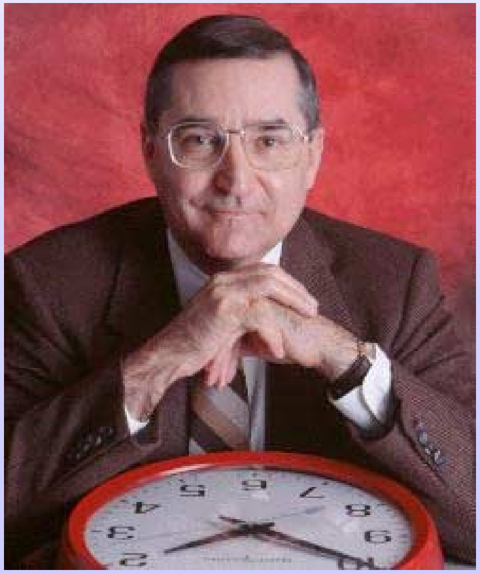
A Guided Tour of Real-time CORBA

Part 3a

E. Douglas Jensen
Chief Scientist, D500
MITRE, Bedford, MA
jensen@{real-time | mitre}.org
www.{real-time | mitre}.org



A Guided Tour of Real-time CORBA

Part 3a
Using RT-CORBA in Dynamically Scheduled Systems

Introduction to Distributable Threads

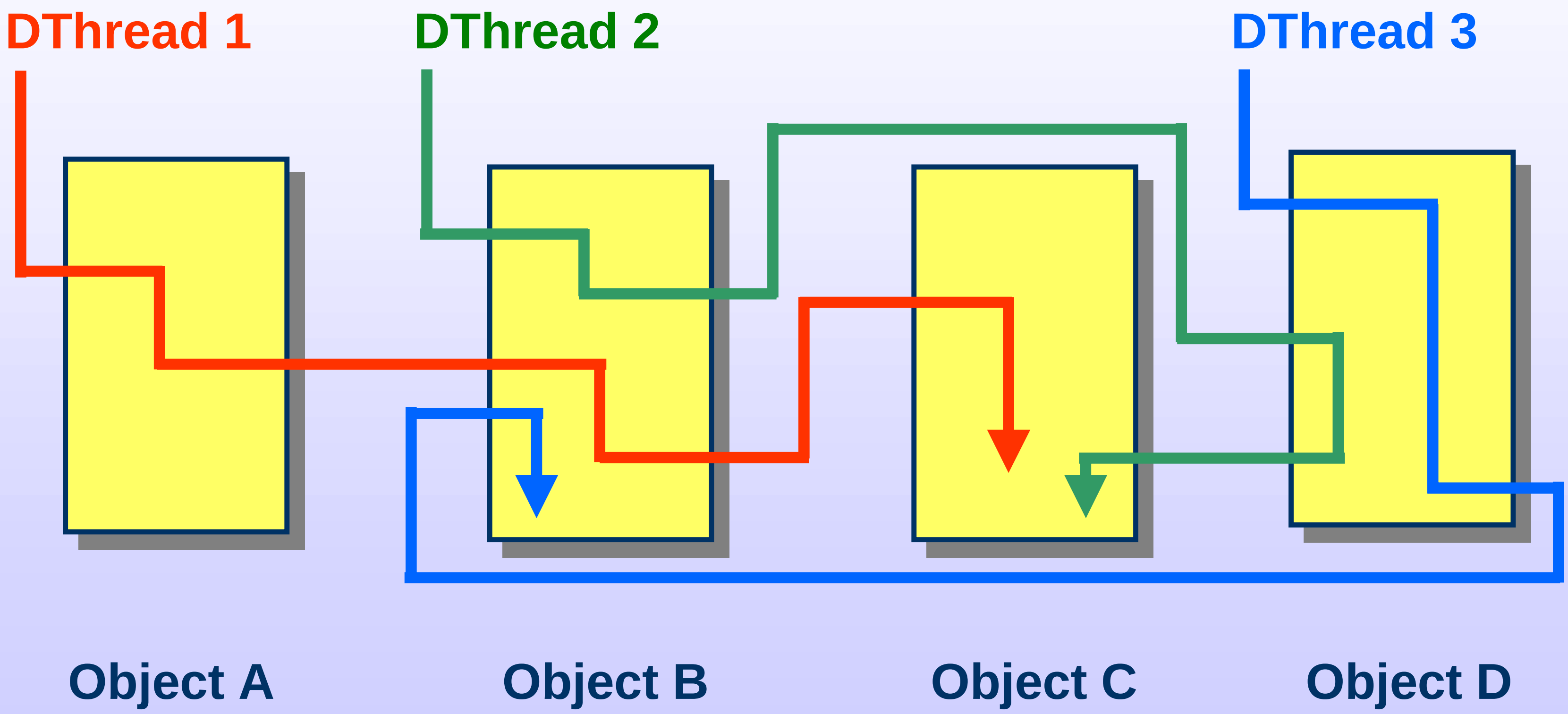
- This part of the presentation introduces the concepts and rationale for one of Real-Time CORBA 1.2's two major new features –

