Experience Implementing and Evaluating Real-Time CORBA 1.2

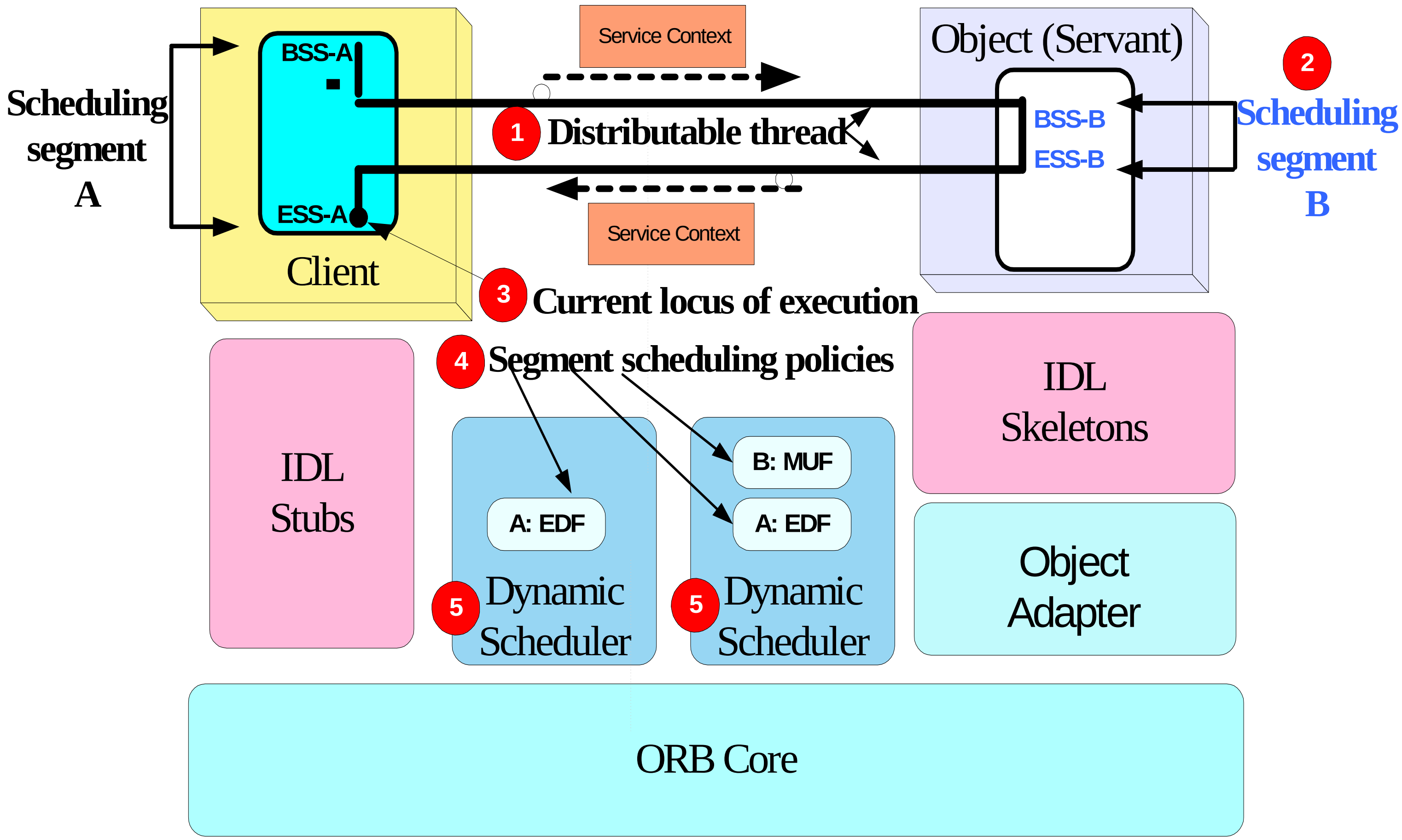
Yamuna Krishnamurthy
and Irfan Pyarali
OOMWorks
Metuchen, NJ
{yamuna, irfan}@oomworks.com

