Activity:
An End-to-End Abstraction for Real-Time CORBA

E. Douglas Jensen

jensen@real-time.org

http://www.real-time.org

OMG Workshop on Real-Time and Embedded CORBA

Revised 7/16/2000 11:56 PM

MITRE

Abstract

- The OMG Real-Time CORBA 1.0 Specification used the "activity" abstraction as an analysis/design concept for end-to-end predictability of timeliness, but it left the details of the abstraction undefined
- This presentation outlines a complete definition of the activity abstraction
- This definition of the abstraction is expected to be explicitly supported in
  - the next revision of the Joint Proposal for Dynamic Real-Time CORBA
  - the Distributed Real-Time Specification for Java
- Very similar versions of this abstraction have been successfully implemented in several other real-time operating system and middleware contexts, and employed in a number of experimental applications

Introduction

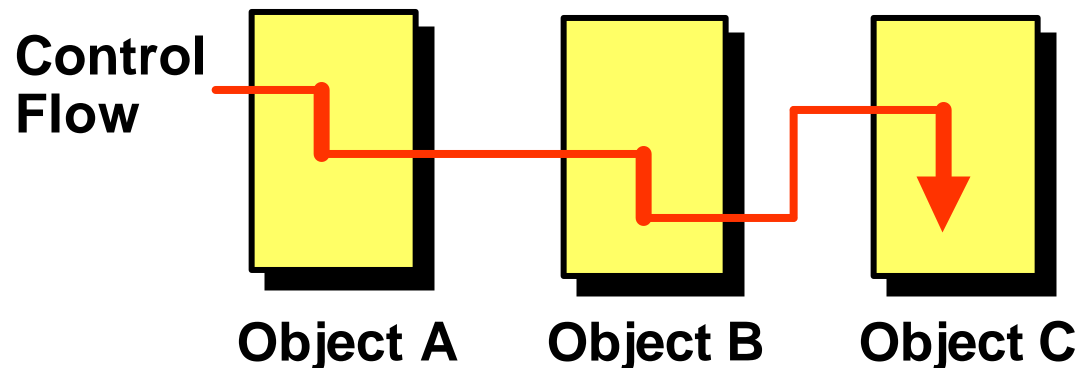
- Many distributed systems consist of inter-operating centralized systems (cf. “enterprise application integration”)
- Many others seek to be logically singular systems that must be physically dispersed – hence they have
  - software activities that are each distributed (trans-node)
  - data that is distributed (partitioned, replicated)
- Real-time distributed systems have end-to-end timeliness requirements – trans-node activities which must complete
  - at acceptable times
  - with acceptable predictability
- Developing real-time distributed software is very difficult

There are many different ways that distributed systems could be taxonomized

- For our purposes here, a very simple way is sufficient: according to their programming model for interaction
  - networked – i.e., asynchronous message-passing
  - control flow – e.g., RPC, method invocation
  - data flow – e.g., publish/subscribe
  - blackboards/spaces – e.g., Linda, JavaSpaces
  - mobile objects – e.g., Voyager
  - autonomous agents
  - autonomous decentralized systems
- Of course, some form of communication facility underlies each programming model – e.g.,
  - method invocation
  - on RPC
  - on message-passing

Our focus here is on control flow programming models in distributed object systems

- Well known technologies for such systems include CORBA and DCOM/COM+
- Such systems allow writing programs whose components may be spread across multiple computing nodes
- Those programs are *trans-node*
  - one or more causally ordered sequences of constituent operations occurring on multiple nodes
  - each via a sequence of method invocations/returns



Control flow has compelling advantages as a native model for distributed real-time software

- Distributed control flow is a natural, well-understood, incremental extension to local control flow (procedure calls)
  - distribution adds complications (partial failures, concurrency control, network latencies, etc.)
- Data flow per se doesn't include resource management to effectively get end-to-end properties, such as timeliness
- Data-flow abstractions can be built cost-effectively using control flow abstractions
  - the converse – e.g., method invocations based on TCP/IP streams – is usually semantically difficult
- Control flow support for predictability of end-to-end timeliness can be used with any distributed system programming model