  Distributable Threads

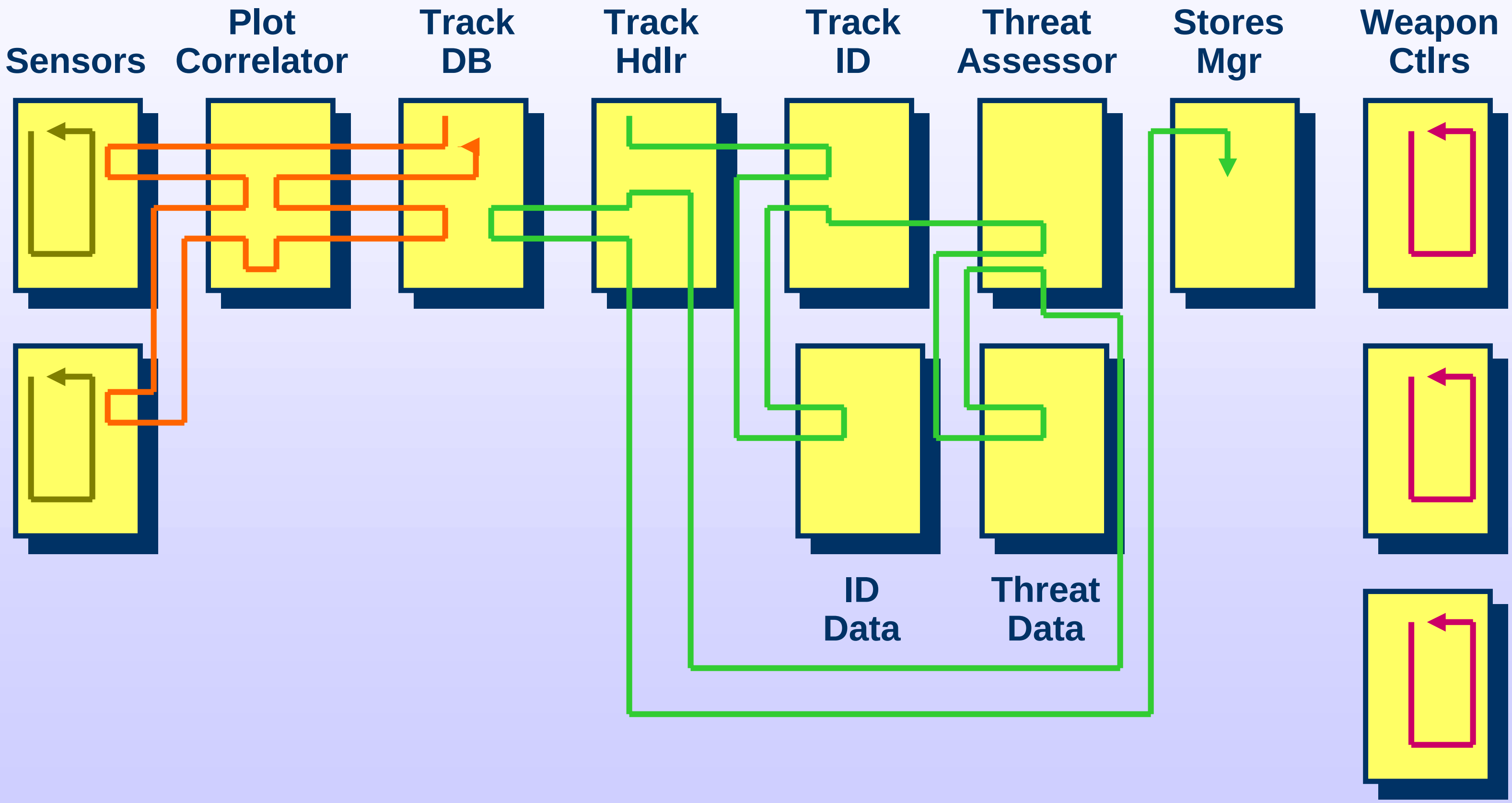
Real-Time CORBA 1.2 provides a programming model based on the *distributable thread*