Chris Gill, Louis Mgeta,
Yuanfang Zhang, and Stephen Torri
Washington University
St. Louis, MO
{cdgill,lmm1,yfzhang,storri}@cse.wustl.edu

Douglas C. Schmidt
Vanderbilt University
Nashville,TN
schmidt@dre.vanderbilt.edu

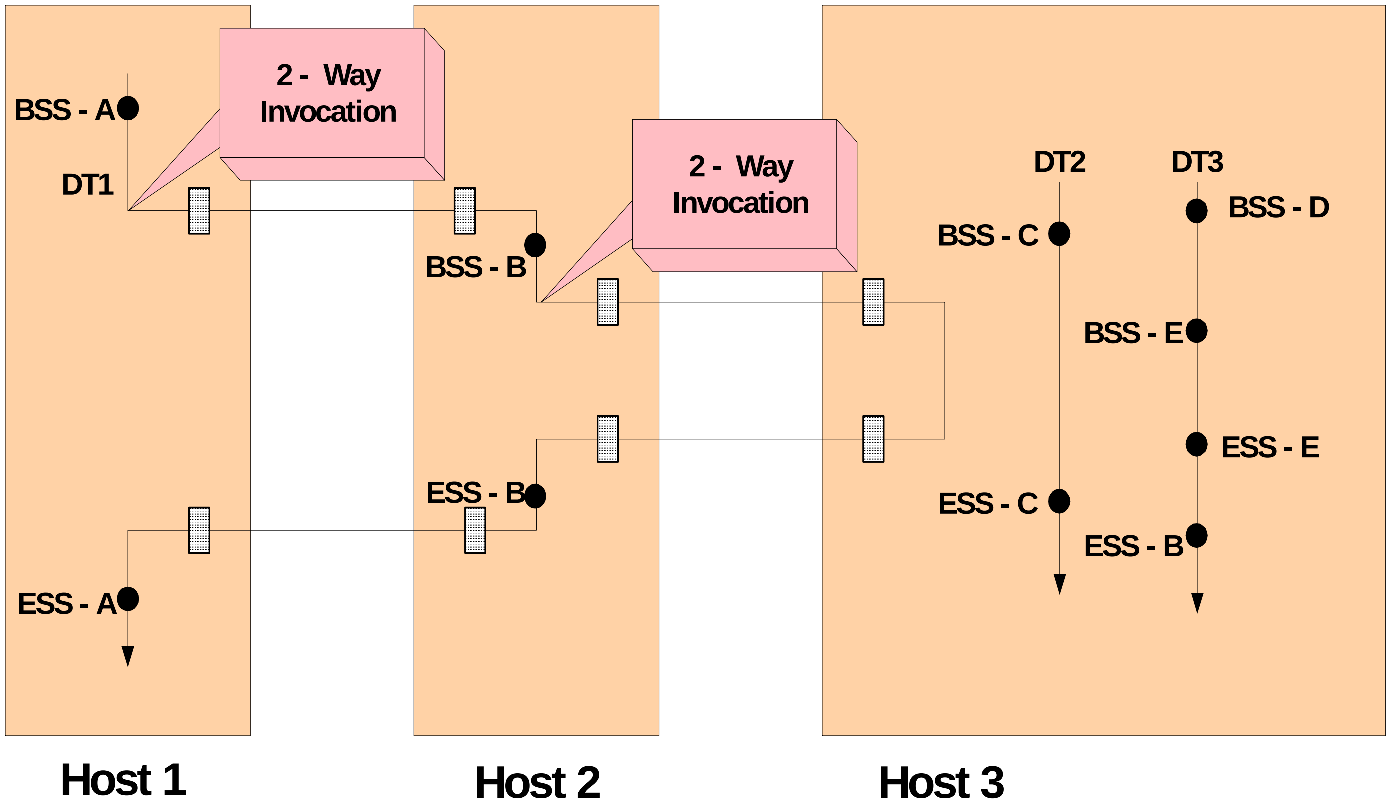
Wednesday, May 26, 2004
OMG Real-Time and Embedded Systems Workshop
July 2004, Reston, VA