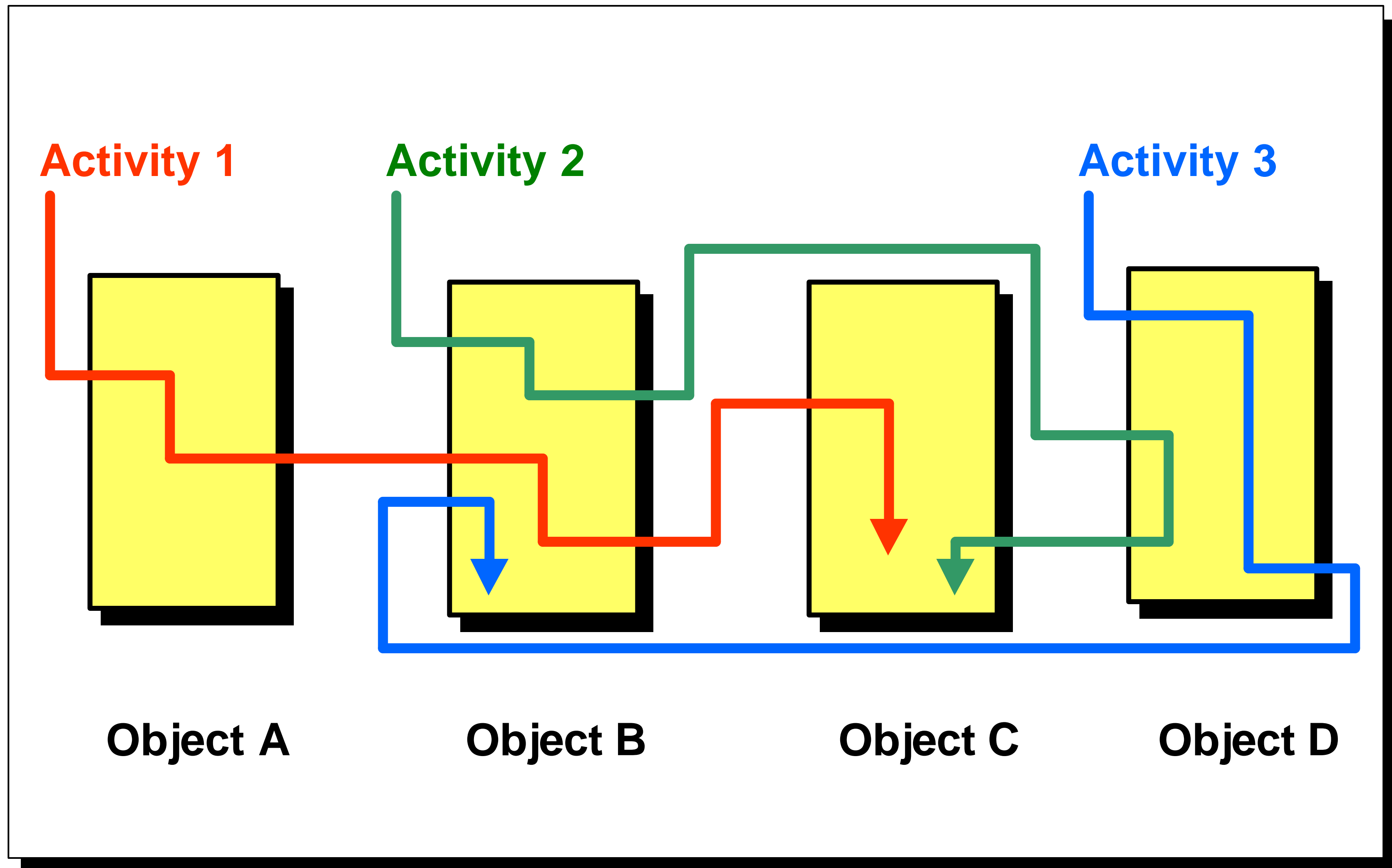
Control flow model method invocations are location-independent, and other code is not

- All the code in a distributed object program is not usually expected to be location-independent
- That would be very difficult, due to
  - network latencies
  - partial failures
  - synchronization
  - concurrency control
  - etc.
- The primary benefits of distributed object programming can be obtained by a model having
  - location-independent invocations and returns
  - location-dependent (node-local) code otherwise

An *activity* is an end-to-end control flow abstraction

- A control flow distributed object program can be thought of in terms of an end-to-end abstraction we'll call an *activity*
- An activity is a logically distinct and identifiable locus of control flow movement, within and among objects and nodes
- An activity executes a remote method, like a local one, directly itself –
  by extending and retracting itself between objects and (transparently) nodes
- It has the same semantics for remote invocations as for local ones
- The activity is the schedulable entity
- A program may consist of multiple concurrent activities

