

The Distributed Real-Time Specification for Java

Presented by: Douglas M. Wells
The Open Group
d.wells@opengroup.org

Original Briefing by: E. Douglas Jensen
The MITRE Corporation
jensen@real-time.org
<http://www.real-time.org>

Revised April 10, 2001

Outline

- ❑ **Java and “real-time Java”**
- ❑ **JSR-50 the Distributed Real-Time Specification for Java**
- ❑ **Distributed Control Flow**
- ❑ **Distribution Design Choices**
- ❑ **Conclusion**

The raison d'être of real-time computing is predictability of collective thread timeliness

- ❑ Informally, 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 $C_v = \text{variance}/\text{mean}^2$
 - the maximum predictability end point is the deterministic distribution, whose $C_v = 0$
 - at the minimum end point is the extreme mixture of exponentials distribution, whose $C_v = \infty$

Timeliness predictability generally must be traded off against other properties

- ❑ Predictability of timeliness generally just be traded off against
 - other real-time timeliness properties, particularly optimality of timeliness – e.g.,
 - better number of missed deadlines, mean tardiness, etc., but worse predictability of those
 - worse number of missed deadlines, mean tardiness, etc., but better predictability of those
 - other non-real-time performance properties, such as throughput
 - resource utilization
- ❑ These trade-offs are application-specific
- ❑ The programmers and users must be able to reason about these trade-offs

There have been many “real-time” programming languages

- ❑ However, none have been commercially successful in the sense of being significantly adopted in that domain
 - many were academic research projects
 - most did not focus on the core real-time issues of managing computing resources in order to satisfy application timeliness optimality and predictability requirements
- ❑ Instead, they typically emphasized
 - the orthogonal (albeit important) topic of concurrency
 - and other topics important to the whole field of embedded computing systems (of which real-time computing systems are a subset)

Ada 95 has been the most successful real-time programming language

- ❑ **Ada 95, including its Real-Time Systems Annex, has been the most successful real-time language, in terms of both adoption and real-time technology**
- ❑ **One reason is that Ada is unusually effective (among real-time languages and also operating systems) across the real-time computing system spectrum**
 - **from programming-in-the-small in traditional device-level control subsystems**
 - **to programming-in-the-large in enterprise command and control systems**
- ❑ **Despite that achievement, a variety of non-technical factors crippled Ada's commercial success (including in defense applications)**

JSR-50: the Distributed Real-Time Specification for Java

- ❑ **There is strong pull (e.g., in the defense, industrial automation, telecom, and financial markets) for “distributed real-time Java”**
- ❑ **JSR-50 is to extend the RTSJ to include distributed real-time systems – the DRTSJ**

JSR-50: the Distributed Real-Time Specification for Java

- ❑ Java Specification Request written by Jensen (MITRE); supported by IBM and seven other organizations**
- ❑ Sun approved JSR-50 in April 2000**
- ❑ Sun appointed Jensen (MITRE) as Specification Lead**
- ❑ At the time of this writing (April 2000), the Expert Group has met three times**
- ❑ The Expert Group began with the approach proposed in JSR-50, and may modify it somewhat**
- ❑ The technical approach is very similar to that of the proposed specification for Dynamic Scheduling Real-Time CORBA**

The Distributed Real-Time Specification for Java: Summary

- ❑ Work has begun on the Distributed Real-Time Specification for Java (DRTSJ), in Sun's Java Community Process
- ❑ "Distributed real-time" means acceptable predictability of multi-node application behaviors' collective timeliness (given whatever OS and network infrastructure), regardless of the programming model (control flow, mobile code, etc.)
- ❑ A multi-node application behavior's timeliness properties must be explicitly employed for resource management consistently on each node involved in that application multi-node behavior
- ❑ In Java, using RMI, these properties must be acquired, propagated, and deposited when RMI's and any associated returns occur
- ❑ With control flow programming models, that enhancement supports predictability of end-to-end timeliness

JSR-50: the Expert Group member organizations

- Ada Core**
- aJile**
- Boeing**
- CMU**
- IBM**
- LaCross Consulting**
- Lockheed Martin**
- MITRE**
- Nokia**
- Nortel**
- NSIcom**
- Omron**
- Sun**
- TimeSys**
- USAF-RL**
- Wellings (York U)**
- Wells (The Open Group)**
- Yamatake**

A distributed system has application entities that exhibit multi-node behaviors

- ❑ For the purpose of this presentation, the term *distributed system* informally refers to a computing system whose programming model is based on there being application entities that exhibit multi-node behaviors
- ❑ Each of these multi-node behaviors has multi-node properties – these properties may include (but are not limited to)
 - a unique identifier
 - timeliness
 - security
 - resource ownership
 - transactional context

Distributed systems can be categorized in various ways for various purposes

- ❑ Here we categorize them in a very simple way according to their programming model for the multi-node interaction aspect of application behaviors
 - *networked* (asynchronous message passing among objects)
 - *control flow* (method invocation between objects)
 - *data flow* (e.g., publish/subscribe among objects)
 - *blackboards/spaces* (e.g., Linda, JavaSpaces)
 - *mobile objects, autonomous agents, autonomous decentralized systems*
- ❑ The first three of these categories have long histories of successful use in real-time as well as non-real-time application domains

