

The Distributed Real-Time Specification for Java

A Proposed Initial Approach

E. Douglas Jensen

jensen@real-time.org

<http://www.real-time.org>, <http://www.drtsj.org>

Revised September 28, 2000

MITRE

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 trans-node application behaviors' end-to-end timeliness (given whatever OS and network infrastructure), regardless of the programming model (control flow, mobile code, etc.)**
- ❑ **A trans-node application behavior's timeliness properties must be explicitly employed for resource management consistently on each node involved in that application trans-node behavior**
- ❑ **These properties must be acquired, propagated, and deposited when RMI's and any associated returns occur**
- ❑ **Another objective is to apply that RMI enhancement mechanism to support control flow programming models**

Outline

- ❑ Java and “real-time Java”
- ❑ Sun’s Java Community Process
- ❑ JSR-1 the Real-Time Specification for Java
- ❑ JSR-50 the Distributed Real-Time Specification for Java
- ❑ A real-time lexicon
- ❑ Categories of “distributed systems”
- ❑ DRTSJ: the initial approach
- ❑ The *activity* abstraction
- ❑ Activity-based programming models: past, present, and future
- ❑ Conclusion

Conclusion

- ❑ **Real-time computing is about predictability of timeliness**
- ❑ **Distributed real-time computing is about predictability of timeliness of end-to-end behaviors**
- ❑ **Acceptable predictability of timeliness of end-to-end 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 properties**
- ❑ **Shared information in a network must be explicitly propagated among the nodes (no shared memory)**
- ❑ **The DRTSJ will allow RTSJ Java systems that use RMI to propagate shared information for consistent node resource management to meet 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**

NIST coordinated several workshops on requirements for real-time Java

- ❑ Beginning 6/98, NIST convened a series of six workshops to establish a consensus on requirements for real-time Java**
- ❑ More than 50 companies participated**
- ❑ NIST Special Publication 500-243 “Requirements for Real-time Extensions for the Java Platform”**
 - Terms and Concepts (also used in this presentation)**
 - Java Traits**
 - Core Functionality and Profiles**
 - Core Requirements**
 - Goals and Derived Requirements**

There are two independent groups working on real-time Java

- ❑ Sun's Java Community Process**
 - Java Specification Request JSR-1 Expert Group**
 - “Real-Time Specification for Java”**
- ❑ The J Consortium**
 - Real-Time Java Working Group**
 - “Real-Time Core Extensions”**
- ❑ The distributed real-time Java work presented here is being done in Sun’s Java Community Process**
- ❑ Currently there is no corresponding work being done by the J Consortium**

Sun's Java Community Process (JCP 2)

- ❑ **Java specification requests (JSR's) are submitted by JCP members to the JCP Executive Committee (EC)**
- ❑ **Specification requests are open to public comment**
- ❑ **If the EC approves a specification request (90 thus far)**
 - **the EC publishes a Call for Experts**
 - **the EC selects a specification lead**
 - **the spec lead selects his Expert Group (EG) members**
 - **the EG writes a specification, which the EC and then the community review**
 - **the spec lead is responsible for the creation of a reference implementation (RI) and a technology compatibility kit (TCK)**
 - **the spec lead defines the RI and TCK licensing terms, which must be fair and equitable**

Sun's Java Community Process (JCP 2)

(continued)

- **the draft specification is open first to community, and then (after EC approval) to public, review**
- **first the EG, then the EC, approves the specification, RI and TCK**
- **the final specification is posted on the JCP website**
- **the EC names a maintenance lead, usually the specification lead, to shepherd the specification, RI, and TCK throughout its life cycle**

JSR-1: the Real-Time Specification for Java (RTSJ)

- ❑ **To facilitate its major goal of operating system and hardware independence, the Java language was deliberately given a weak vocabulary in areas such as**
 - **thread behavior, synchronization, interrupts, memory management, and input/output**
 - **all these areas are critical to real-time computing**
- ❑ **The JSR-1 Expert Group is extending the Java platform (language and JVM) for real-time systems**
 - **began 3/99**
 - **spec 1.0 0th edition publication 6/00, 1st edition 9/00**
 - **reference implementation: alpha 4Q/00, version 1 1Q/01**
 - **technology compatibility kit: version 1 1Q/01**
- ❑ **JSR-1 deliberately deferred distributed real-time systems**

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”**
- ❑ This presentation is about JSR-50: extend the RTSJ to include distributed real-time systems – the DRTSJ**
- ❑ Sun approved JSR-50 in 4/00**
- ❑ At the time of this presentation (9/00), the Expert Group has been formed and is planning its kick-off meeting (12/00)**
- ❑ The Expert Group begins with the approach proposed in JSR-50, and may modify it somewhat**

