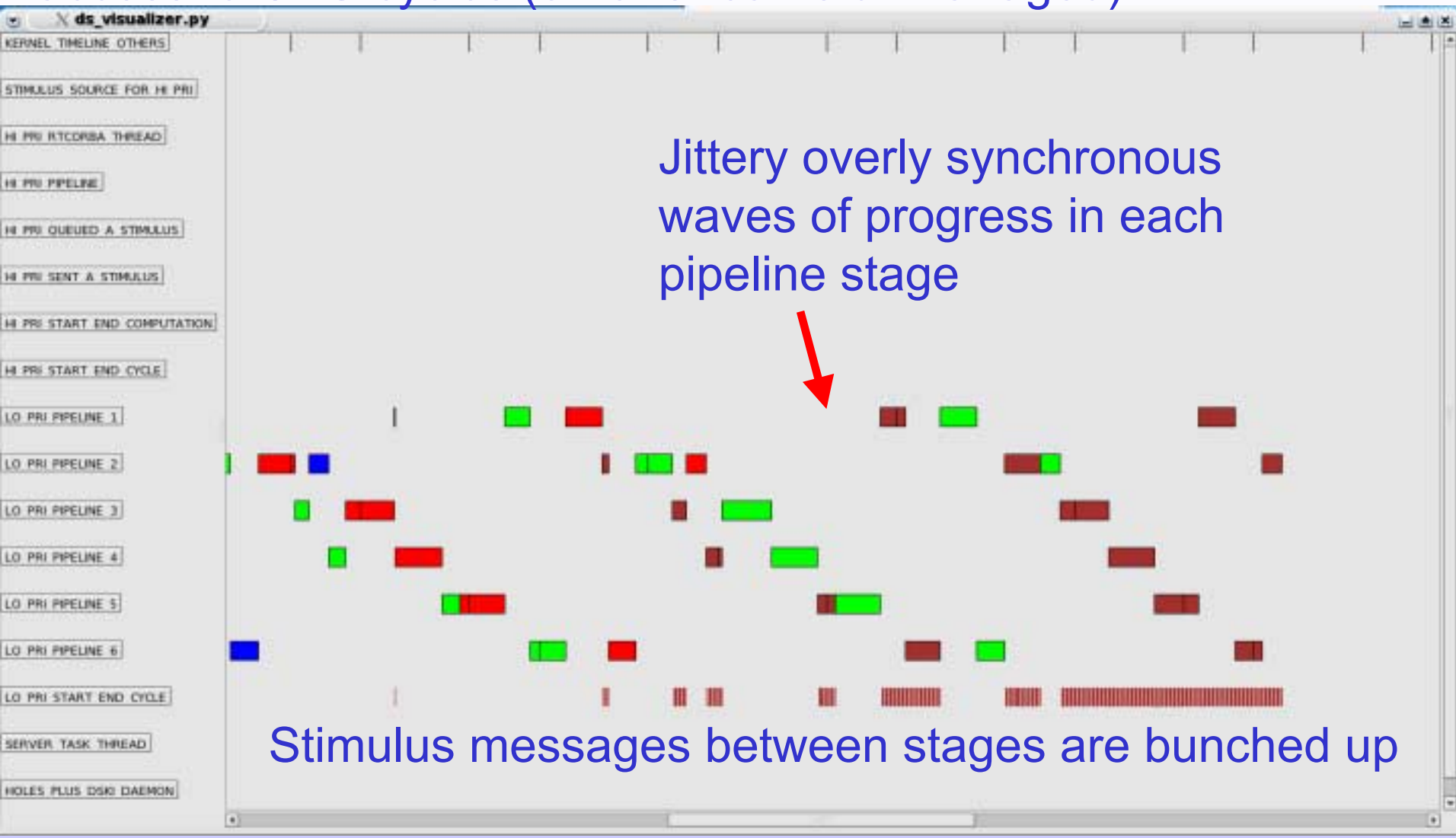


Metric: *Frames* Processed per Stream



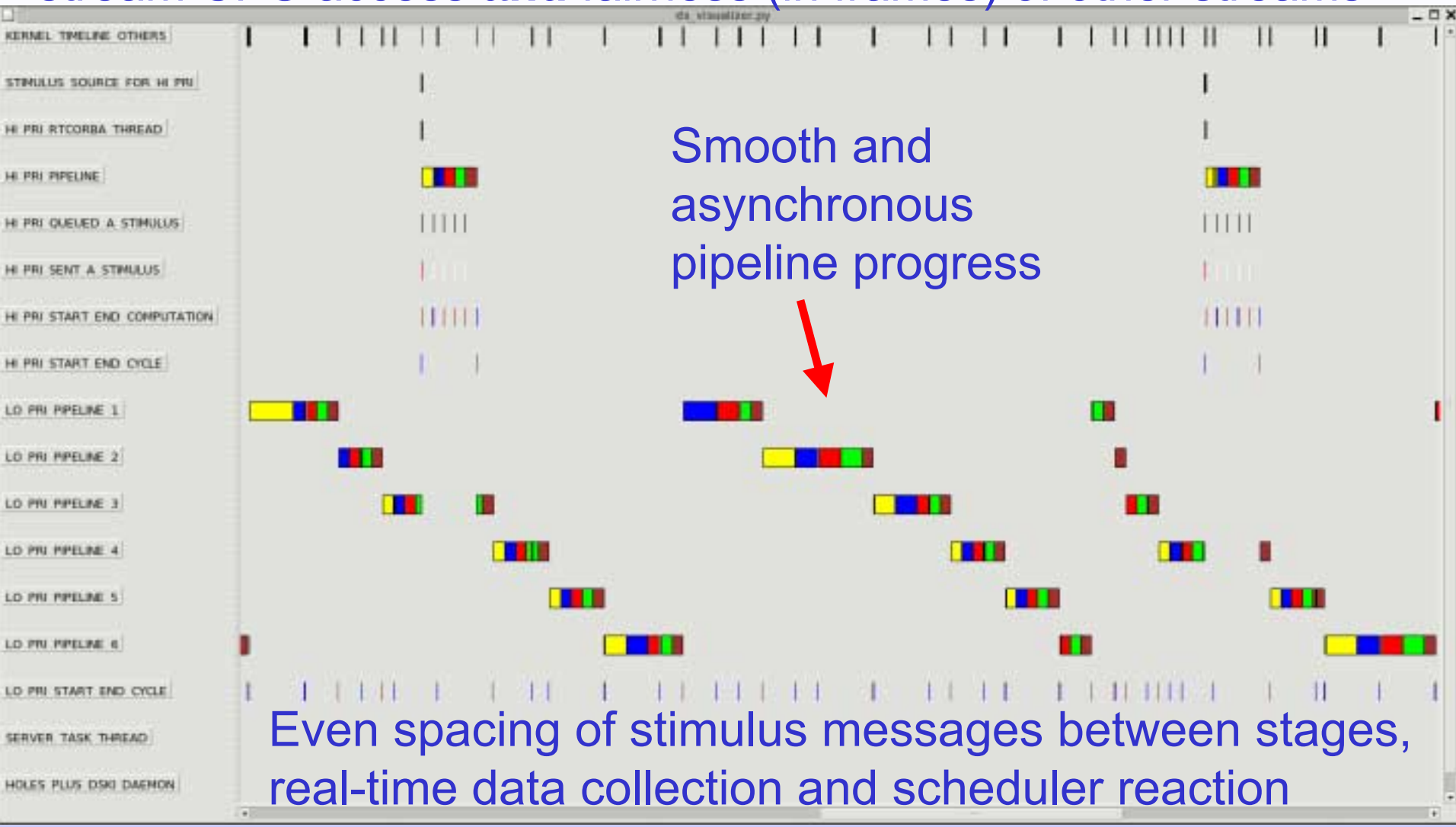
Unmanaged Pipelines "Stop-and-Go"