An *activity* is an end-to-end control flow abstraction



Concurrency is at the activity level

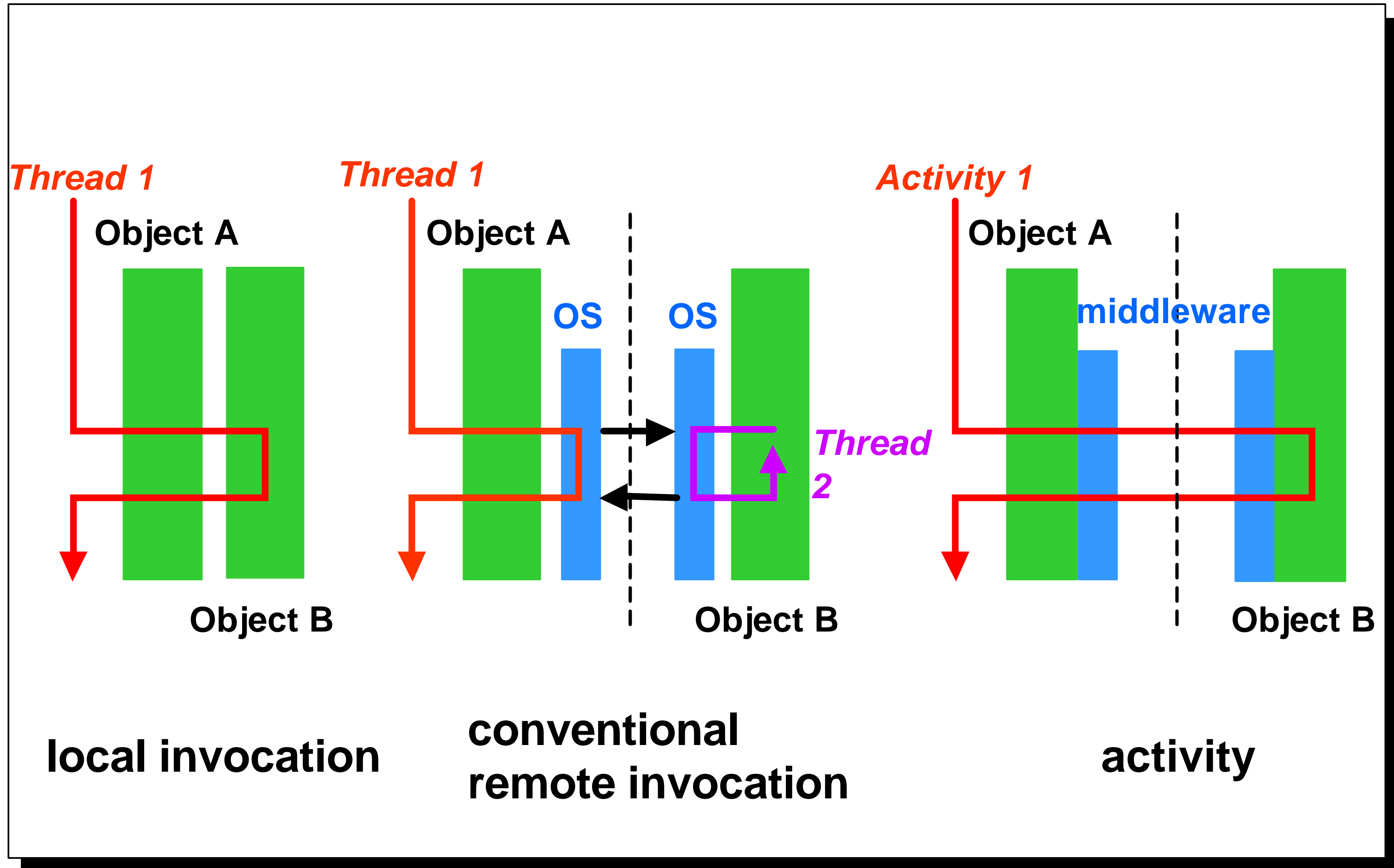
- An activity always has exactly one execution point (*head*) in the whole system
  - new activities can be created or awakened when needed
- Multiple activities execute concurrently and asynchronously, by default
- Activities synchronize through method execution
  - object writers control activity concurrency