- The distributable thread is a subset of the *distributed thread* in the Alpha distributed real-time OS kernel
- A *distributable thread* is an end-to-end control flow abstraction
- A distributable thread is a locus of control flow movement, within and among objects and nodes
- A distributable thread has a unique global ID
- A distributable thread executes a remote method, like a local one, directly itself –
by extending and retracting itself between objects and nodes
- The distributable thread is the schedulable entity
- A program may consist of multiple concurrent distributable threads

A *distributable thread* is an end-to-end control flow abstraction



An example of distributable threads in a battle management/C^2 system



From: ***An Example Real-Time Command, Control and Battle Management Application for Alpha (1988)***

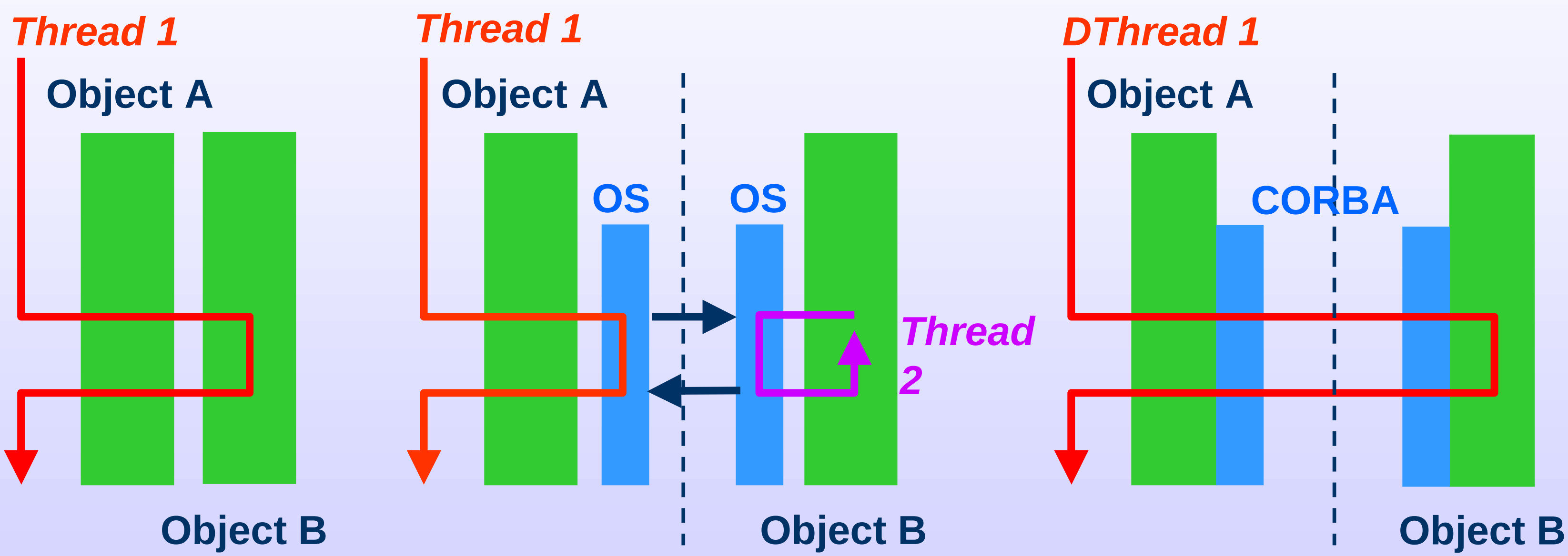
Concurrency is at the distributable thread level

- A distributable thread always has exactly one execution point (*head*) in the whole system
  - control flow can be forked by creating or awakening other distributable threads
- Multiple distributable threads execute concurrently and asynchronously, by default
- distributable threads synchronize through method execution
  - object writers control distributed thread concurrency – e.g.,
    - monitor (no concurrency)
    - re-entrant
    - recursive