Kernel-level group scheduler enforces critical stream CPU access to CPU cycles (other streams unmanaged)



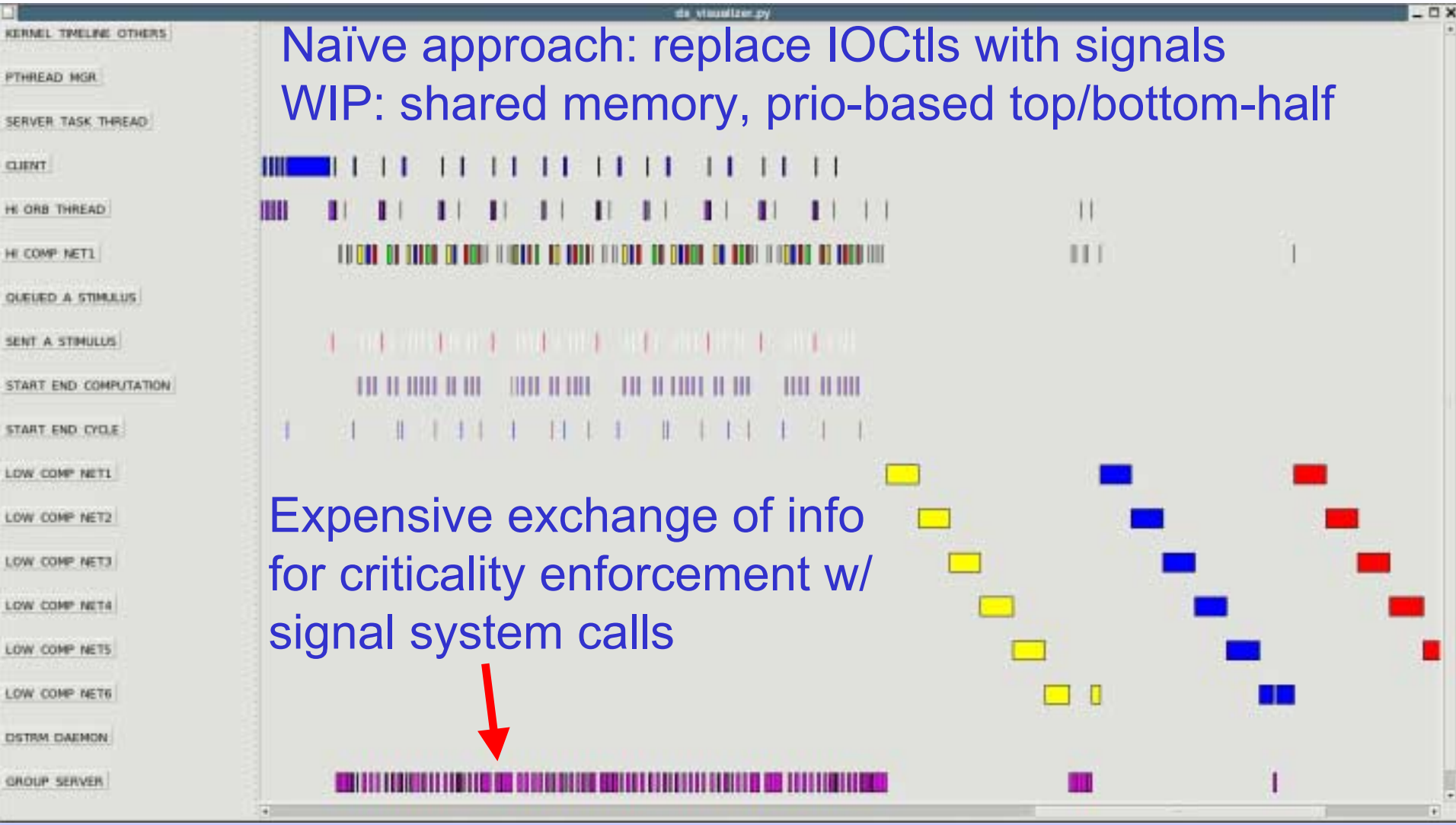
Pipelines Made "Progress Fair"

Re-ran with kernel-level group scheduler to enforce critical stream CPU access **and** fairness (in frames) of other streams



Toward Efficient Middleware GS

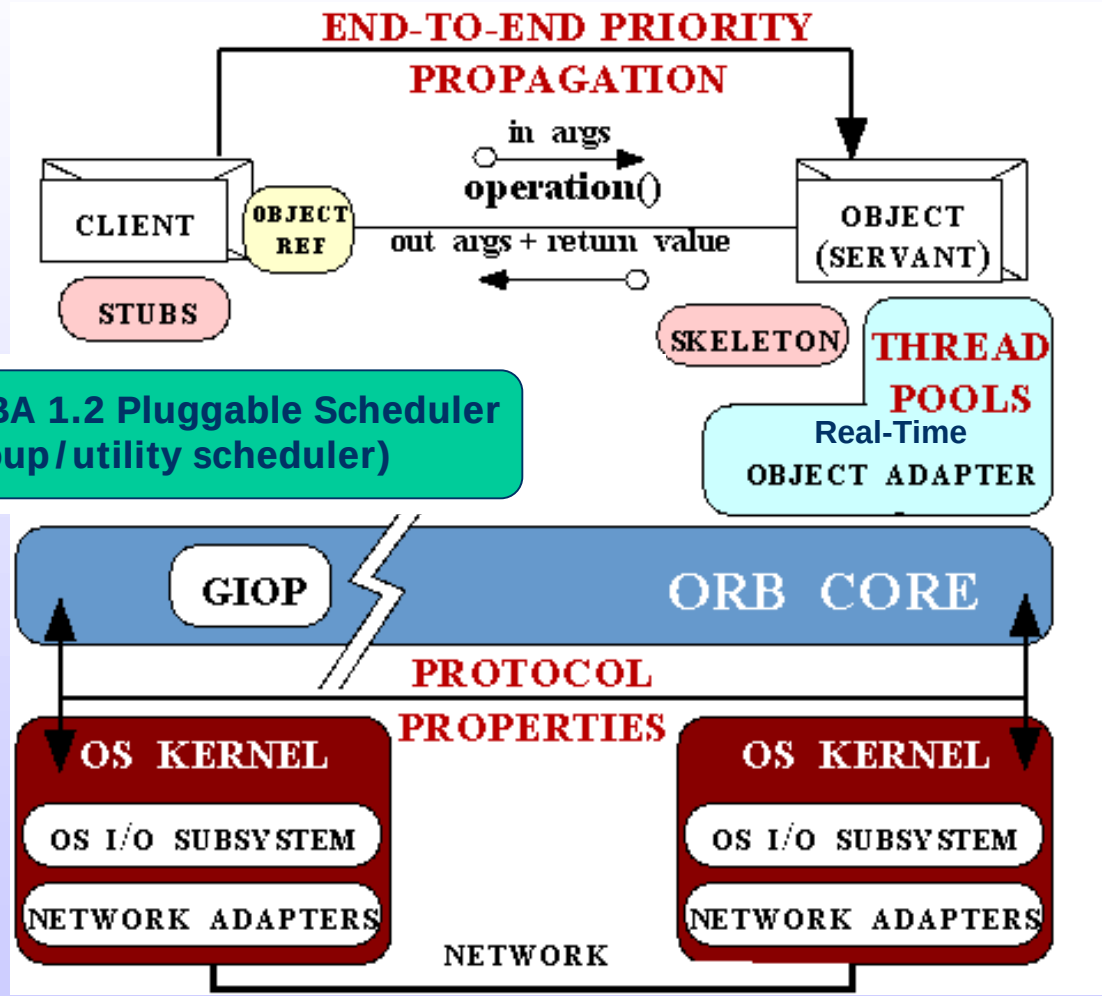
Open design challenge: cutting cost of telling MW scheduler progress info (kernel scheduler can already access implicitly