MITRE

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

- 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 thus physical nodes, in a location-independent way
  - impedes maintaining end-to-end properties
- Whereas for a local invocation
  - there is only one thread
  - it retains its identity and properties (e.g., timeliness) whichever method it executes

An activity has location-independent method invocations



MITRE

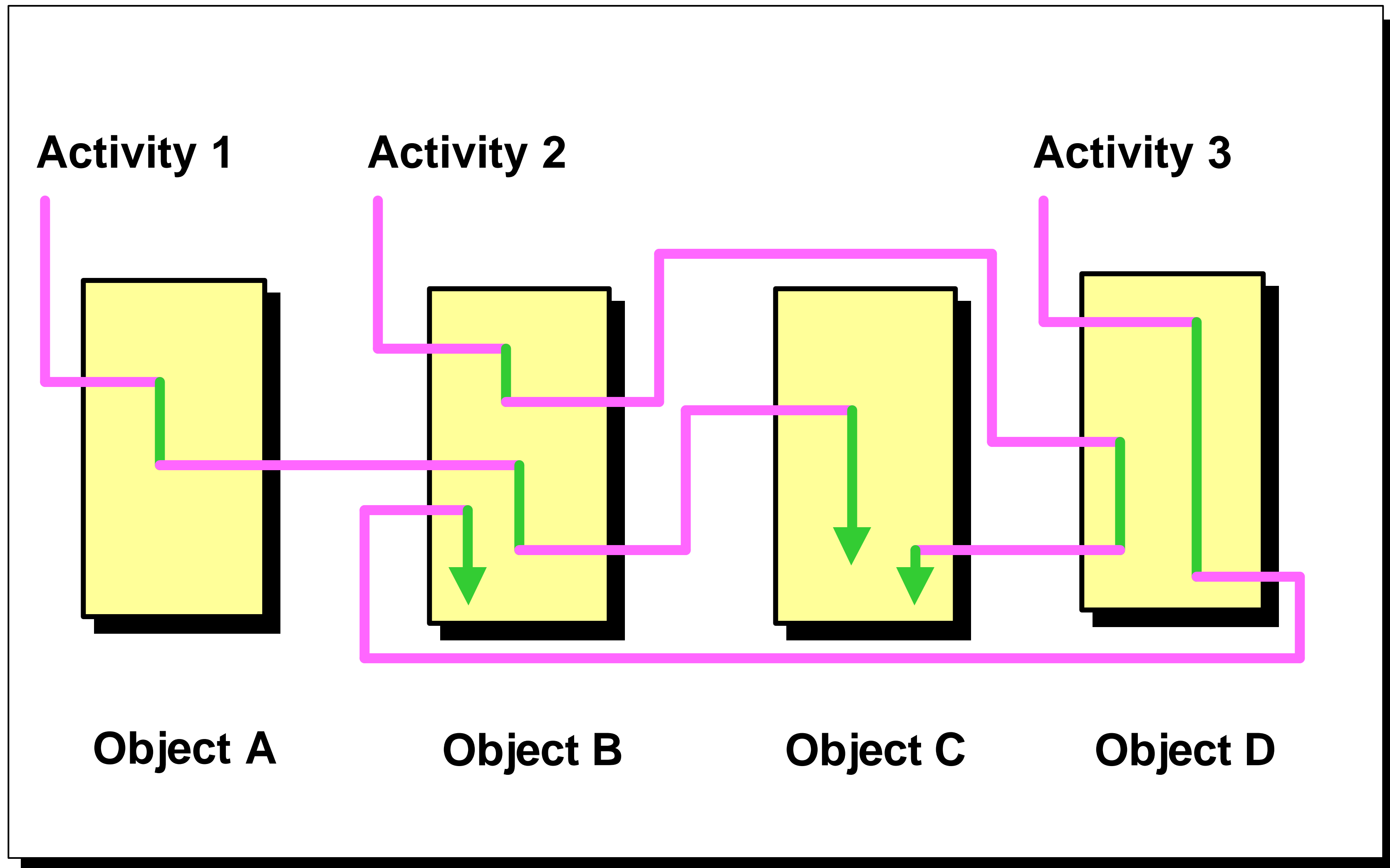
An activity is sequential rather than synchronous