A distributed system usually has one primary application programming model

- ❑ A distributed system usually has one programming model as the first class abstraction, and sometimes others are implemented in terms of it (perhaps at lower performance)**
- ❑ For example, OMG's CORBA standard specifies a first class control flow programming model, but an optional data flow programming model is being proposed as a layered service**
- ❑ Of course, a first class distributed system programming model is normally implemented on a communication facility that typically has multiple levels which are not visible to the application –
e.g., a blackboard abstraction may be implemented using RPC that is implemented using asynchronous message passing**

A real-time distributed system has acceptable timeliness of multi-node application behaviors

- ❑ **The defining characteristic of any real-time distributed computing system, whatever its programming model, is that**
 - **the timeliness (optimality and predictability of optimality) of each multi-node application behavior**
 - **on each individual node it involves**
 - **collectively on all nodes it involves**
 - **is acceptable to that application**
 - **under the current circumstances**
- ❑ **The “current circumstances” include the latency characteristics of the underlying infrastructure (OS’s, network)**

A multi-node behavior's timeliness properties must be used coherently on all involved nodes

- ❑ In most cases, the fundamental requirement for achieving acceptable multi-node timeliness is that a multi-node application behavior's timeliness properties**
 - time constraints**
 - expected execution time**
 - execution time received thus far**
 - etc.**

be explicitly employed for resource management (scheduling, etc.)

coherently on each node involved in that application multi-node behavior

Multi-node application behavior 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;
but the Distributed Real-Time Specification for Java is concerned primarily with dynamic systems

Real-time distributed Java systems can use RMI to propagate timeliness properties

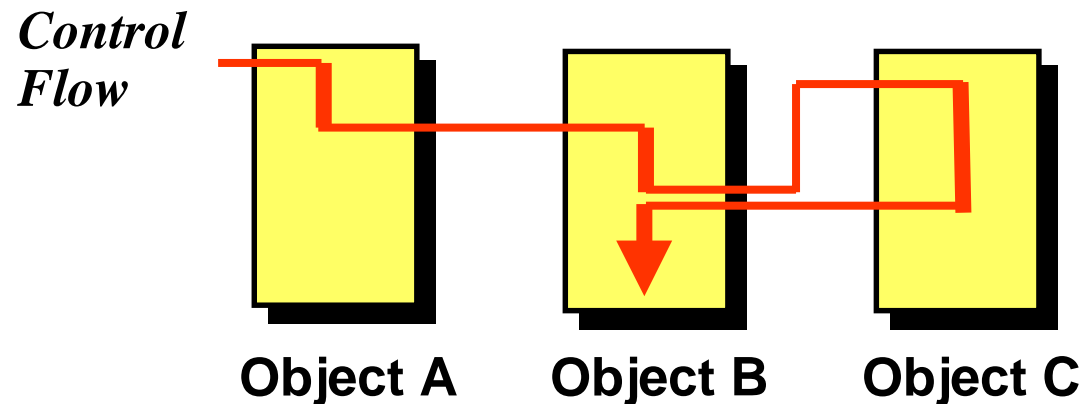
- ❑ In real-time distributed Java systems that use RMI, a multi-node application behavior's end-to-end timeliness (and other) properties must be
 - acquired
 - propagated
 - and depositedwhen RMI's and any associated returns occur
- ❑ This should be transparent to the application programmer
- ❑ This enhancement is an appropriate mechanism, regardless of what programming model semantics are manifest with RMI's –
e.g., whether performing a basic RPC-like method invocation, or passing an object by copy or reference

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

- ❑ Timeliness properties can be propagated, and used consistently by node resource managers
 - e.g., every node schedules using the same policy
 - 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
- ❑ This RMI enhancement can be used for any of these cases

JSR-50's initial emphasis is on control flow programming models

- ❑ Control flow models are familiar and accepted in distributed real-time practice
- ❑ A distributed control flow program's components may be spread across multiple computing nodes
- ❑ Execution flows as one or more sequences of constituent operations occurring on multiple nodes, each via a sequence of method invocations/returns



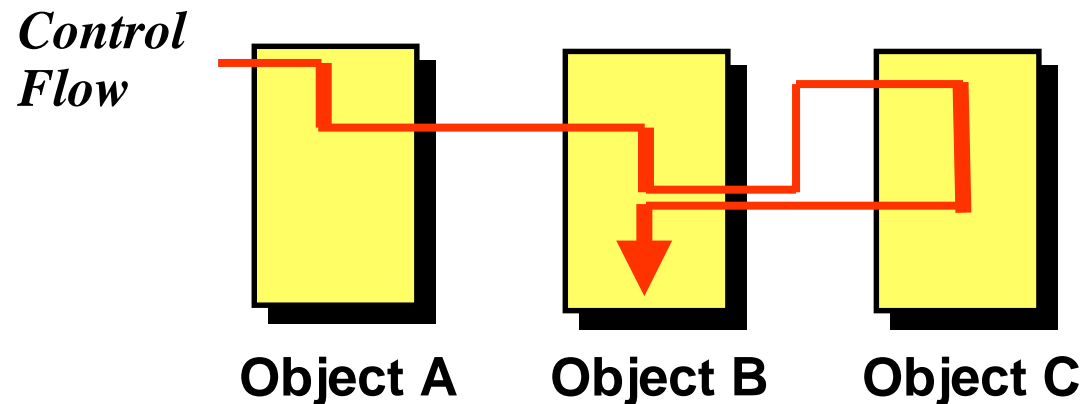
Control flow has compelling advantages as a native model for distributed real-time software