Future Evolution: Integration with Distributed Dynamic Scheduling Service

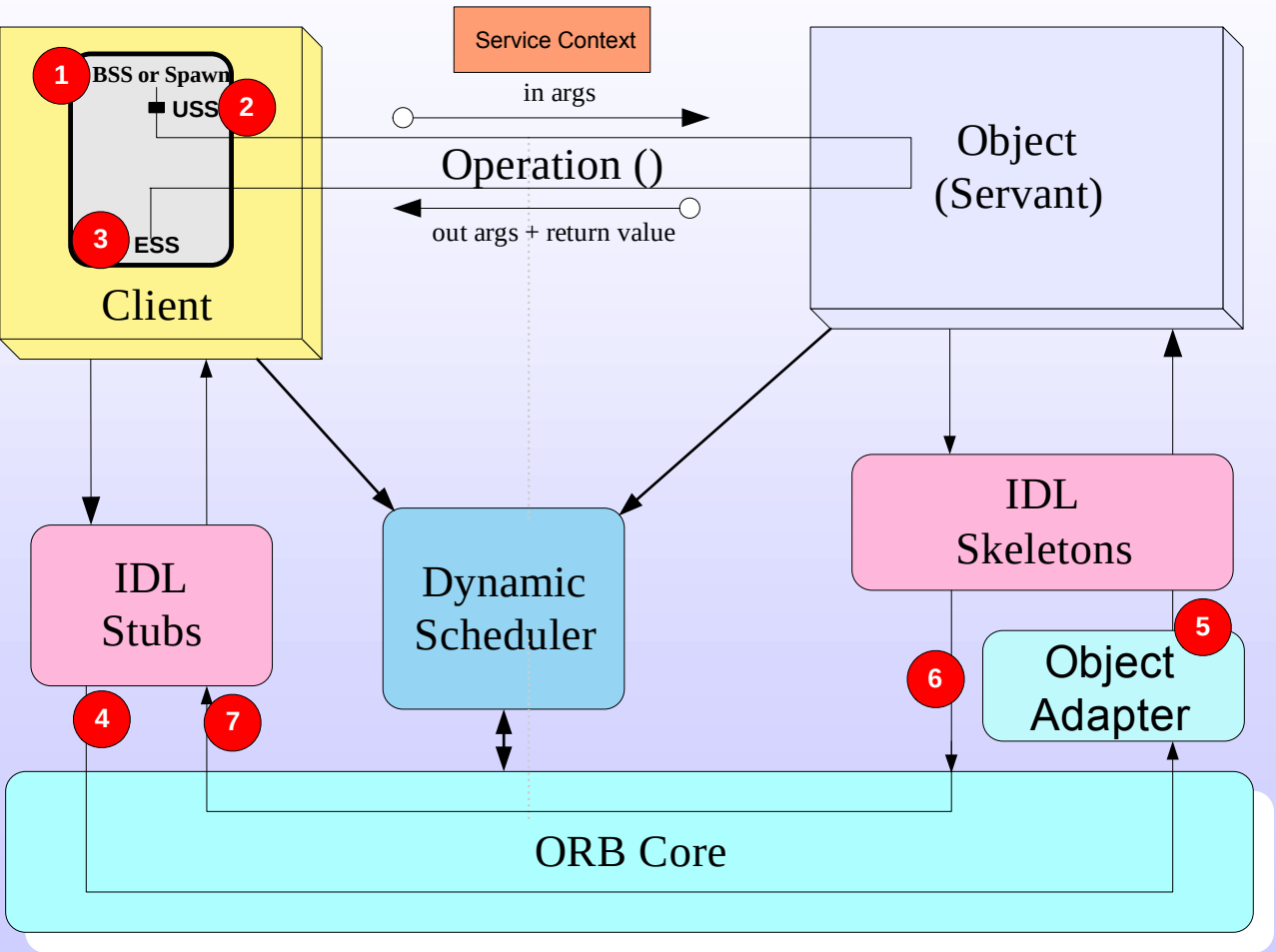
Distributed Scheduling Service

- Distributed Scheduling Service works with ORB local scheduler to enforce global scheduling
- Set *end-to-end* Urgencies (MUF) or raw CORBA priorities for DTs
- Determine cancellation points for overload management
- Interact with future scheduling adaptation mechanisms



Thanks to Douglas Niehaus (KU), Lisa DiPippo and Victor Fay-Wolfe (URI) for the use of slides from the collaborative research funded by the DARPA PCES program.

Scheduling Points



1. Begin scheduling segment or spawn
2. Update scheduling segment
3. End scheduling segment
4. Send request
5. Receive request
6. Send reply
7. Receive reply

Scheduling Points, Continued

User / ORB	*Scheduler Upcall*
`Current::spawn()`	`begin_new_scheduling_segment()`
`Current::begin_scheduling_segment()` [*when creating DTs*]	`begin_new_scheduling_segment()`
`Current::begin_scheduling_segment()` [*when creating nested segments*]	`begin_nested_scheduling_segment()`
`Current::update_scheduling_segment()`	`update_scheduling_segment()`
`Current::end_scheduling_segment()` [*when ending scheduling nested segments*]	`end_nested_scheduling_segment()`
`Current::end_scheduling_segment()` [*when destroying DTs*]	`end_scheduling_segment()`
`DistributableThread::cancel()`	`cancel()`
`Outgoing request`	`send_request()`
`Incoming request`	`receive_request()`
`Outgoing reply`	`send_reply()`
`Incoming reply`	`receive_reply()`