- The synchrony of a conventional method invocation (or RPC) is often cited as a concurrency limitation
- But an activity is a sequential model like a local thread
- An activity 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 activity there
  - the other activities there wait as they should
- Local method invocations/returns benefit from not requiring context switches like threads normally do

An activity is built using local (e.g., RTOS) threads and method invocations

- The activity abstraction is implemented using local (RTOS or JVM) threads
- The activity abstraction can be implemented
  - as part of the operating system – e.g.,
    - Alpha
    - MK7.3a
  - as part of the middleware – e.g.,
    - Libra
    - ActiveRT (real-time DCOM)
    - Dynamic Scheduling Real-Time CORBA (proposed)
  - as part of a programming language (e.g., Java)
    - semantics
    - run-time

An activity is built using local (e.g., RTOS) threads and method invocations



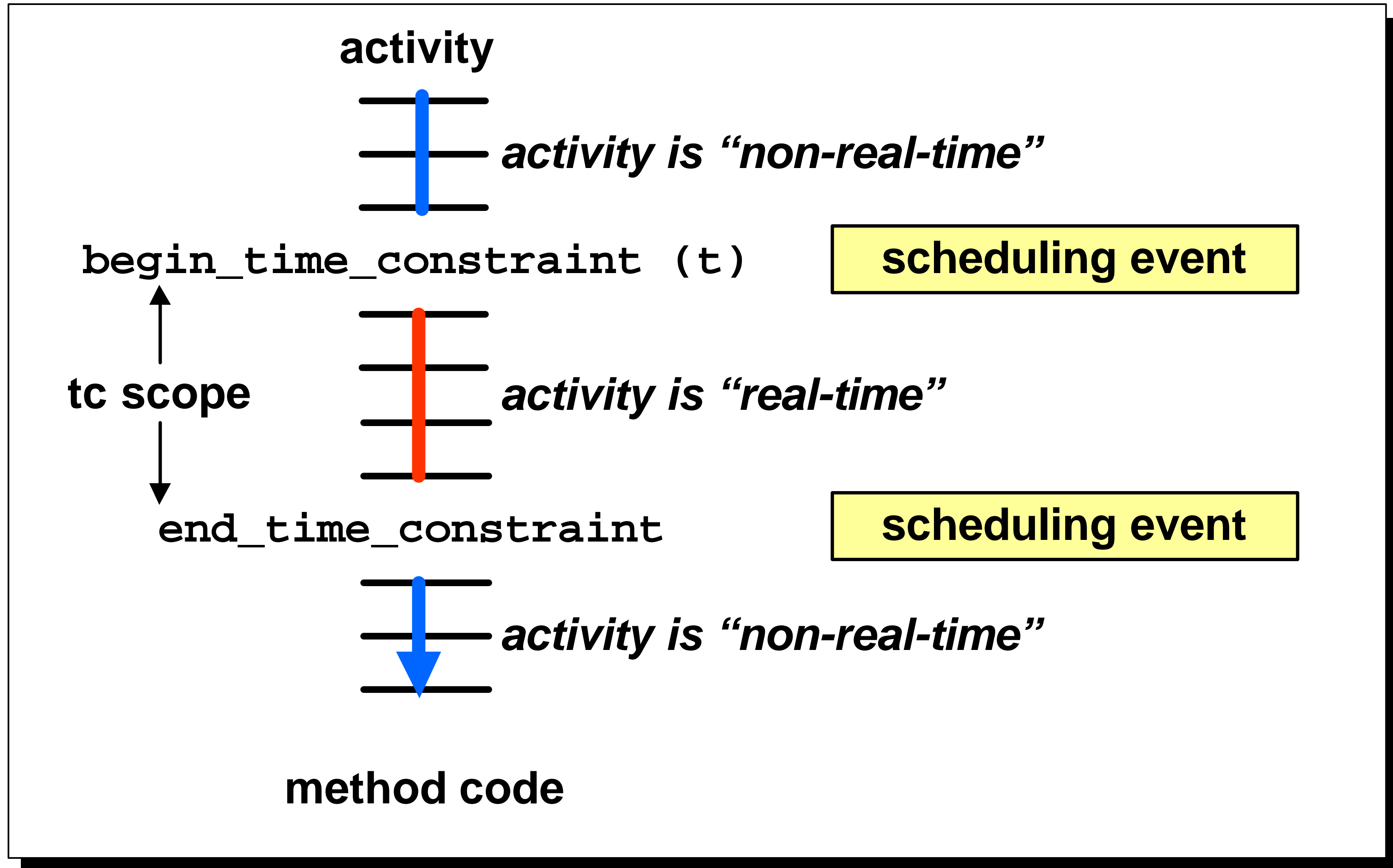
An activity has end-to-end timeliness attributes