- ❑ Many distributed real-time computer systems and applications involve a mixture of control flow and data flow
- ❑ Distributed control flow is a natural, well-understood, incremental extension to local control flow (procedure calls)
- ❑ Familiarity is an especially important factor in attaining adoption in real-time application domains, such as industrial automation, defense, and telecommunications
- ❑ All distributed systems have inherent complications due to network latencies, partial failures, synchronization and concurrency control, etc. –

many of these are more easily managed in control flow programming models than in other (e.g., data flow, autonomous agent) programming models

Timeliness in distributed real-time control flow programming models is end-to-end

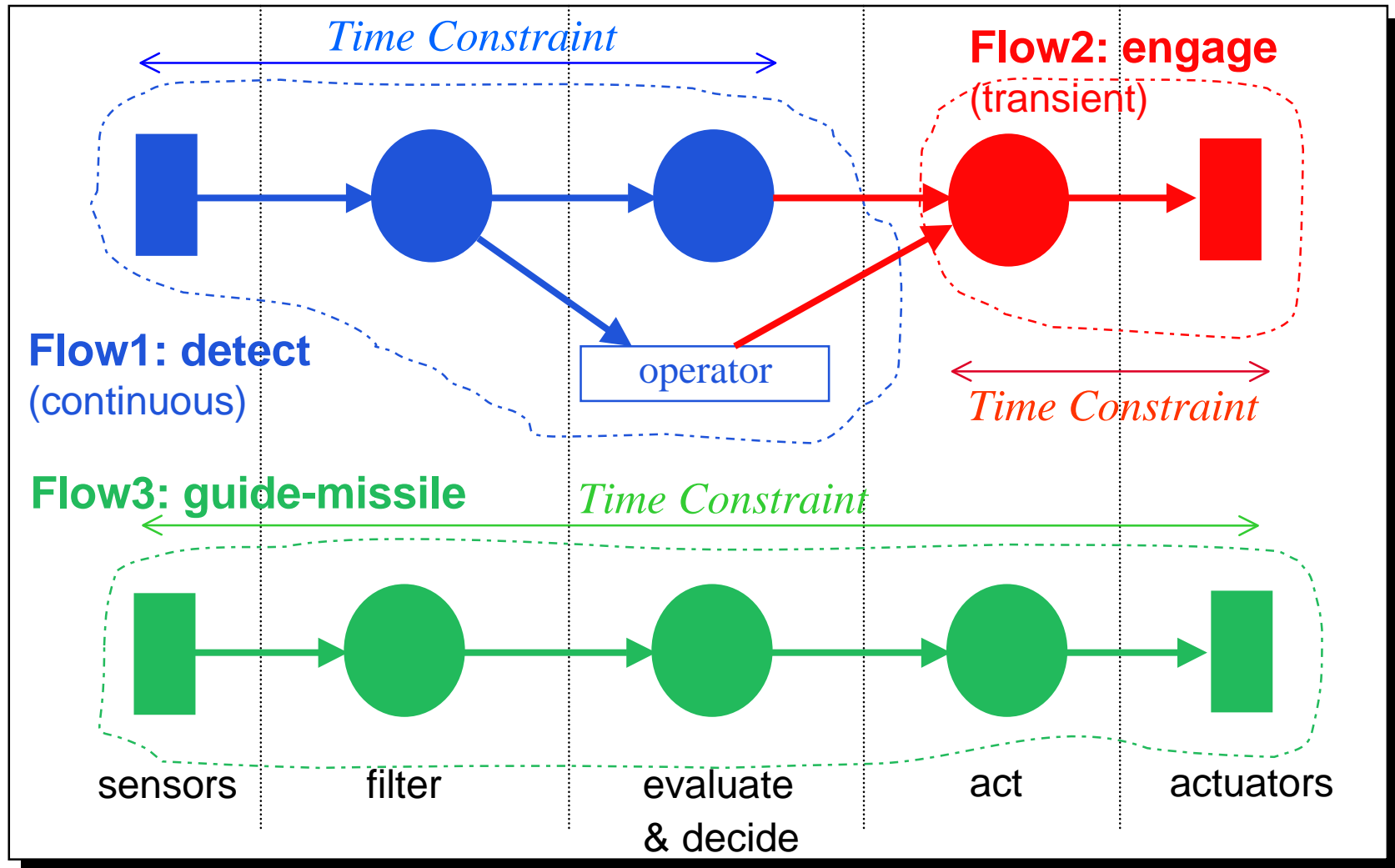
- ❑ The defining characteristic of any real-time distributed computing system having a control flow programming model is that
 - the timeliness (optimality and predictability of optimality) of each trans-node application control flow
 - is acceptable to that application
 - under the current circumstances