Implementation Challenges II: Distributable Threads

- Distributable threads span multiple heterogeneous hosts
- Different OS threads host multiple distributable threads
- Both OS and distributable thread identities must be represented explicitly
- Cancellation is made even more challenging by distribution
  - Network partition, asynchronous communication
- Correctness of thread-identity aware mechanisms
  - *E.g.*, TSS, recursive locks, leader-followers pattern
  - Need awareness of both OS tid and distributed UUID
  - May require emulation in middleware: performance cost

Motivation

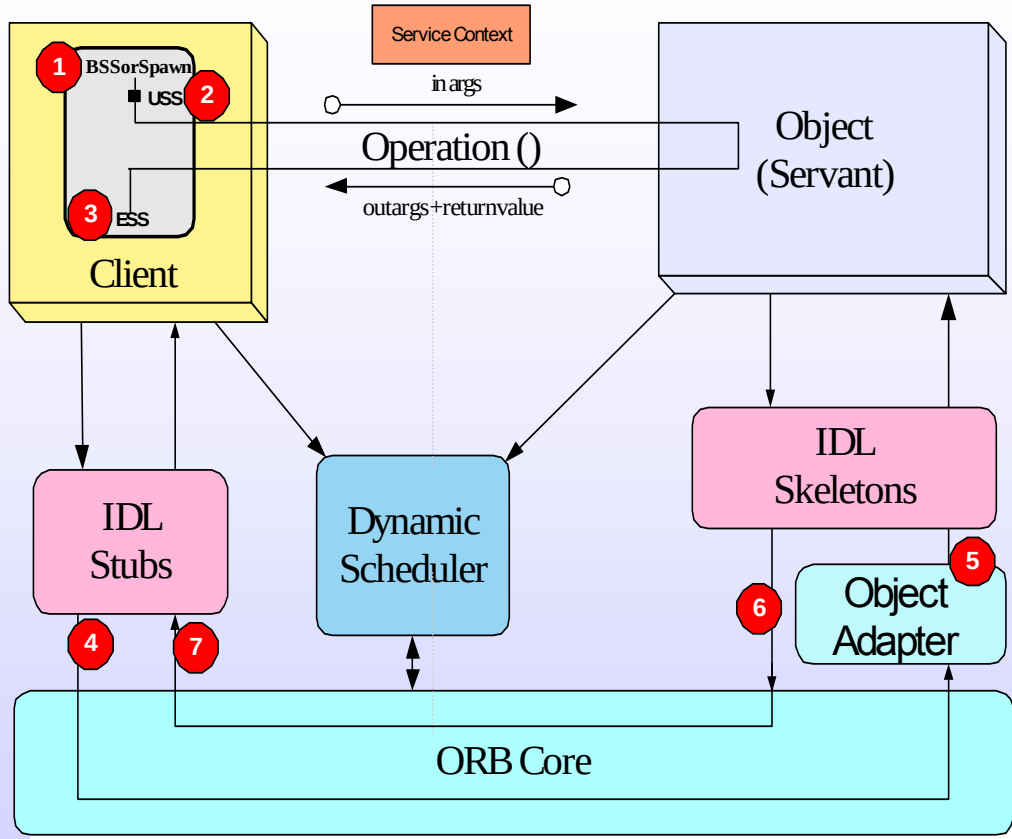
- An evolving middleware programming model
  - Started with RPC-style remote function call abstractions
  - CORBA, RMI, COM+ introduced method invocations
  - Naming, Event, Trader, and Notification services added object lookup, anonymous communication, service lookup
  - Many applications can be built using only these services
- An important "new" middleware abstraction
  - Distributable threads are natural for certain applications
    - *I.e.*, long-running distributed sequential activities
  - Integrated with dynamic scheduling semantics
- Design and implementation goals
  - Allow flexible on-the-fly adaptation of functional and real-time properties on the paths a distributable thread traverses
  - Provide efficient and predictable enforcement mechanisms

Implementation Differences for Distributable Threads vs. Other Models

Distributable Thread	EC / Notify	RT CORBA Call
End-to-end QoS	Point-to-point QoS	Point-to-point QoS
Parameters passed in Context field or in distributable thread	Parameters passed in Context field or in event	Parameters passed in Context field
Interceptors map parameters from Current (and/or scheduler) to Context field and back	Interceptors or handlers map parameters from Current or event to Context field & back	ORB core maps parameters from Current to Context field and back