JSR-50: the Expert Group member organizations

- Ada Core**
- aJile**
- Boeing**
- CMU**
- Cyberonics**
- DARPA**
- IBM**
- LaCross Consulting**
- Lockheed Martin**
- MITRE**
- NASA-JPL**
- Nokia**
- Nortel**
- NSIcom**
- Omron**
- Sun**
- Telkel**
- TimeSys**
- USAF-RL**
- Wellings (York U)**
- Wells (TOG)**
- Yamatake**

Outline

- ❑ Java and “real-time Java”
- ❑ Sun’s Java Community Process
- ❑ JSR-1 the Real-Time Specification for Java
- ❑ JSR-50 the Distributed Real-Time Specification for Java
- ❑ **A real-time lexicon**
- ❑ Categories of “distributed systems”
- ❑ DRTSJ: the initial approach
- ❑ The *activity* abstraction
- ❑ Activity-based programming models: past, present, and future
- ❑ Conclusion

A real-time lexicon is essential for the Distributed Real-Time Specification for Java

- ❑ Real-time concepts and terms are popularly used in ways that are widely different, confused, and often incorrect**
- ❑ To deal with real-time computing as a scientific and engineering discipline requires the concepts and terms to be much more well-defined**
- ❑ The NIST workshops on requirements for real-time Java adopted a lexicon of real-time concepts and terms**
- ❑ That lexicon**
 - was useful for**
 - the real-time Java requirements group**
 - both real-time Java specification groups**
 - is essential for the DRTSJ**
 - is therefore synopsized here**

The concepts and terms in this real-time lexicon are well-defined, general, and scale up

- ❑ **The intent of this lexicon is to define real-time concepts and terms that**
 - **are well-defined – have thoughtful, unambiguous meanings**
 - **are general – cover a wide range of cases**
 - **scale up – apply**
 - **just as well to complex, dynamic, and distributed systems**
 - **as to simple, static, centralized ones**

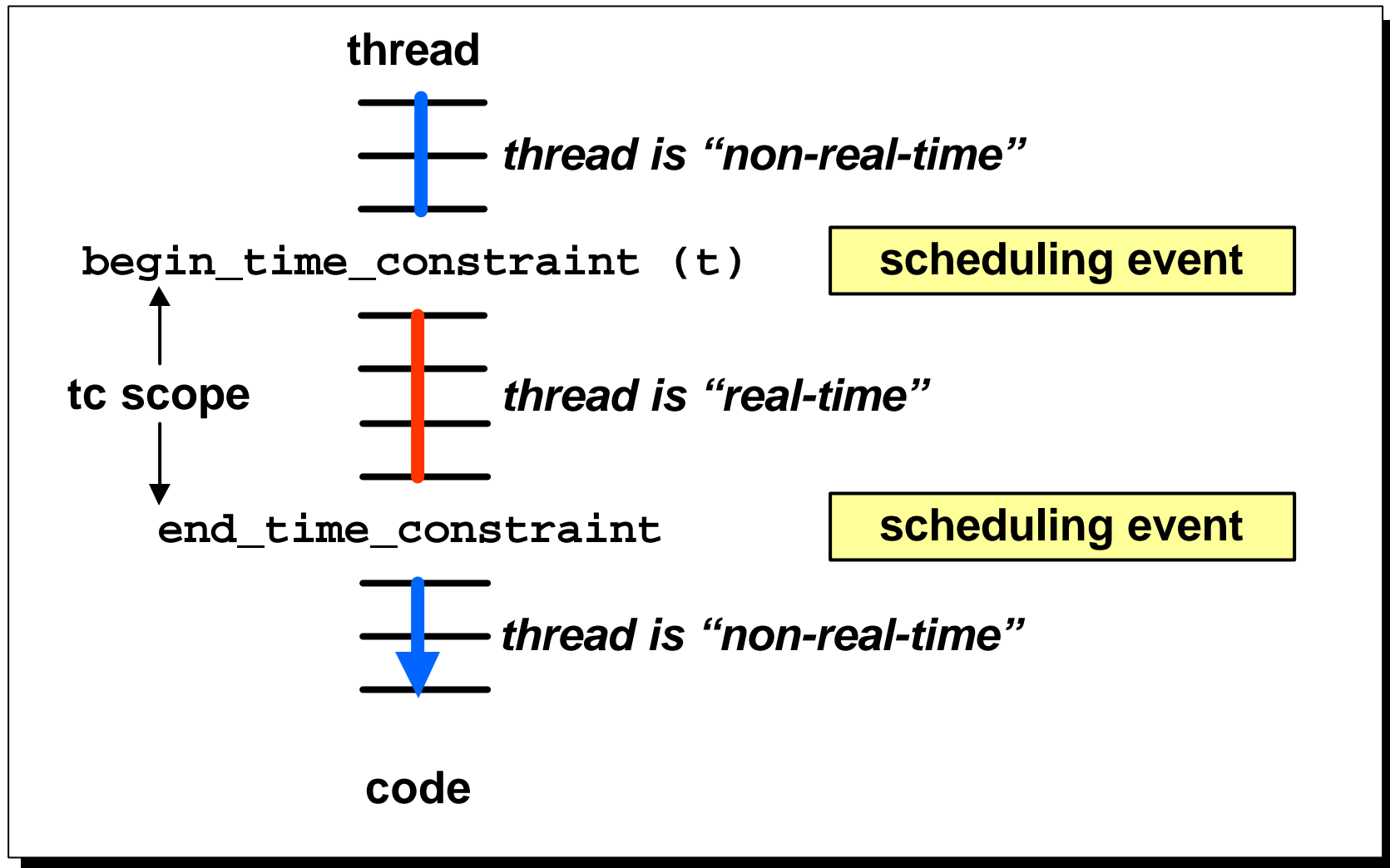
A real-time application has time-constrained actions