Summary of RT CORBA 1.2 Concepts



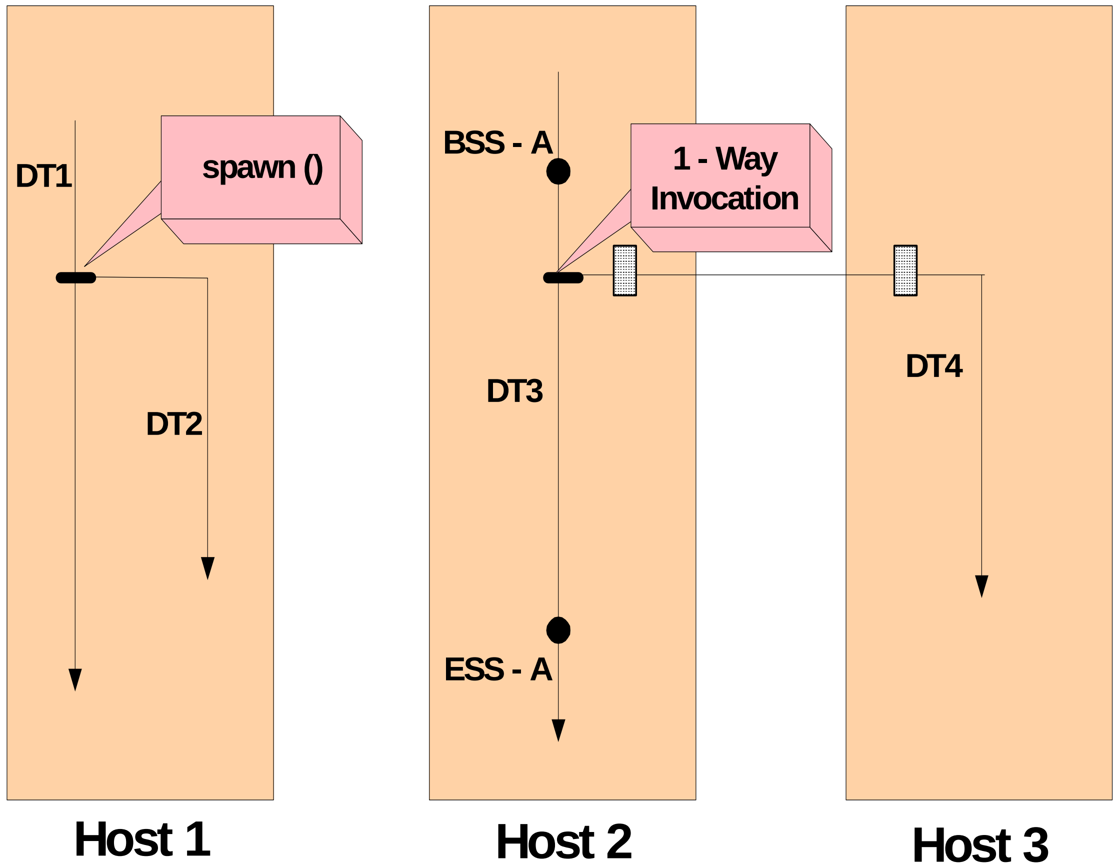
- Distributable thread – distributed concurrency abstraction
- Scheduling segment – governed by a single scheduling policy
- Locus of execution – point at which distributable thread is currently running
- Scheduling policies – determine eligibility of different distributable threads
- Dynamic schedulers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