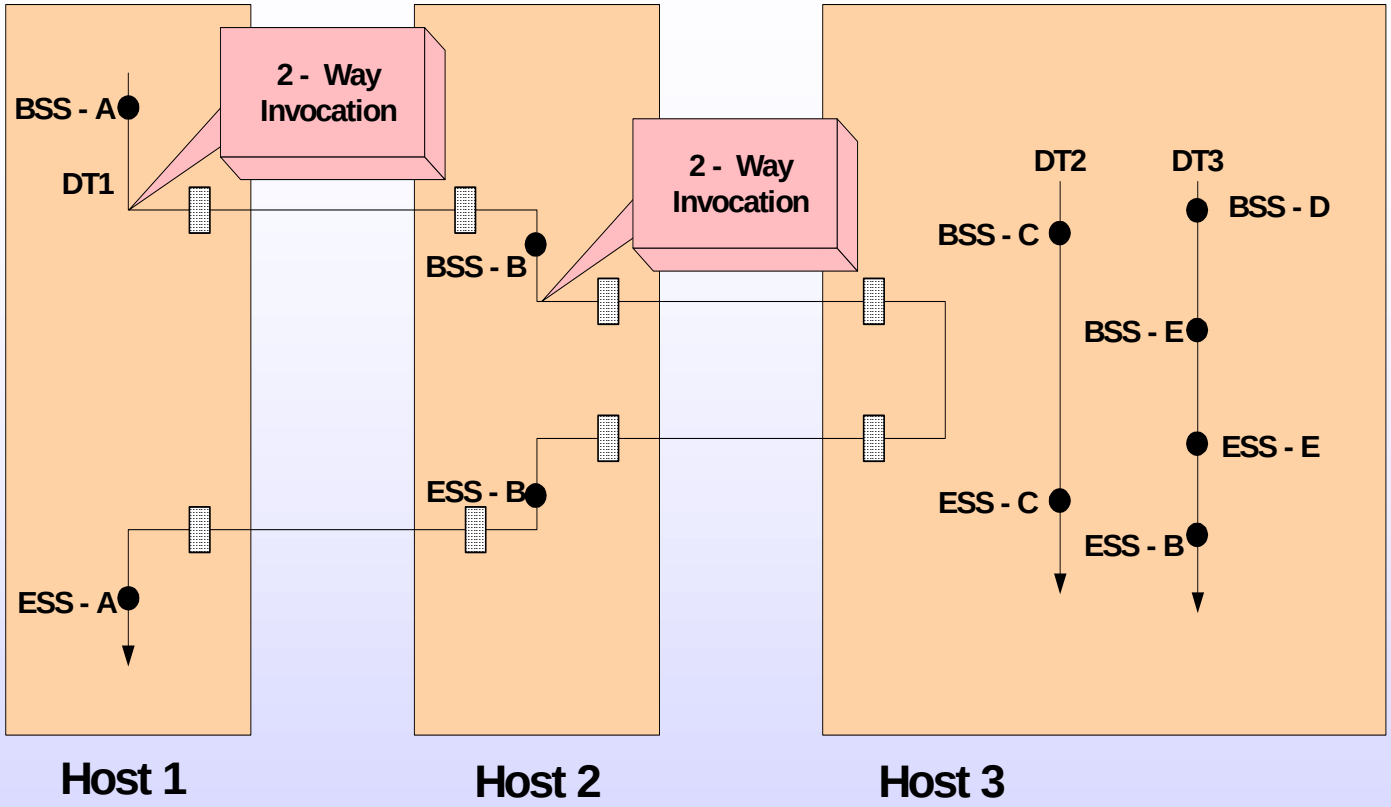
Distributable Thread Path Example

- Scheduler upcalls are done at several points on path
  - At creation of a new distributable thread
  - At BSS, USS, ESS calls
  - When a GIOP request is sent
  - On receipt of GIOP request
  - When GIOP reply is sent
  - When GIOP reply is received
- At each upcall point, scheduling information is updated
  - Additional interception points can (and sometimes should) be supported by the ORB and the scheduler/policy



1. BSS - RTScheduling::Current::begin_scheduling_segment() or RTScheduling::Current::spawn()
2. USS - RTScheduling::Current::update_scheduling_segment()
3. ESS - RTScheduling::Current::end_scheduling_segment()
4. send_request() interceptor call
5. receive_request() interceptor call
6. send_reply() interceptor call
7. receive_reply() interceptor call

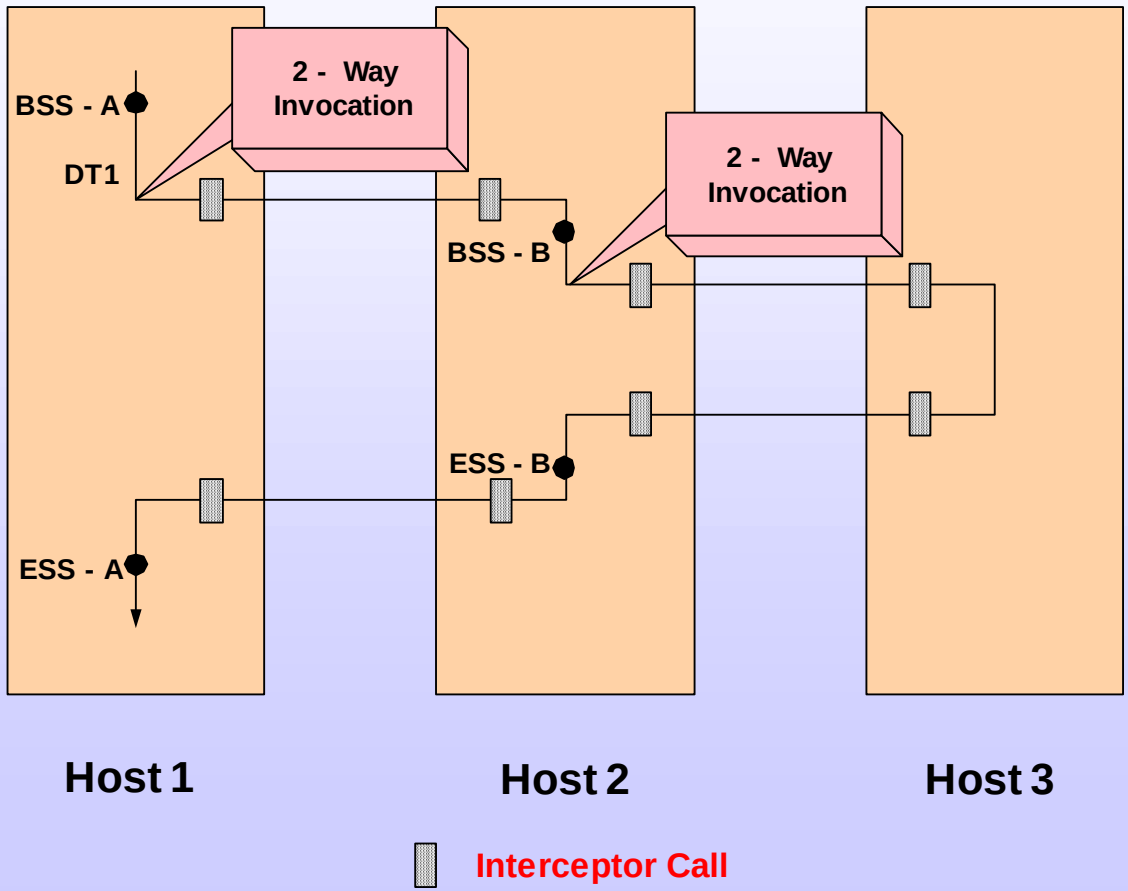
CORBA Distributable Threads



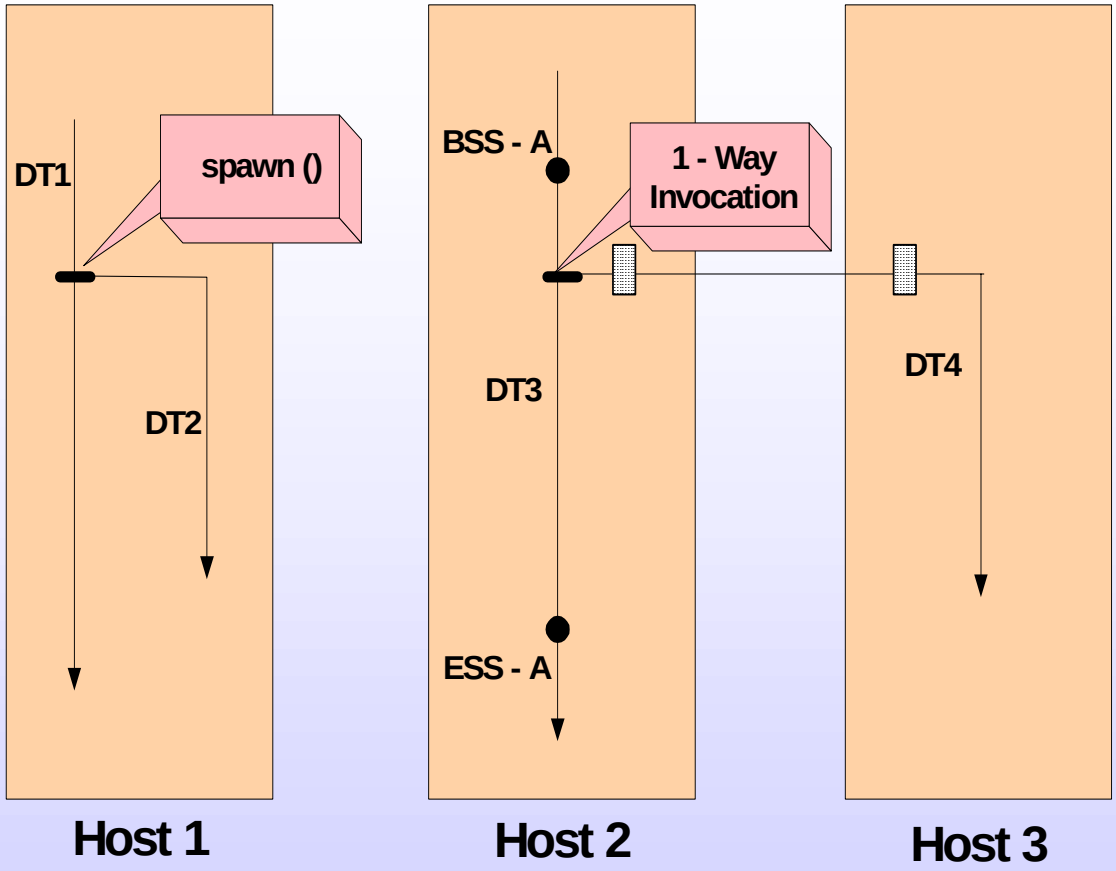
- With only 2-way CORBA invocations, distributable threads behave much like traditional OS threads
  - Though they and their context can move from one endsystem to another
  - Results in different resource scheduling domains being transited
- Distributable threads contend with OS threads and each other
  - With locking, this can span endsystems, though scheduling is local