- Each activity may have execution scheduling attributes – e.g.,
  - time constraints
  - relative importance
- 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 activity is governed by those scheduling parameters, according to the scheduling policy, regardless of the activity's execution point transiting nodes
- The goal is to provide acceptably predictable (as defined by the application) end-to-end timeliness of collective activity execution

A time constraint is a lexically scoped attribute of an activity

- The logic of a real-time application includes actions whose completions are time-constrained
- The most common example of a completion time constraint is a deadline (but there are many others)
- The action is performed by an activity executing code that has a time-constrained region called the *scope* of the time constraint
- While executing in a time constraint scope, an activity is a "real-time" one, and otherwise it is a "non-real-time" one
- The passage of an activity's execution point into or out of a time constraint scope is a *scheduling event*
  - a decision must be made about which activity should be executing

A time constraint is a lexically scoped attribute of an activity



A property is predictable to the degree that it is known in advance

- Predictability is a continuum
- One end point of the predictability scale is *determinism*, in the sense that the property is known exactly in advance
- The other end point is maximum entropy, in the sense that nothing at all is known in advance about the property
- In stochastic systems (which include hard real-time ones as a special case), one way to measure predictability is coefficient of variation C_n = variance/mean^2
  - the maximum predictability end point is the deterministic distribution, whose $C_n = 0$
  - at the minimum end point is the extreme mixture of exponentials distribution, whose C_n = ¥

MITRE

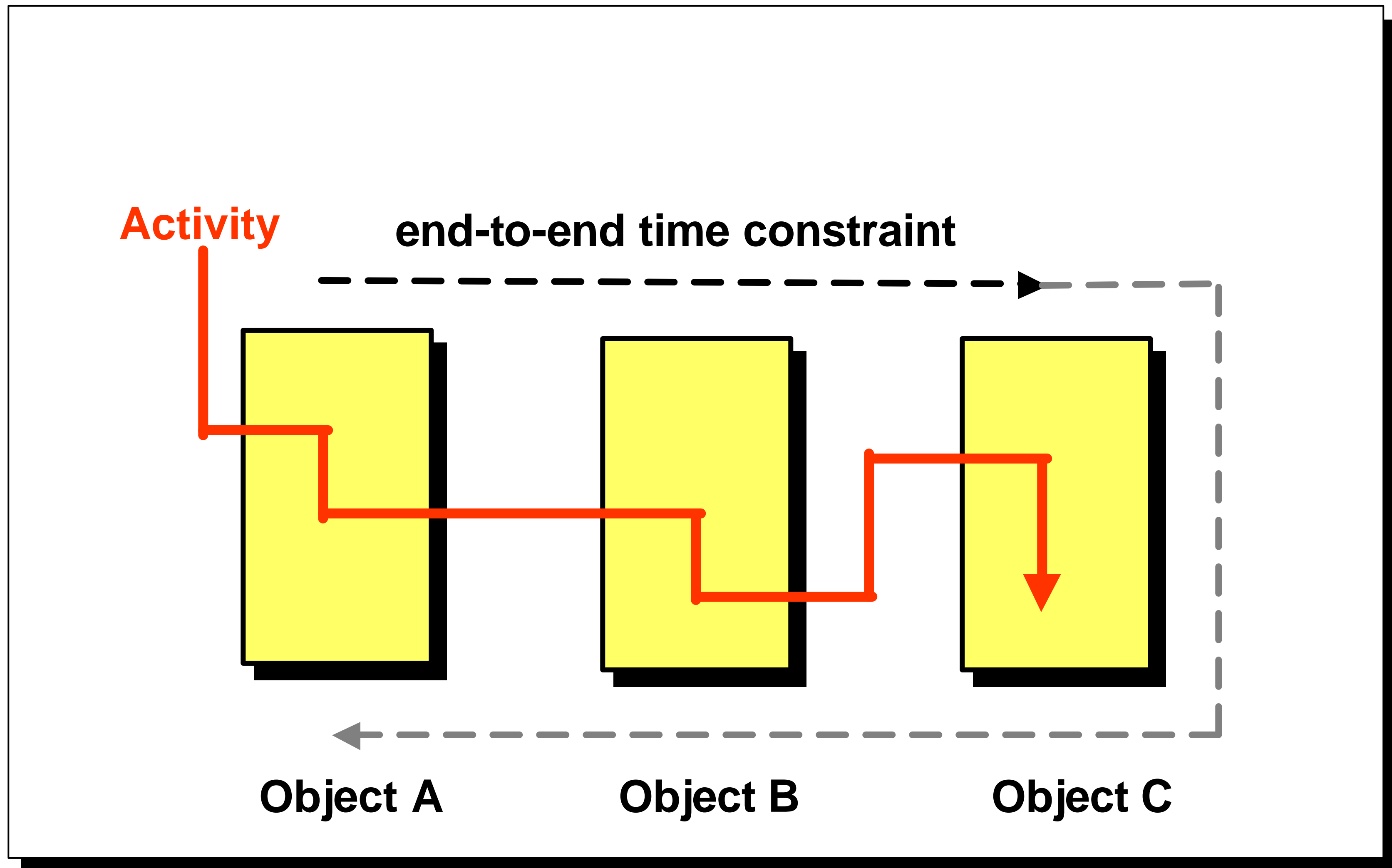
Real-time middleware provides predictability of end-to-end timeliness