Intro: 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

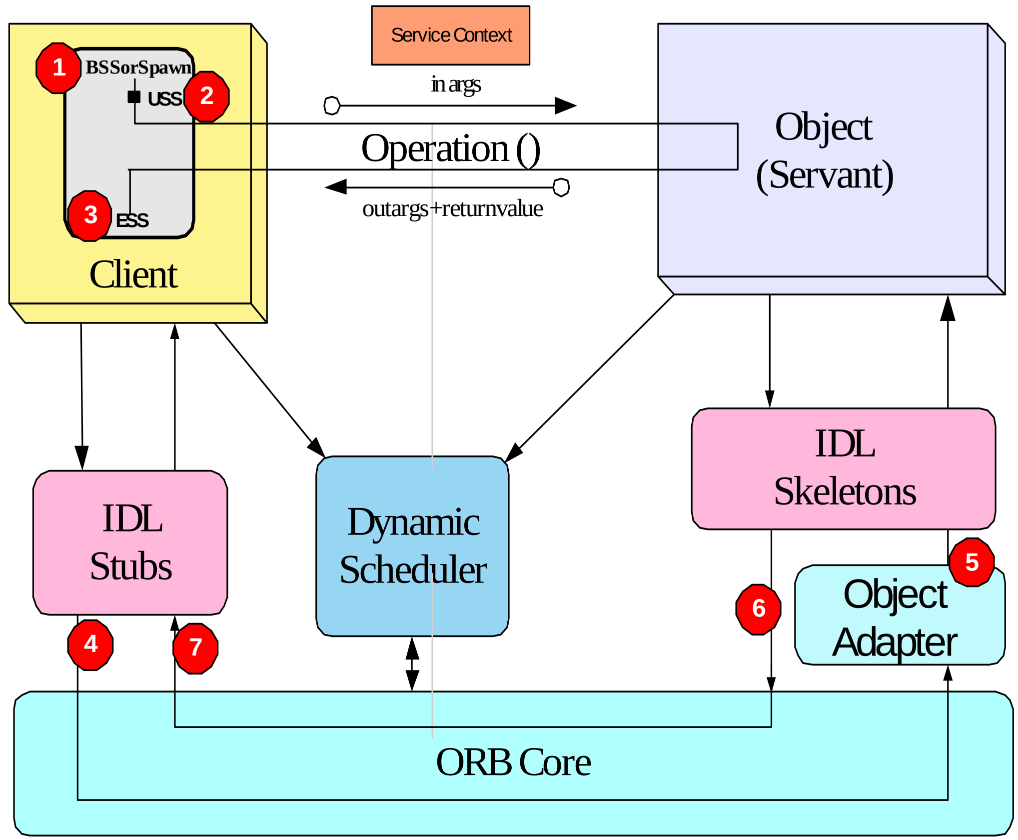
Creation of Distributable Threads



- 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

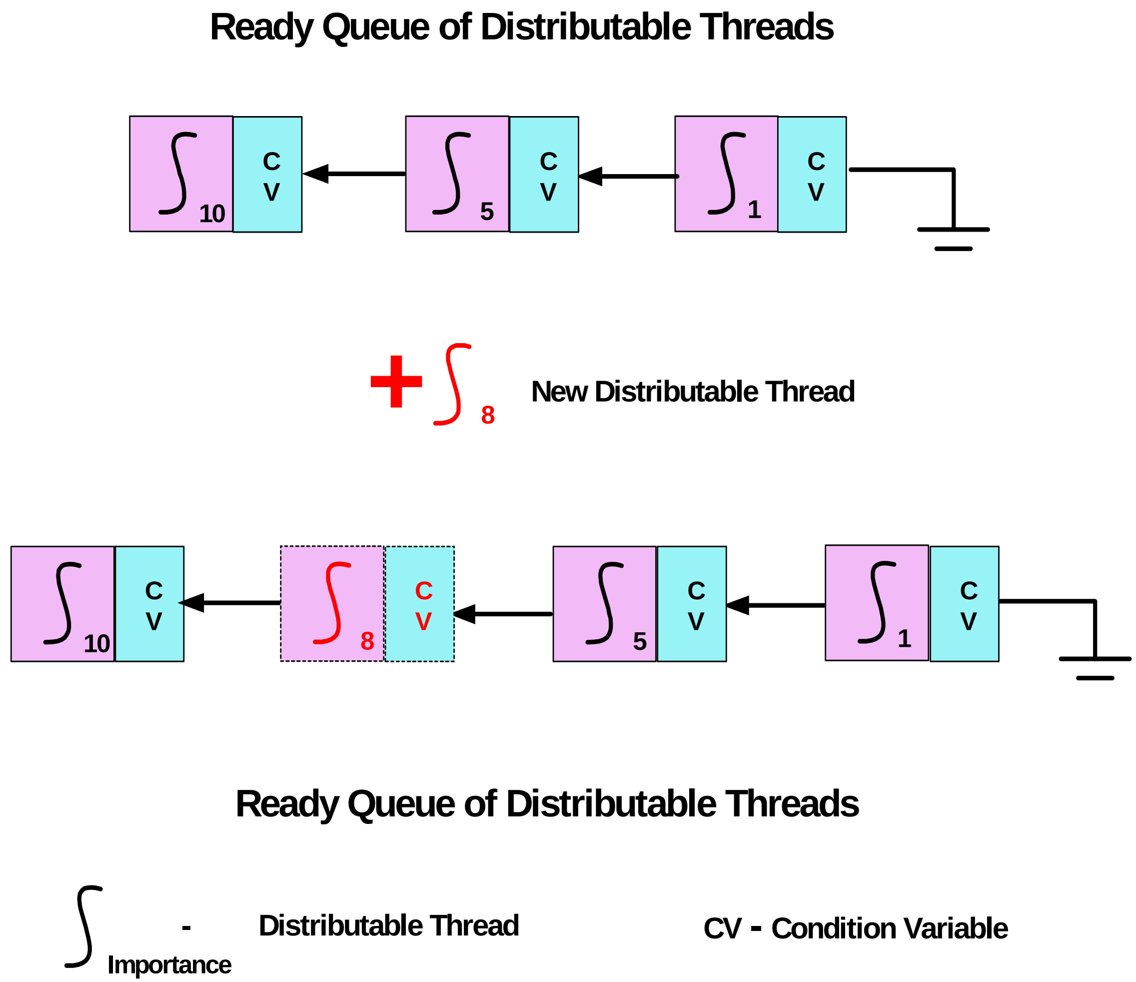