Conventional distributed object models don't retain local semantics on remote invocations

- For a local invocation in most distributed object systems
  - there is only one thread
  - it retains its identity and properties (e.g., timeliness) whichever method it executes
- Conventional remote method invocation (and RPC) involve
  - separate schedulable entities on each node (client, servant)
  - which communicate with each other
- That
  - doesn't accurately reflect the programmer's intention of control flow spanning objects, and perhaps physical nodes
  - impedes maintaining end-to-end properties

A distributable thread has location-independent method invocations



local invocation

conventional remote invocation

distributable thread

A distributable thread is sequential rather than synchronous

- The synchrony of a conventional method invocation (or RPC) is often cited as a concurrency limitation
- But a distributable thread is a sequential entity like a local thread
- A distributable thread is always executing somewhere, while it is the most eligible there
    - it is not doing “send/wait’s” as with conventional method invocations (or RPC’s)
- Remote invocations and returns are scheduling events at both source and destination nodes
    - each node’s processor is always executing the most eligible distributable thread there
    - the other distributable threads there wait as they should
- Local method invocations/returns benefit from not requiring context switches like threads normally do

Multi-node application activities have end-to-end properties

- Distributed systems have requirements for end-to-end properties of their collective multi-mode activities – e.g.,
  - timeliness
  - reliability/availability
  - security
  - transactional context
  - resource ownership
  - dependencies
  - etc.
- For control flow programming models, such as in CORBA, these are end-to-end *trans-node* properties

A real-time distributed system must have acceptable timeliness of multi-node application activities

- The defining characteristic of any real-time distributed computing system, whatever its programming model, is that
  - the collective timeliness
    - optimality
    - predictability of optimality

    of multi-node application activities
  - is acceptable to that application
  - under the current circumstances

A distributable thread has end-to-end timeliness attributes

- Each distributable thread has execution scheduling attributes – e.g.,
  - time constraints (deadlines, time/utility functions)
  - relative importance, expected duration, precedence, etc.
- These specify the end-to-end timeliness for it completing the sequential execution of methods in object instances that may reside on multiple physical nodes
- Execution of the distributable thread is governed by those scheduling parameters, according to the scheduling discipline, regardless of the distributable thread's execution point transiting nodes
- The goal is to provide acceptable (as defined by the application) end-to-end timeliness of collective distributable thread execution

A multi-node activity's timeliness properties must be used consistently on all involved nodes

- In most cases, the fundamental requirement for achieving acceptable end-to-end timeliness is that

  a multi-node application activity's timeliness properties

  - time constraints
  - expected execution time
  - execution time received thus far
  - etc.