Distributable Threads May Span Hosts

Distributable Thread traversing multiple hosts with two-way invocations



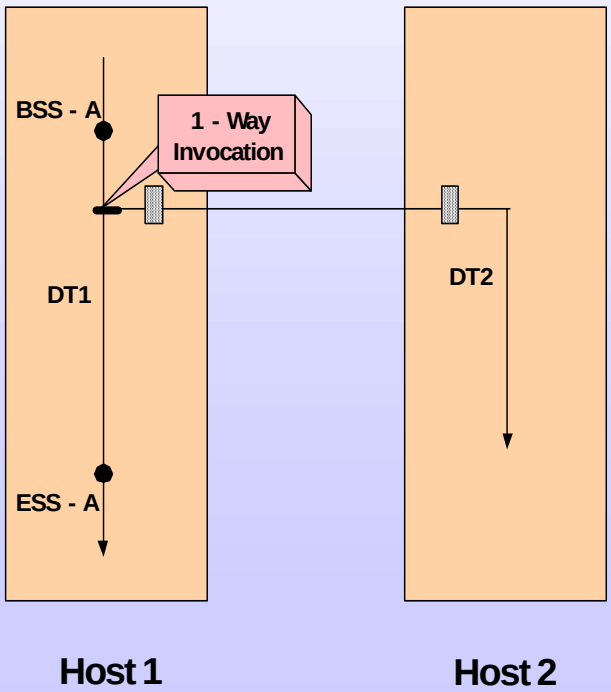
Creation of Distributable Threads



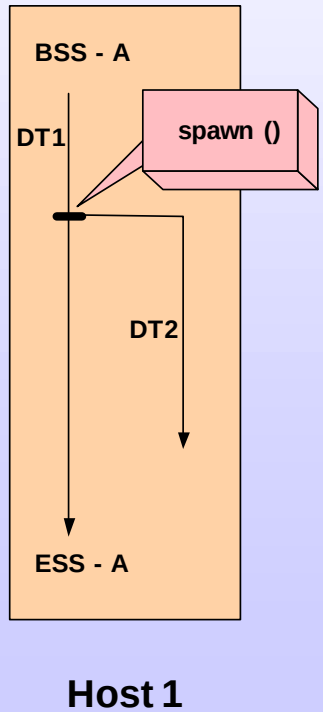
- Distributable threads can be created in three different ways
  - An application thread calling BSS outside a distributable thread
  - A distributable thread calling the **`spawn()`** method
  - A distributable thread making an asynchronous (one-way) invocation
- The new distributable thread inherits default sched parameters

Distributable Thread Creation, Cont.

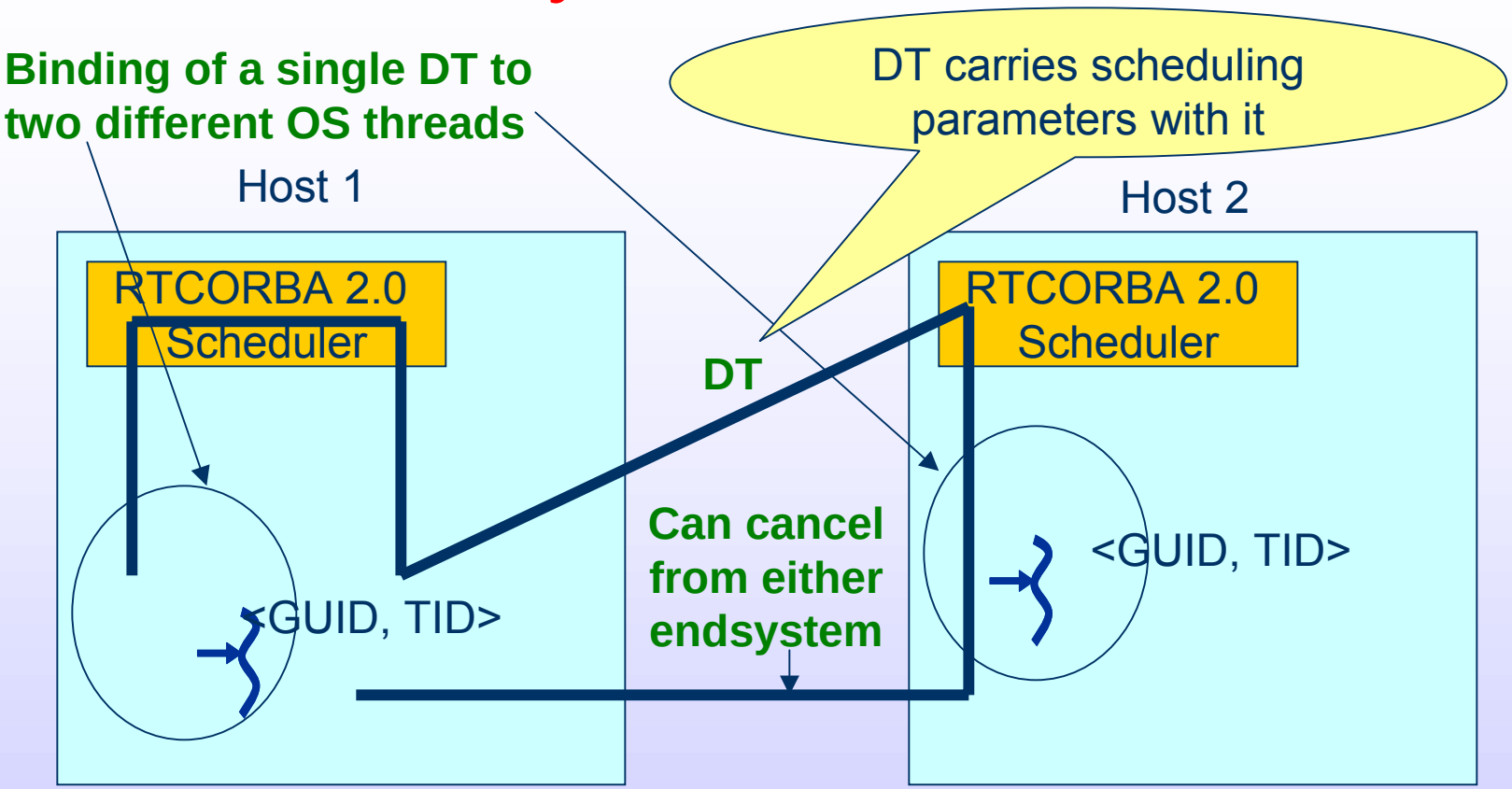
- Distributable Thread making One-Way invocation
  - New DT created implicitly
  - With implicit scheduling parameters