Distributable Thread Path Example

- 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



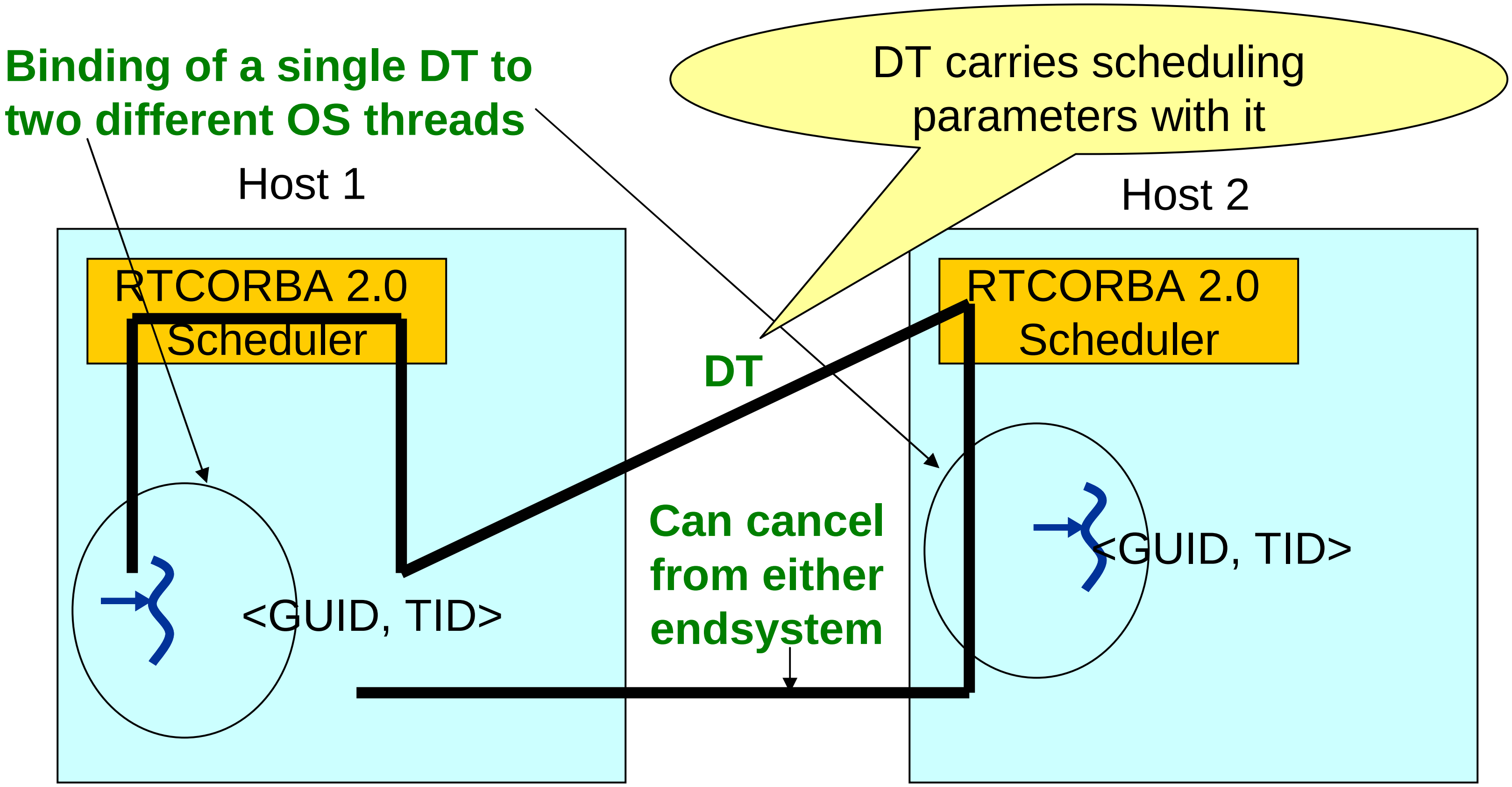
1. BSS - RTScheduling::Current::begin_scheduling_segment() or RTScheduling::Current::spawn()
2. USS - RTScheduling::Current::update_scheduling_segment()
3. ESS - RTScheduling::Current::end_scheduling_segment()
4. send_request() interceptor call
5. receive_request() interceptor call
6. send_reply() interceptor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