  be explicitly employed for managing (scheduling, etc.) resources

  - hardware (e.g., processor, i/o)
  - software (e.g., synchronizers) resources

  consistently on each node involved in that application multi-node activity

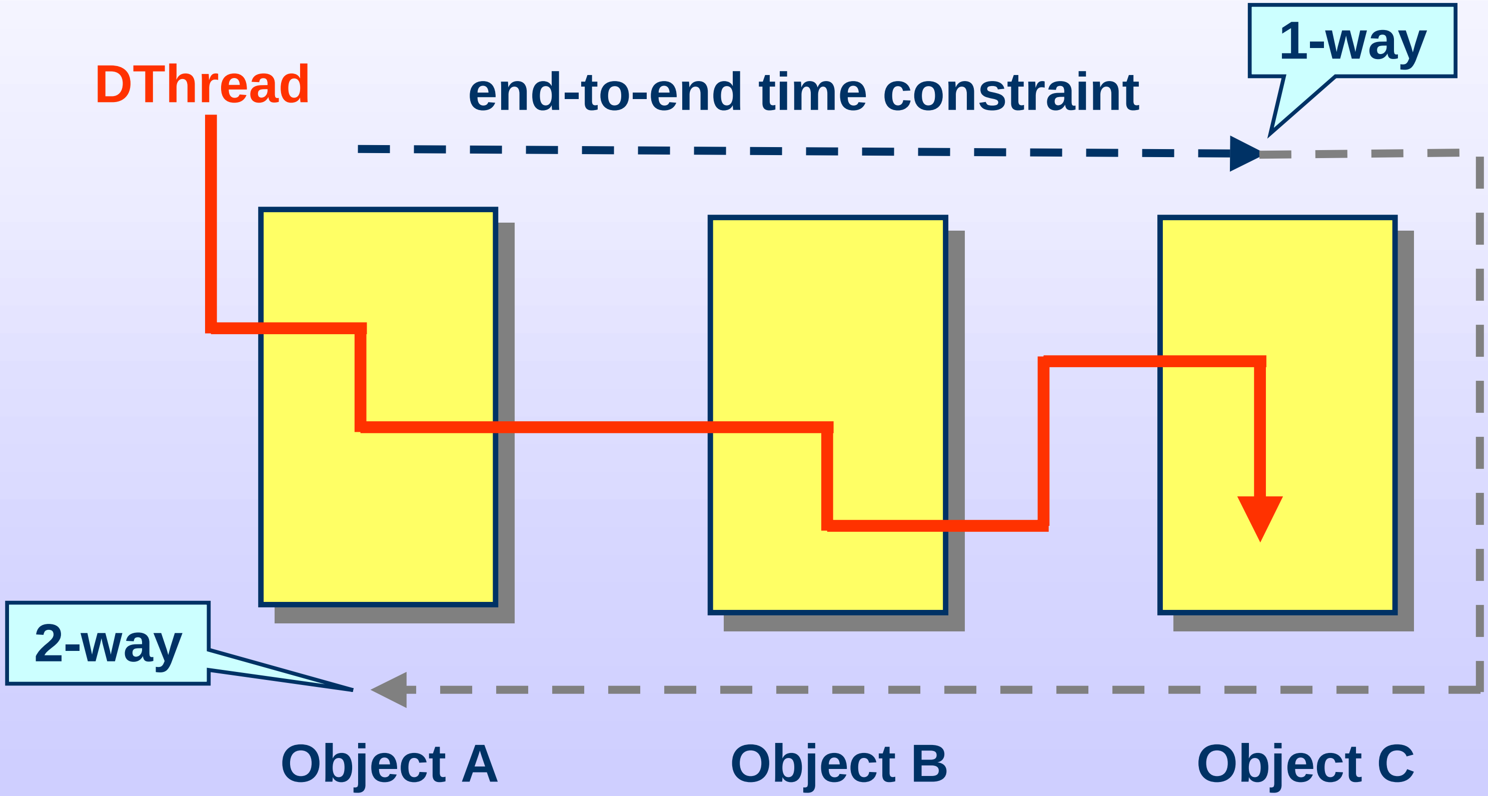
Multi-node application activity timeliness properties must be propagated among nodes

- Thus, in dynamic real-time distributed systems, these properties must be propagated among corresponding computing node resource managers in
  - operating systems
  - Java virtual machines (JVM's)
  - middleware
  - etc.
- In static real-time distributed systems, these properties can be instantiated á priori