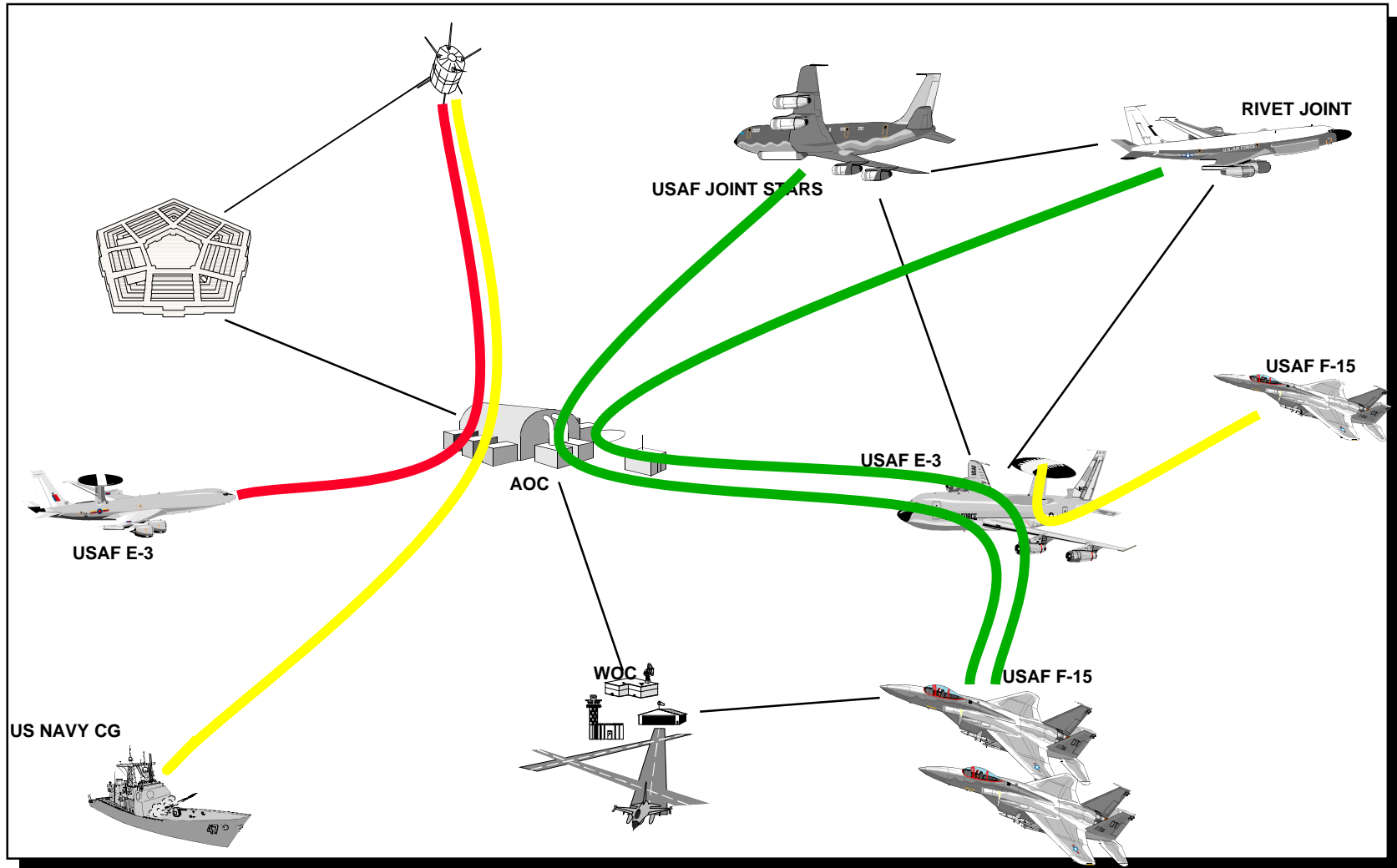
Control flow method invocations (and returns) are location-independent, and other code is not

- ❑ All the code in a control flow model 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 control flow can be obtained by a programming model having
 - location-independent invocations and returns
 - location-dependent (node-local) code otherwise