Middleware Based Scheduling



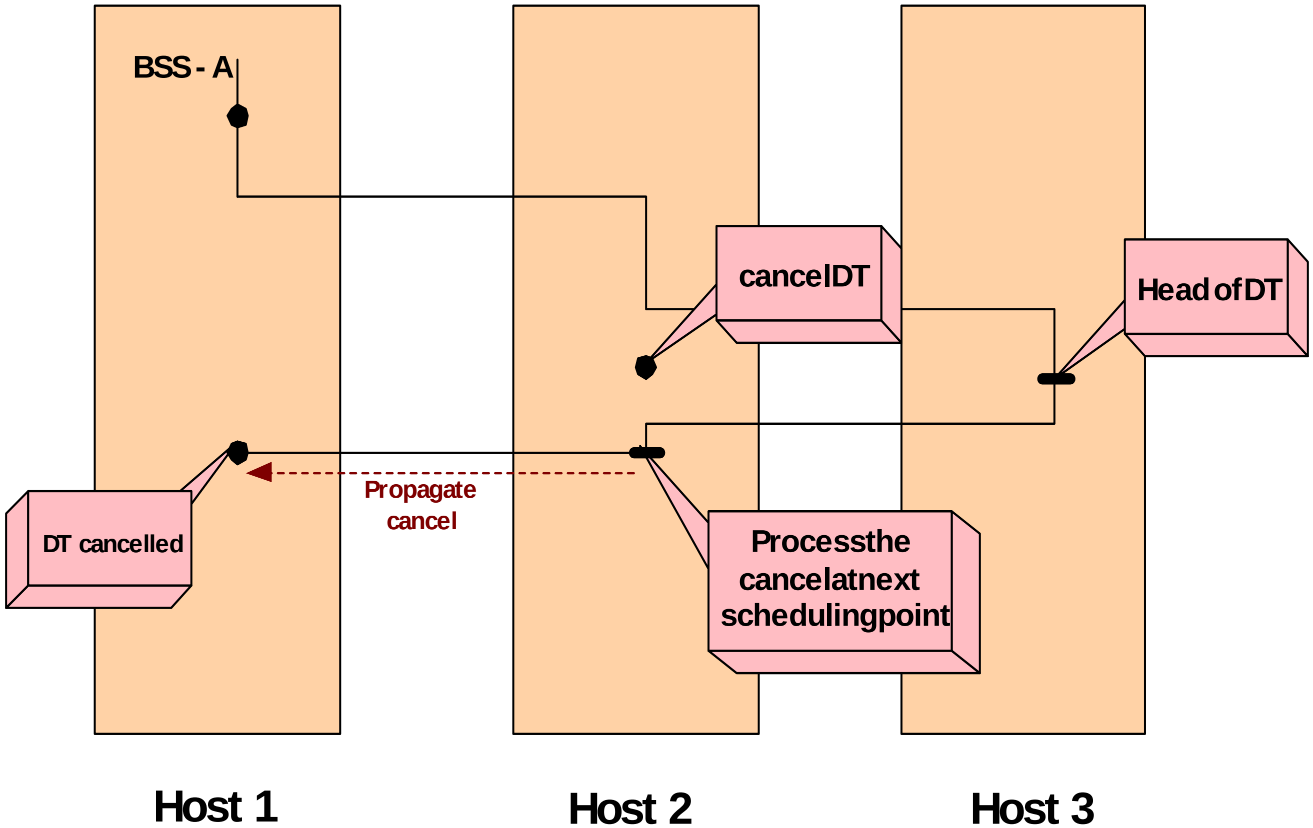
- Benefits: scalable in # of distributable threads per OS thread
- Drawbacks: queue management costs for some sched policies
- Alternatives: OS thread per distributable thread, lanes