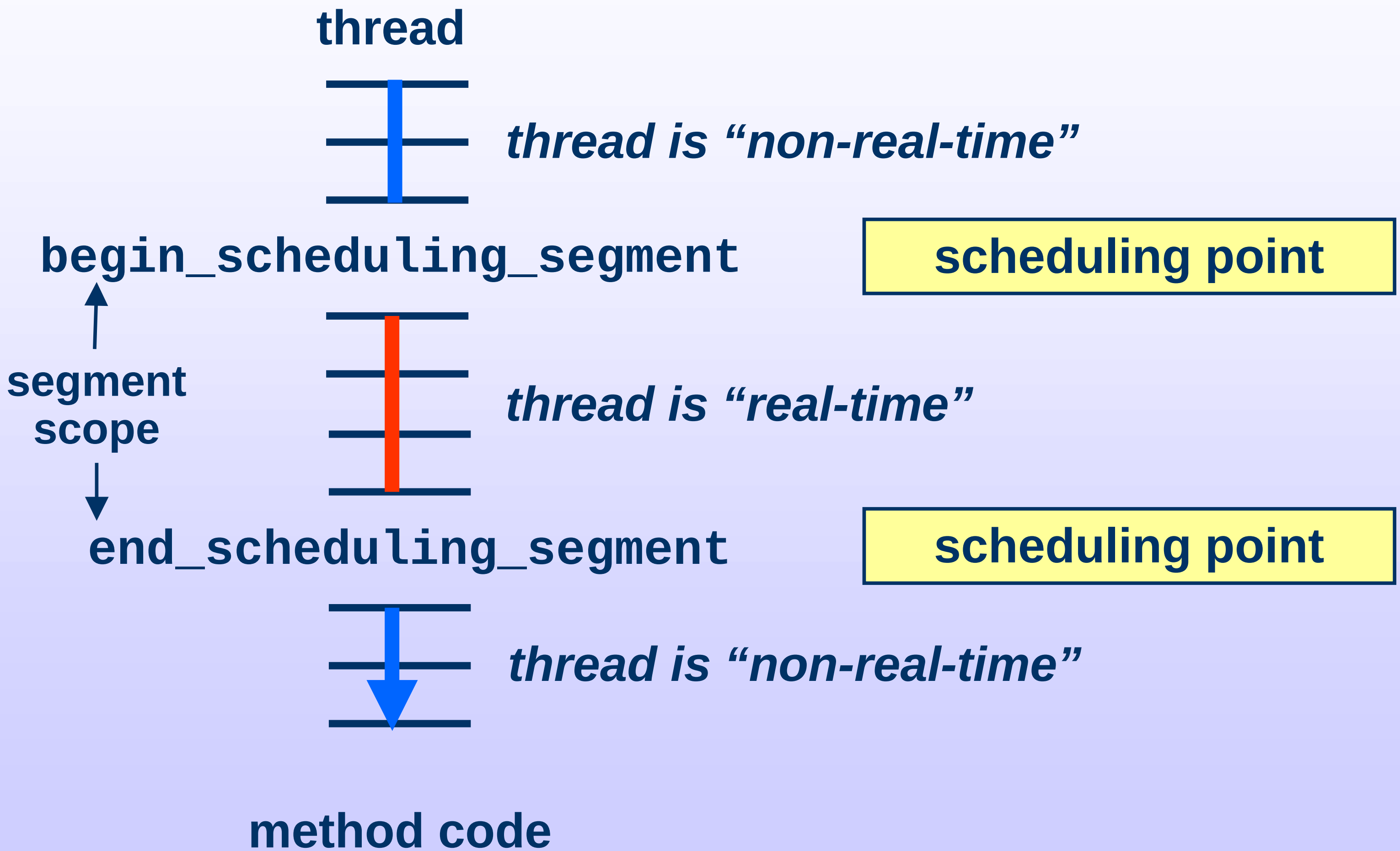
The distributable thread abstraction propagates computational context end-to-end

- When a distributable thread transits a node boundary, its timeliness parameters are propagated to the remote scheduling discipline instance
  - in Real-Time CORBA 1.2, via a GIOP service context
- When/if it returns, updated timeliness parameters are propagated back to the invoker’s scheduling discipline instance
- Other end-to-end properties may also be propagated – e.g., resource ownership, dependencies, rights, security, transactional context
  - not part of the Real-Time CORBA 1.2 specification
- This should be transparent to the application programmer