Example control flows in a notional anti-air warfare system



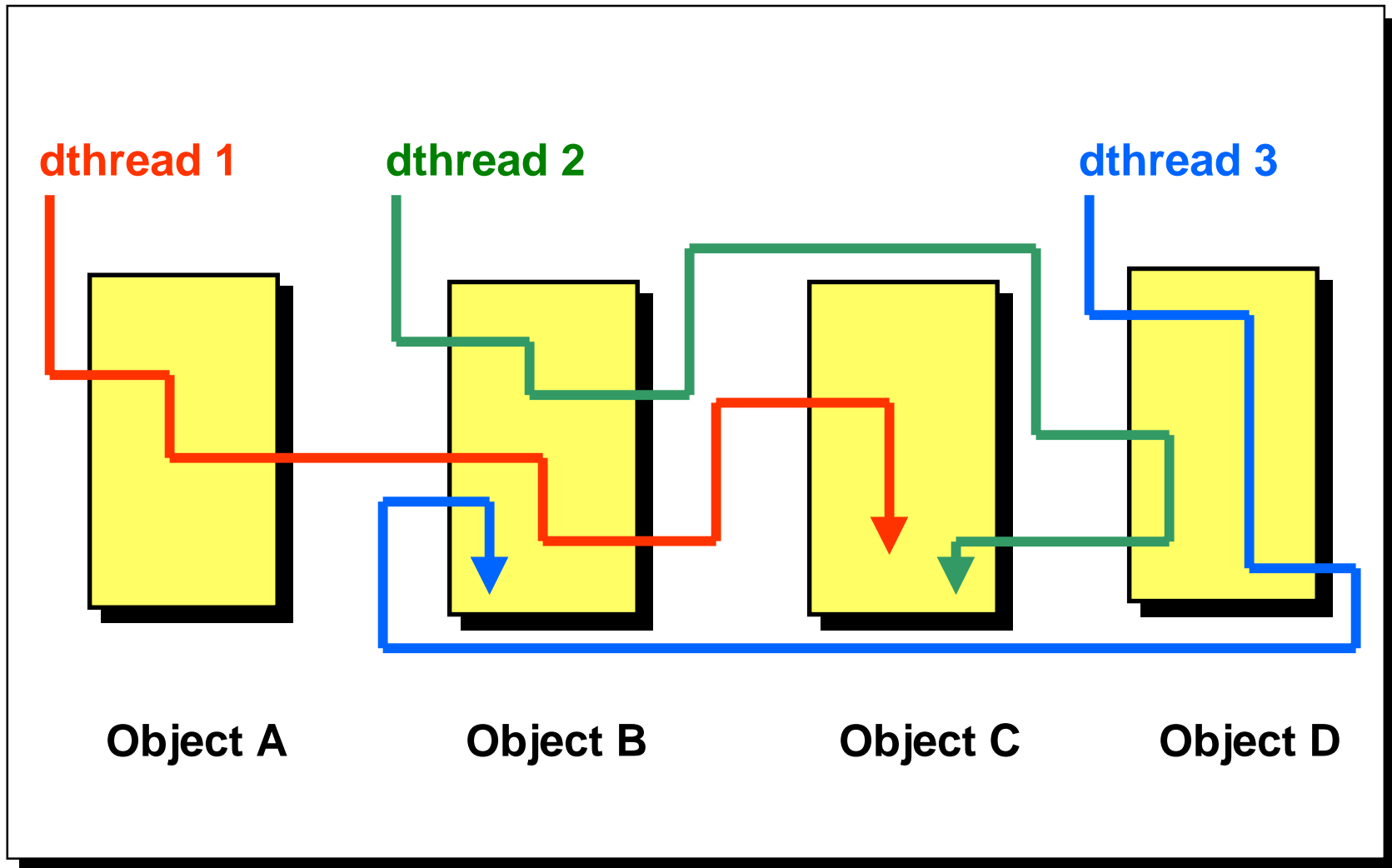
Example of control flows in a notional C² system of systems



A distributable thread 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 a *distributable thread***
- ❑ **A distributable thread is a logically distinct and identifiable locus of control flow movement, within and among objects (and thus nodes)**
- ❑ **A distributable thread executes a remote method, like a local one, directly itself – by extending and retracting itself between objects and (transparently) nodes**
- ❑ **The distributable thread (not a local 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



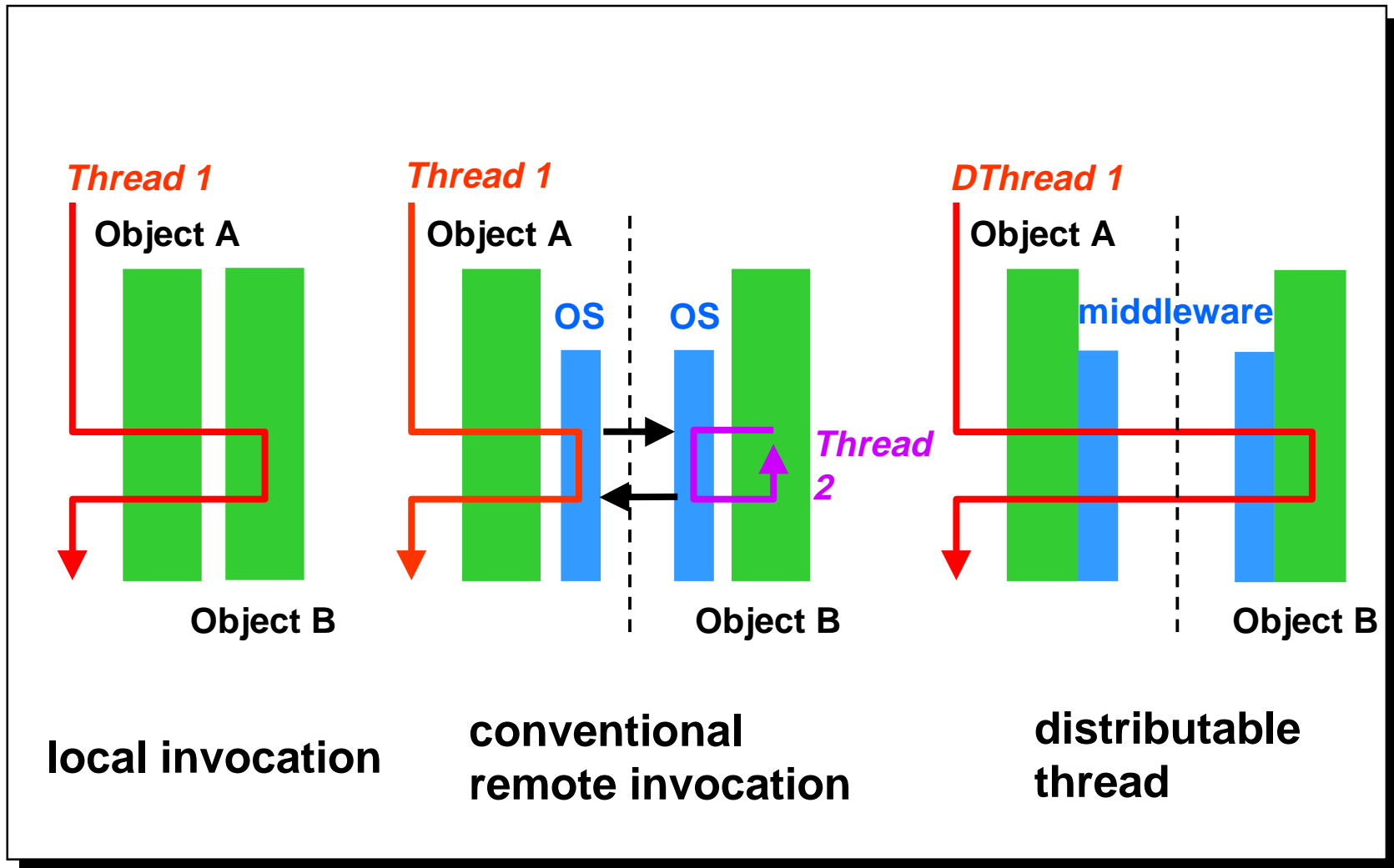
Concurrency is at the distributable thread level