- Distributable Thread Spawn
  - New native thread created
  - New DT created
  - With implicit scheduling parameters

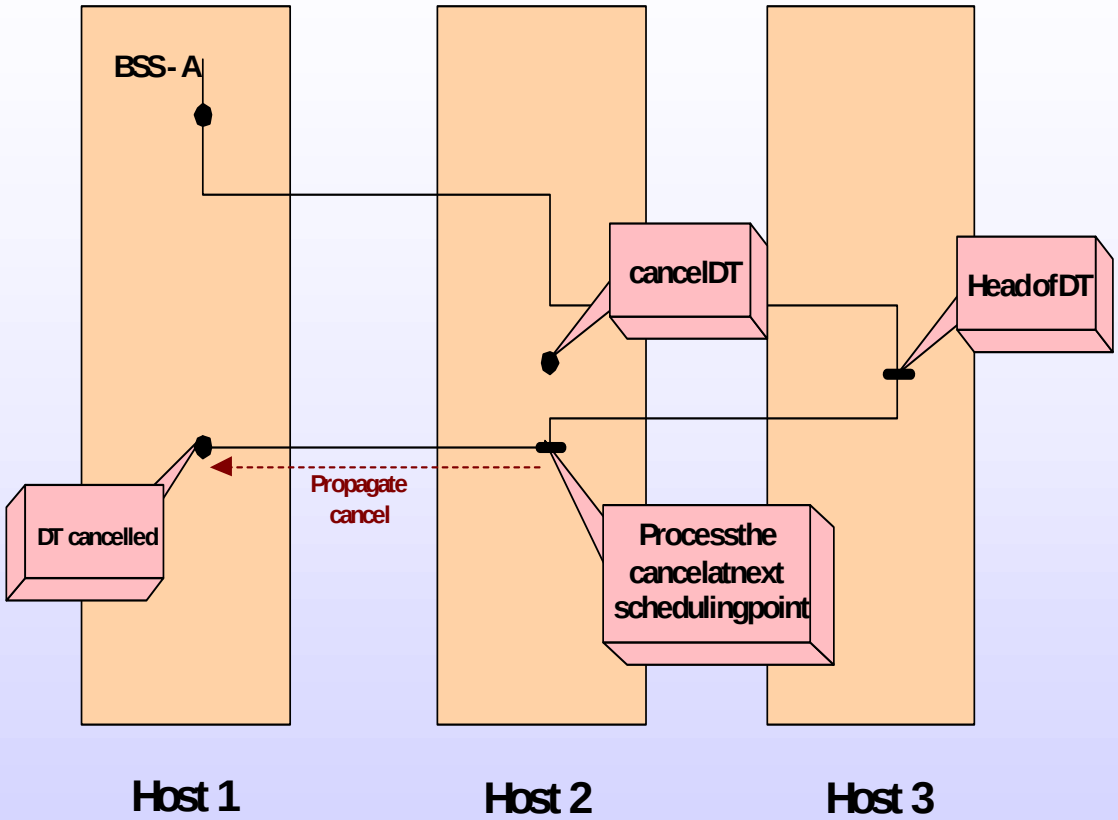


Thread Identity and Cancellation Issues



- Other mechanisms affect real-time performance, too
  - Managing identities of distributable and OS threads
  - Configuring and using mechanisms sensitive to thread identity
  - Supporting safe and efficient cancellation of thread execution
- Similar kinds of issues seen with RT Notification Service filters

Distributable Thread Cancellation

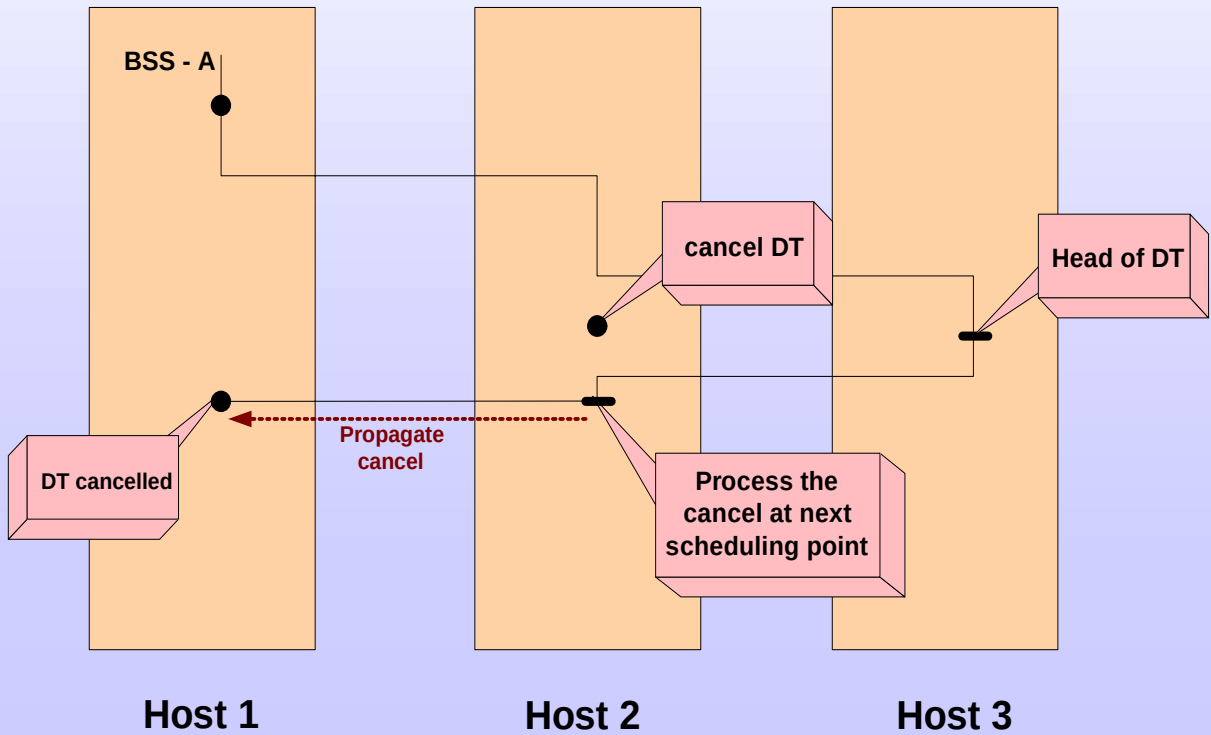


- Context: distributable thread can be cancelled to save cost
- Problem: only safe to cancel on endsystem in thread's "call stack", and when thread is at a safe preemption point
- Solution: cancellation is invoked via cancel method on distributable thread instance, handled at next scheduling point