A distributable thread supports end-to-end properties such as timeliness

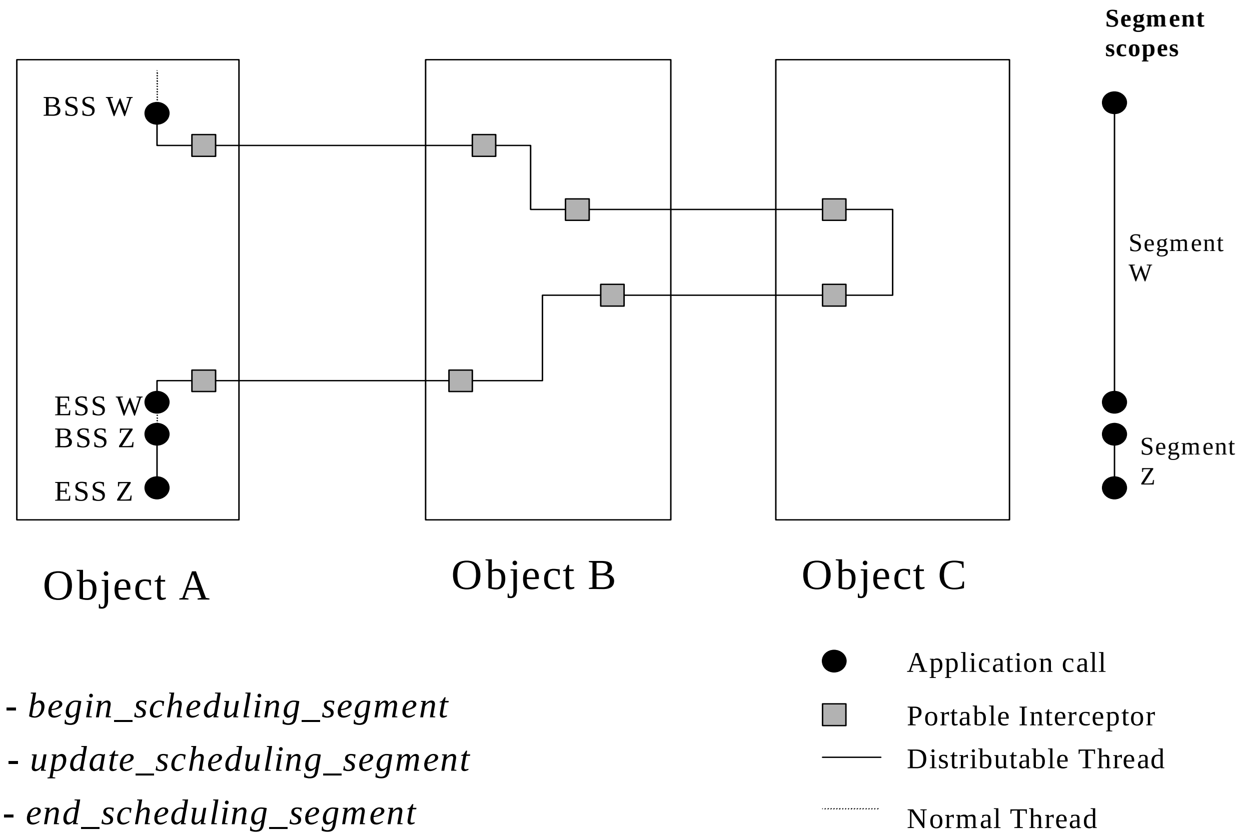


A distributable thread has one or more *scheduling segments* with corresponding time constraints



The distributable thread as it appears in the Real-Time CORBA 1.2 specification

Distributable Thread Traversing CORBA Objects



BSS - *begin_scheduling_segment*

USS - *update_scheduling_segment*

ESS - *end_scheduling_segment*

There are several ways to use timeliness properties for scheduling consistently on each node

- Timeliness properties can be propagated, and used consistently by node resource managers
  - e.g., every node schedules using the same discipline
  - this can provide some approximate global optimality
- Timeliness properties can be propagated, and used by a logically singular global scheduler that is instantiated on every node
  - the schedulers interact to schedule all the nodes
  - global optimality is formally impossible in general, but may be better approximated in many cases
- One or more levels of “meta” resource managers above the node resource managers function according to either of the above two cases
- End-to-end time constraints must be decomposed per node

Real-Time CORBA 1.2 distributable thread is a subset of the distributed thread abstraction

- The CORBA distributable thread abstraction is based on the Alpha *distributed thread* abstraction
- The current Real-Time CORBA 1.2 distributable thread abstraction does not yet have all the distributed thread features
- Initially (at least) these missing features will be
  - optional
  - application-specific
  - vendor value-added

  (just as has always been the case for CORBA per se)
- Future OMG RFP's are expected to incrementally add features based on implementation and user experience

A completely defined distributed thread abstraction must include other features

- Distributed event handling
  - events of interest to a distributed thread (i.e., changes in predicated system state) are delivered to its head
  - and perhaps from its head back up the invocation chain
    - Real-Time CORBA 1.2 does this
- The distributed thread's execution point can be controlled – paused, aborted, resumed, etc.
- Distributed handling of partial (node, network) failures – e.g.,
  - maintaining distributed thread integrity
  - orphan detection and termination/continuation
- Distributed thread concurrency control
- These all must be performed in accordance with the application's time constraints

The distributed thread abstraction also has implementation advantages

- The distributed thread abstraction automatically supports
  - resource limit and consumption tracking
  - server thread management
- Each object no longer has the burden of managing its own pool of threads and related resources (stacks, etc.)
- This minimizes the tendency to do pessimistic resource management strategies
- For these reasons, the distributed thread abstraction has been widely adopted for microkernel-based OS's, independent of the OS's programming model