- ❑ 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 distributable 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 thus physical nodes, in a location-independent way
 - impedes maintaining end-to-end properties

A distributable thread has location-independent method invocations



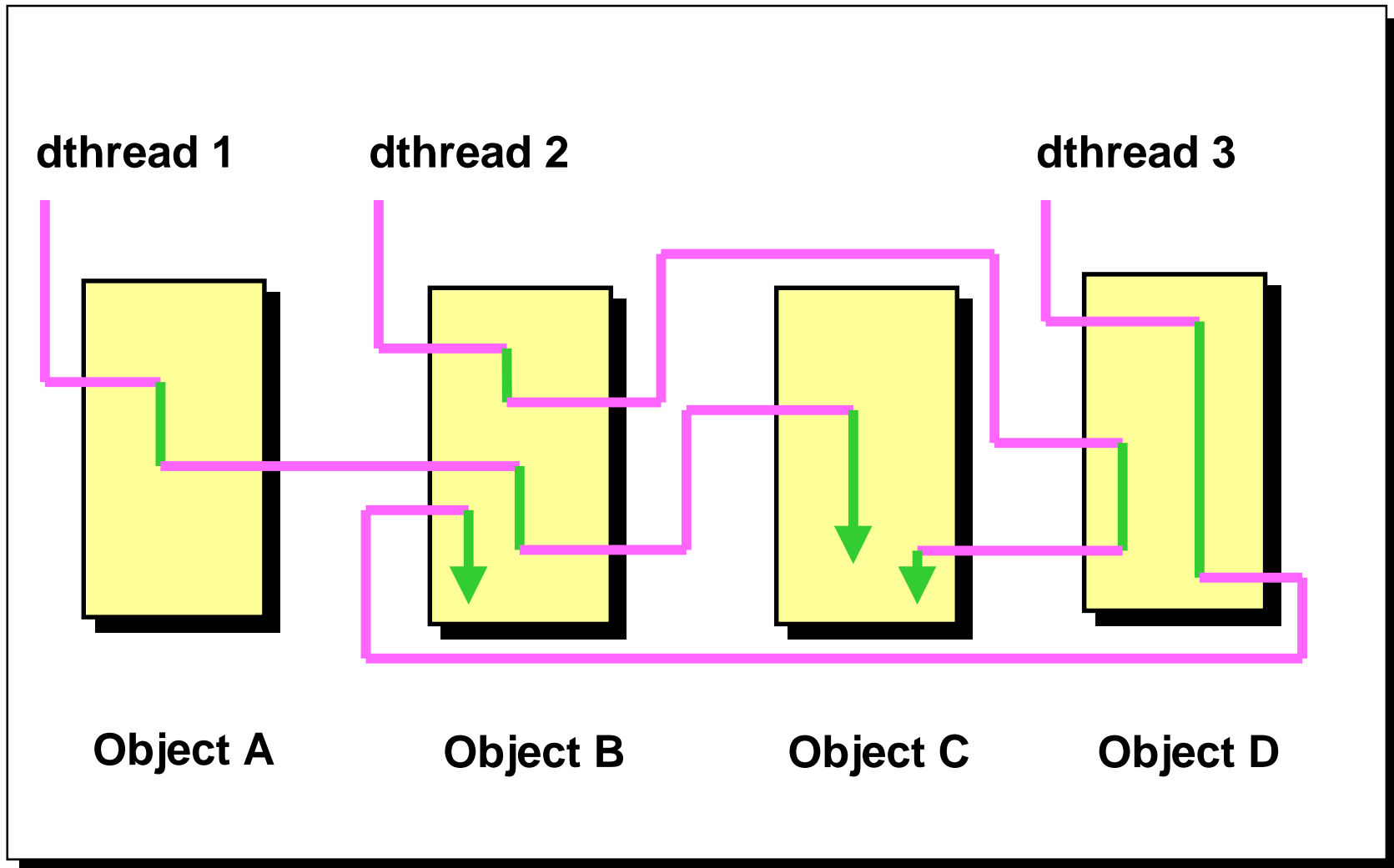
A distributable thread is sequential rather than synchronous