- ❑ A real-time application includes actions whose completions are time-constrained**
- ❑ These time constraints are inherent in the behavior of the application and its execution environment**
- ❑ The best known action completion time constraint is a deadline, but there are many other widely used kinds**

A time constraint is a lexically scoped attribute of a thread

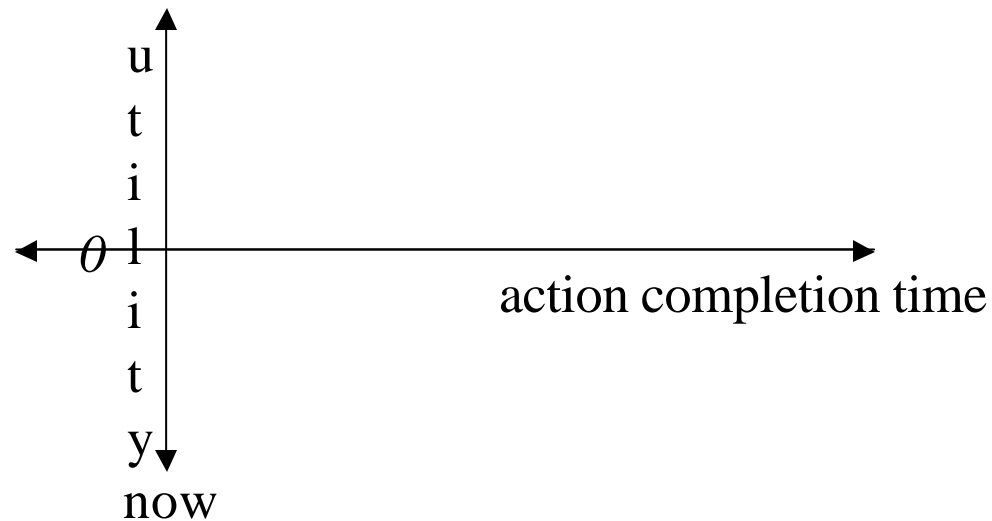
- ❑ A *time constraint* is a lexically scoped attribute of a computational entity (say, thread)
- ❑ Each time-constrained action is performed by a thread executing code that includes a time-constrained block of instructions called the time constraint's *scope*
- ❑ The timeliness of a thread's transit through that scope depends on the time at which the thread's execution point reaches the end of the scope
- ❑ While a thread is executing within a time constraint scope, it is a “real-time thread,” otherwise, it is a “non-real-time thread”
- ❑ The passage of a thread's execution point into or out of a time constraint scope is a *scheduling event* – a decision must be made about which thread should be executing

A time constraint is a lexically scoped attribute of a thread



An action completion time constraint can be expressed as a utility function

- An action's time constraint can be expressed as the *utility* – either reward or penalty – for completing the action as a function of when the action is completed



A deadline time constraint specifies action completion timeliness in terms of a deadline time