Thread Cancellation and Identity Issues



- Other mechanisms affect real-time performance, too
  - Supporting safe, efficient cancellation of thread execution
  - Managing identities of distributable and OS threads
  - Configuring/using mechanisms sensitive to thread identity

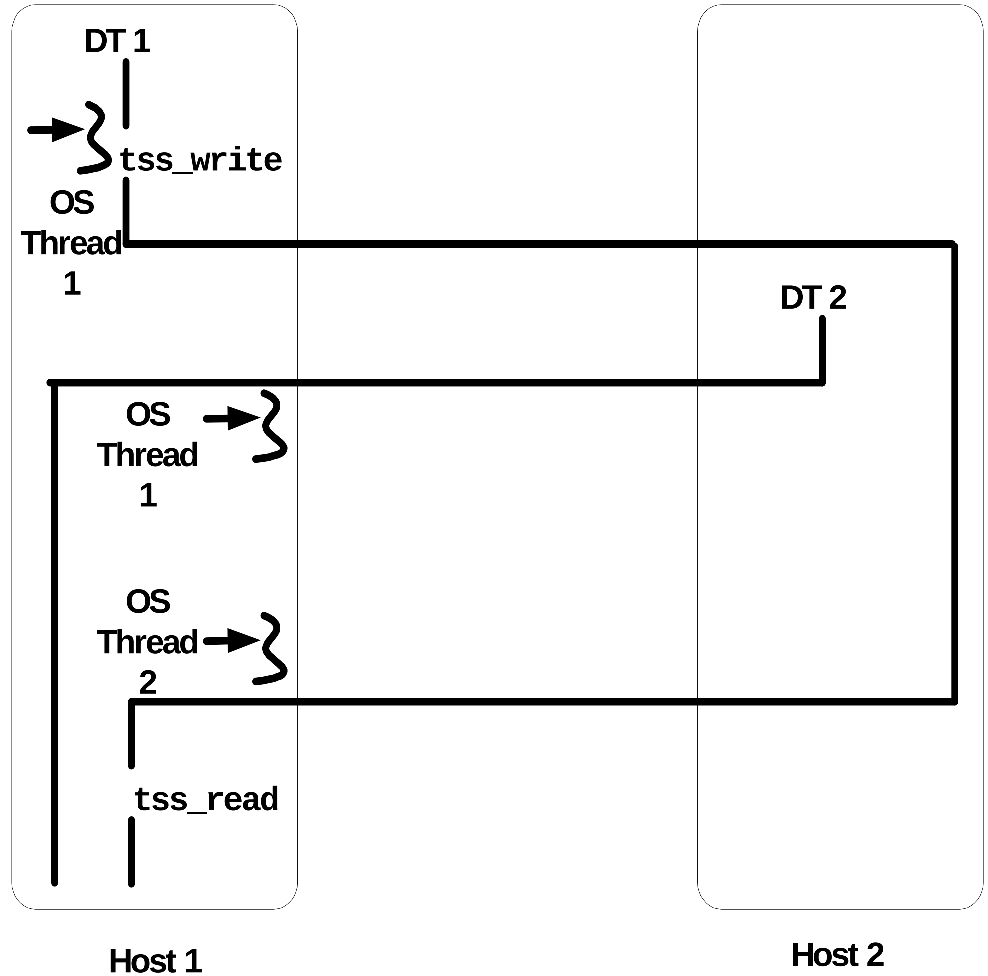
Distributable Thread Cancellation



- 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

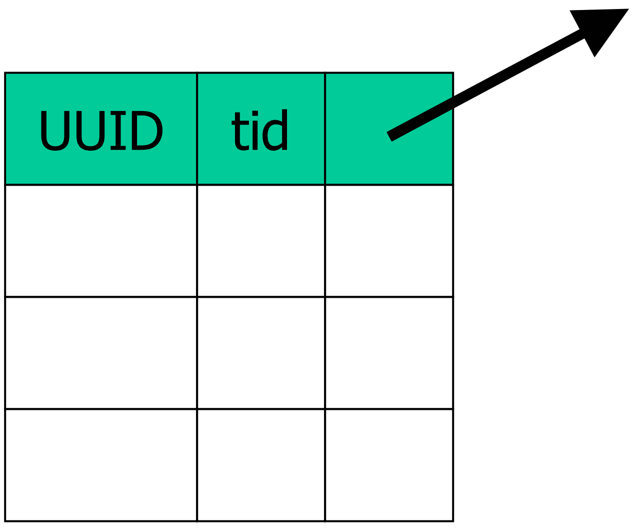
Thread Specific Storage Example

- Distributable-thread-specific storage
  - Avoids locking of global data, other benefits
- Context:
  - OS provides efficient TSS
  - TSS uses OS thread id
- Problem: distributable thread may span several OS threads at once
- Solution: "DTSS" emulation based on **`<UUID,tid>`**

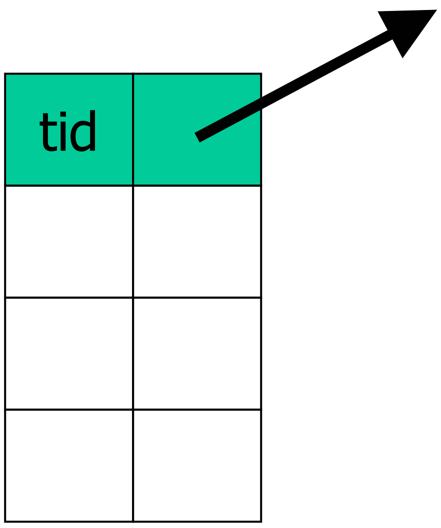


“DTSS” Emulation Overview

- Store void * pointers to objects in a hash table
- Use both distributable thread’s UUID and OS tid to index the hash table
- Allow “cursor” to collect all void *’s belonging to a distributable thread UUID
  - Across all OS tid’s
- Use TSS to store each tid’s current distributable thread UUID (or null)
- ORB and scheduler can set & get DTSS objects



DTSS



TSS

Concluding Remarks

- Dynamic Scheduling RT CORBA 1.2 (p.k.a. 2.0)
  - Offers flexible and predictable real-time performance for dynamic scheduling of distributable threads
  - A range of thread management mechanisms matter and must also be designed for real-time performance
- Current and Future Work
  - Refinement of GUID aware TSS, locking mechanisms
  - Empirical comparison of distributable threads to other end-to-end communication abstractions (method, event paths)
  - Integration of further scheduling strategies (eventually, progress and other utility based scheduling strategies)