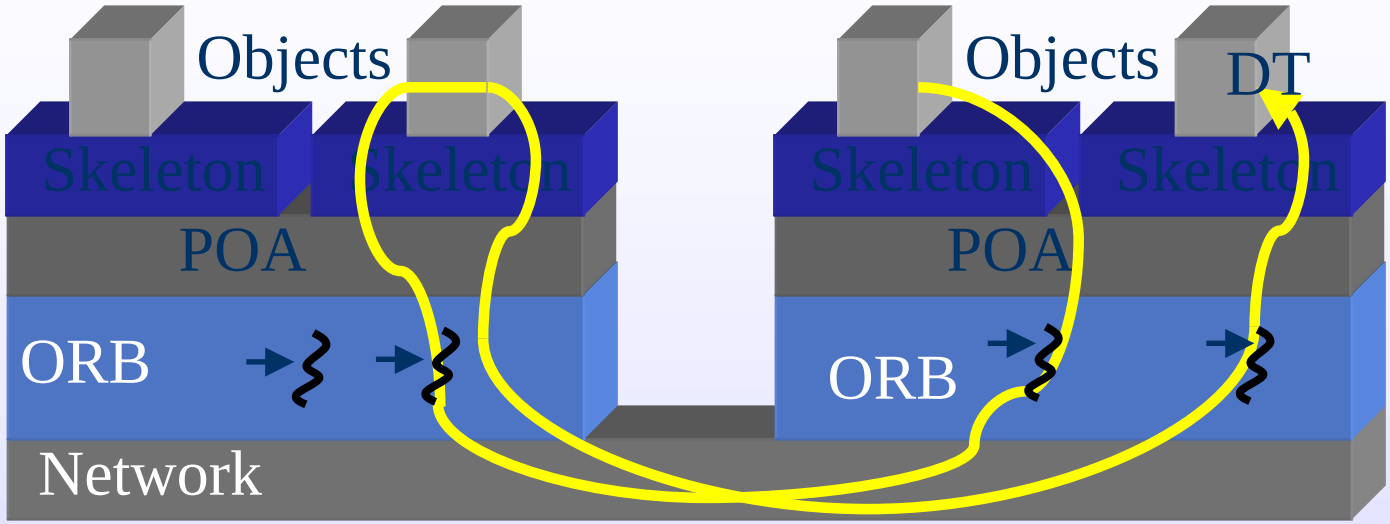
Distributable Thread Cancellation, Cont.

- DistributableThread::cancel() cancels a DT
- Raises CORBA::THREAD_CANCELLED exception at the next scheduling point
- Exception is propagated to the *start* of the DT
- … but is not propagated to the head of the DT
- DT can be cancelled on any host that it currently spans

```
  local interface
RTScheduling::DistributableThread
  {
    // raises CORBA::OBJECT_NOT_FOUND if
    // distributable thread  is not known
    void cancel();
  };
```



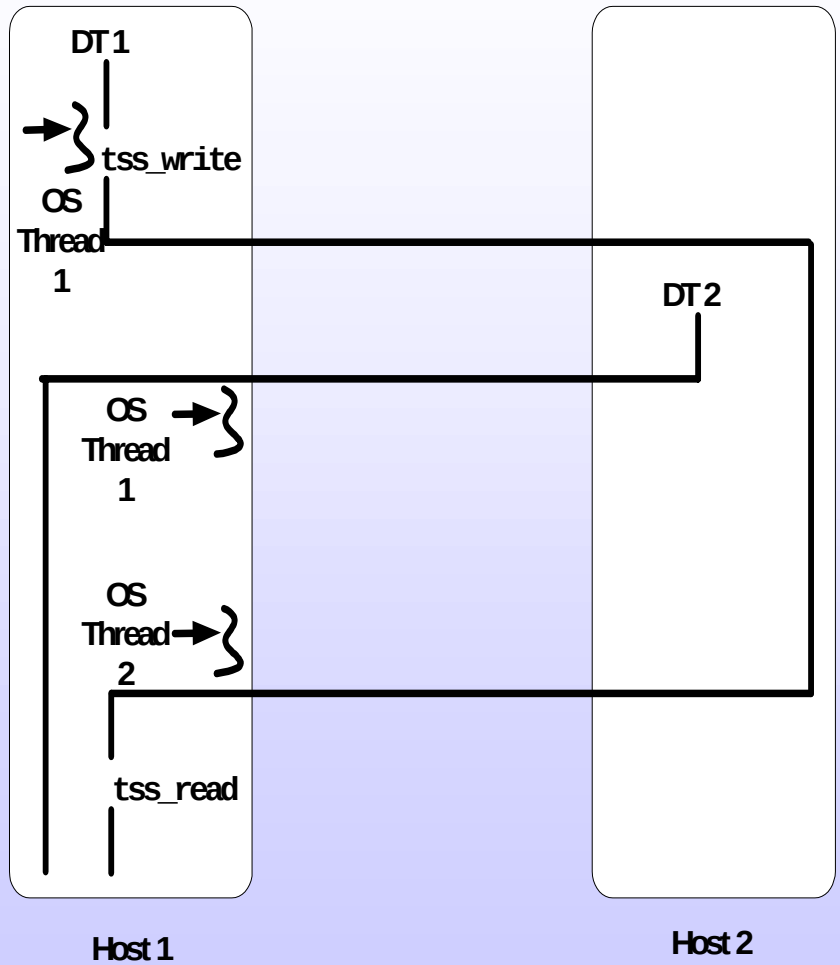
Mapping Distributable Threads to OS Threads



- DT identity (UUID) vs. OS thread identity (tid)
  - Mapping may be many-to-many and may evolve over time
  - Need UUID aware concurrency and synchronization mechanisms
    - ORB-level schedulers, but also lower-level mutexes, TSS, managers
  - DT cancellation semantics depends on ORB-level concurrency
    - *I.e.*, reactive vs. cooperative vs. preemptive

Thread Specific Storage Example

- A distributable thread can use thread-specific storage
  - Avoids locking of global data
- Context: OS provided TSS is efficient, uses OS thread id
- Problem: distributable thread may span OS threads
- Solution: TSS emulation based on **`<UUID,tid>`** pair
- Key question to answer
  - What is the cost of TSS emulation compared to OS provided TSS?



Preliminary TSS Emulation Benchmarks

- Pentium tick timestamps
  - Nanosecond resolution on 2.8 GHz P4, 512KB cache, 512MB RAM
  - RedHat 7.3, real-time class
  - Called create repeatedly
  - Then, called write/read repeatedly on one key
- Upper graph shows scalability of key creation
  - Cost scales linearly with # of keys in OS, ACE TSS
  - Emulation cost ~2usec more per key creation
- Lower graph shows that emulated TSS write costs ~1.5usec more, emulated read costs ~.5usec more