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

A distributable thread is built using local (e.g., RTOS) threads and method invocations

- ❑ The distributable thread abstraction is implemented using local (RTOS or JVM) threads, as part of
 - middleware**
 - local operating systems**
 - JVM's****
- ❑ In Java, a distributable thread would be implemented by the concatenation of local (per node) threads sequentially performing RMI's when they transit nodes**

A distributable thread is built using local (e.g., RTOS) threads and method invocations



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

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

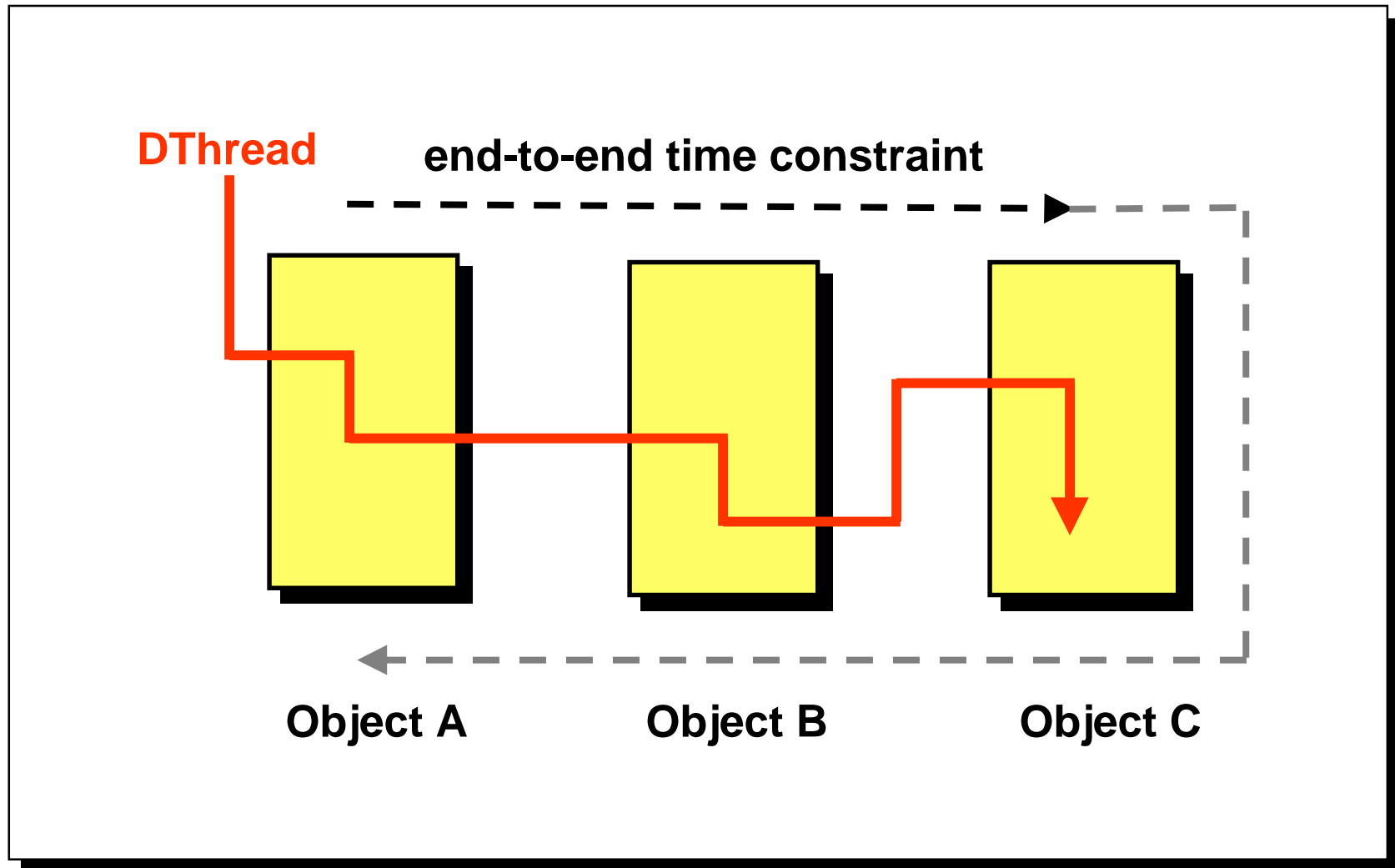
A distributable thread has end-to-end timeliness attributes

- ❑ Each distributable thread may have execution scheduling attributes – e.g., time constraints, 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 policy, regardless of the distributable thread's execution point transiting nodes
- ❑ Any of the three distributed scheduling approaches can be used
- ❑ The goal is to provide acceptable (as defined by the application) end-to-end timeliness of collective distributable thread execution

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 policy instance
 - in the OS, JVM, or middleware
- ❑ When/if it returns, updated timeliness parameters are propagated back to the invoker's scheduling policy instance
- ❑ (Other end-to-end properties may also be propagated – e.g., ID, resource ownership, dependencies, rights, security, transactional context)
- ❑ This should be transparent to the application programmer

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



The distributable thread model applies to the whole predictability/time-frame space of real-time systems