- Real-time distributed programs have end-to-end timeliness requirements
  - i.e., timeliness, and predictability of timeliness, requirements for each entire sequence of trans-node operations
- Today's COTS RTOS's and JVM's don't support predictability of end-to-end timeliness, so real-time middleware implementations are required
- Real-time middleware is not supposed to attempt to violate the laws of physics – it is supposed to
  - first, obey the Hippocratic oath: don't degrade the predictability of the underlying OS's and network
  - then, provide the best predictability possible given that of the underlying OS's and network

The activity abstraction provides a vehicle for propagating computational context end-to-end

- In distributed systems, shared computation context must be explicitly propagated end-to-end among nodes
- An activity's timeliness properties govern its execution eligibility at every node it visits
- When it transits a node boundary, its timeliness parameters are propagated to the remote scheduling policy instance
  - in the OS, JVM, or middleware
- When it returns, updated timeliness parameters are propagated back to invoker's scheduling policy instance
- (Other end-to-end properties may also be propagated – e.g., ID, resource ownership, dependencies, rights, security, transactional context)

An activity supports end-to-end properties such as timeliness



MITRE

The activity abstraction is applicable to the whole predictability/time-frame space of real-time systems

- The activity approach to control-flow style distributed programming is applicable to real-time systems which are
  - hard (i.e., always meet all hard deadlines)
  - or anywhere else on the predictability continuum
- That continuum is orthogonal to application time-frame magnitudes,

  which range in practice from microseconds to megaseconds
- The activity abstraction supports application timeliness requirements everywhere in that two-dimensional predictability/time-frame space of distributed real-time systems

MITRE

The activity abstraction also has implementation advantages

- The activity abstraction automatically provides
  - 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
- The activity abstraction has been widely adopted for microkernel-based OS's for these implementation advantages, independent of the programming model

A completely defined activity abstraction must include other essential features

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

Activity abstractions have been successfully implemented in experimental systems

- Activity abstractions have been implemented in
  - real-time distributed operating systems
    - CMU et al.'s Alpha
    - OSF/RI's MK7.3A
  - real-time middleware
    - Digital/Compaq's real-time DCOM
    - MITRE
- Experimental applications have been prototyped with advantageous results
  - coastal air defense battle management
  - AWACS surveillance tracking system

An activity abstraction is the basis for Sun's Distributed Real-Time Specification for Java

- The Distributed Real-Time Specification for Java is being developed under Sun's Java Community Process
- JSR-0*50 proposes that at least the initial distributed real-time Java platform provide a native control flow mechanism that supports predictability of end-to-end timeliness for
  - activity-based programming models
  - other distributed programming models

Potential future OMG RFP's could include a fully elaborated activity abstraction

- A basic activity abstraction is expected to be included in the current proposal for Dynamic Scheduling Real-Time CORBA
  - it is not expected to have all the necessary features
  - initially (at least) these features will be
    - optional
    - application-specific
    - vendor value-added
- Future OMG RFP's should incrementally add features based on experience

MITRE