- ❑ The distributable thread approach to control-flow style distributed programming model is applicable to real-time systems which are
 - hard
 - 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 distributable thread abstraction supports application timeliness requirements everywhere in that two-dimensional predictability/time-frame space of distributed real-time Java systems

The distributable thread abstraction also has implementation advantages

- ❑ The distributable thread abstraction automatically supports implementation needs, such as
 - 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 distributable thread abstraction has been widely adopted for microkernel-based OS's for these implementation advantages, independent of the programming model

A fully specified DT abstraction must include additional facilities – not necessarily in this DRTSJ

- ❑ **A specification of a complete DT abstraction would include (but not be limited to) facilities to deal with**
 - **DT integrity (failure detection and recovery) despite partial node and path failures**
 - **DT control (like thread control) – pause, resume, abort, etc.**
 - **distributed event handling**
 - **asynchronous events of interest to a DT – i.e., changes in predicated system state, such as a time constraint expiration – are delivered to its execution point for possible handling**
 - **and perhaps from the execution point back up the invocation chain for (additional) handling**
 - **distributed concurrency control among DT's**
- all in a timely manner**

Distributed Threads in a Java Context: Design Choices

- Build on RMI
- Build on JSR-78 RMI
- Devise a new RT-RMI facility
- Use an entirely new remote access approach

Distributed Threads: Building on RMI

- ❑ **DRTSJ could propose a simple extension to RMI that supports timeliness properties among nodes**
 - **facilitates acceptably predictable end-to-end timeliness of distributed threads**
 - **this is application-specific**
 - **extends RMI to**
 - **obtain, propagate, and deposit timeliness attributes among the scheduling policy instances at each node a distributed thread visits**
transparently to the programmer
- ❑ **RMI must be made predictable for the cases of interest**
 - **without affecting non-RT uses, syntax, specified semantics, and tools for RMI**

Distributed Threads: Building on JSR-78's RMI

- ❑ **JSR-78: RMI Custom Remote References**
 - **Objective: provide a general framework in J2SE RMI for customizing remote invocation behavior**
- ❑ **JSR-78 is done for J2EE/J2SE, not J2ME**
 - **technical concerns**
 - **are the intermediate interfaces sufficient for J2ME's needs?**
 - **are the supplied customizations useful for J2ME?**
 - **are they sufficient for efficiently and effectively supporting distributed threads?**
 - **logistical concerns**
 - **approval for J2ME platform**
 - **availability of code on a J2ME platform (for use in Reference Implementation)**

Distributed Threads: Devise a New RT-RMI

- ❑ Design and build a new, flexible, predictable RMI-like facility that coexists with or subsumes RMI
- ❑ RT-RMI
 - permits modification of JSR-78 interfaces and capabilities
 - must presumably “look like” and “feel like” RMI
 - else might confuse programmer
 - should accommodate (subsume?) non-RT RMI uses
 - could still be subject to RMI constraints
 - ability to affect other portions of the “distributed thread”
 - timeliness of responses
 - types of exceptions, inheritance, required interfaces, etc.

Distributed Threads: Devise a New Remote Access Approach

- Ignore RMI (sort of; it should probably work if invoked)
- Allows maximal freedom
- Permits or requires reexamination of fundamentals
 - protocols
 - interfaces
 - naming
 - stubs, skeletons
 - references, leases, garbage collection
 - exceptions
 - communication models
- Increases hurdle to mass acceptance (or at least tolerance)

Conclusion:

the Distributed Real-Time Specification for Java

- ❑ Real-time computing is about predictability of timeliness**
- ❑ Distributed real-time computing is about predictability of timeliness of multi-node (e.g., trans-node) behaviors**
- ❑ Acceptable predictability of timeliness of multi-node behaviors requires suitably consistent resource management on the nodes that the behaviors transit**
- ❑ Suitably consistent resource management on multiple nodes requires that they share sufficient information about the behaviors' timeliness**
- ❑ Shared information in a network must be explicitly propagated among the nodes**
- ❑ The DRTSJ will allow RTSJ Java systems that use RMI to propagate shared information for consistent node resource management to meet multi-node (e.g., end-to-end) timeliness predictability needs**

“Real-time Java” is likely to be the first successful real-time programming language

- ❑ Ada 95 is a successful real-time language technologically but is less successful commercially**
- ❑ Java is already ubiquitous**
- ❑ The Java platform’s WORA promise currently offers the best prospective opportunity for application portability**
- ❑ Real-time Java appears to be successfully addressing the deficiencies of Java for real-time computing systems**
- ❑ Several major vendors have already announced that they will sell real-time Java products**
- ❑ Real-time Java is poised to be the first commercially as well as technologically successful real-time programming language**
- ❑ “Distributed real-time Java” will be very important to Java’s success in the real-time application domains**

Resources:

RTSJ and DRTSJ

- ❑ **NIST Special Publication 500-243 “Requirements for Real-time Extensions for the Java Platform”**
 - <http://www.nist.gov/itl/div897/ctg/real-time/rt-doc/rtj-final-draft.pdf>
- ❑ **Real-Time Specification for Java**
 - <http://www.rtj.org>
- ❑ **Distributed Real-Time Specification for Java**
 - <http://www.drtsj.org>
- ❑ **J Consortium**
 - <http://www.j-consortium.org>