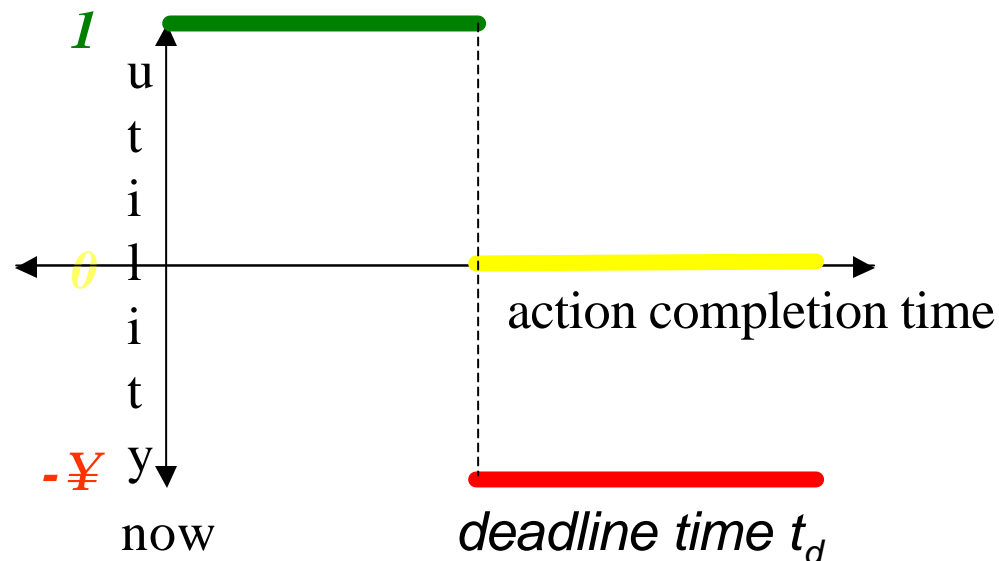
- ❑ **A *deadline* time constraint specifies that**
 - **the timeliness of the thread's transit through the deadline scope**
 - **depends on whether**
 - **the thread's execution point reaches the end of the scope**
 - **before the deadline time has occurred – i.e., on whether the deadline is satisfied**

A hard deadline specifies that an action is timely or untimely with respect to the deadline time

- ❑ A *hard deadline* is a completion time constraint, such that
 - if the deadline is met
 - then the time constrained portion of the thread's execution is timely
 - otherwise
 - that portion is not timely

All hard deadlines have the same binary values on each side of the deadline times

- A hard deadline time constraint is expressed functionally as a binary-valued downward step
 - a singular positive value (e.g., "1") for completion times less than or equal to the deadline time
 - "0" or some singular negative value (according to the desired convention) thereafter

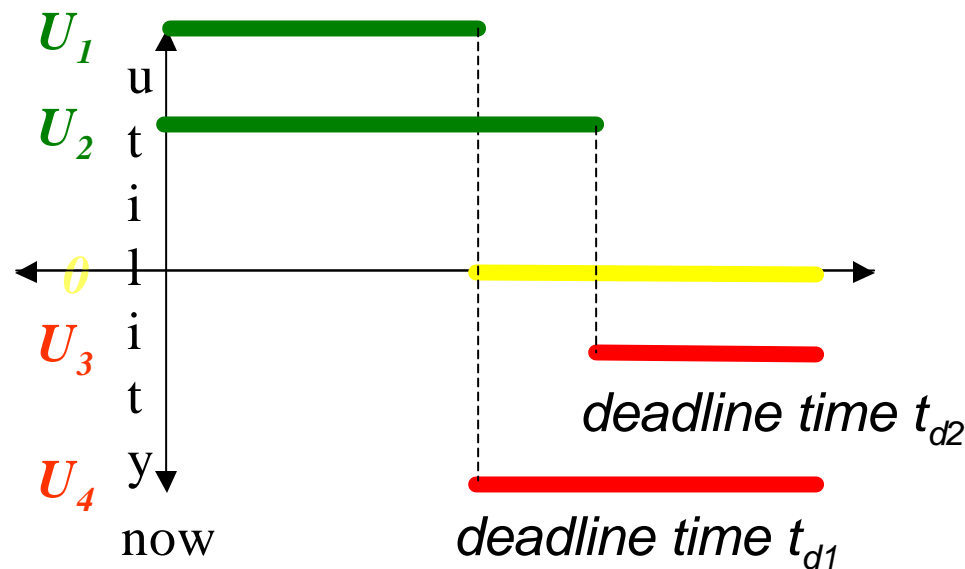


A soft deadline specifies that an action is more or less timely with respect to the deadline time

- ❑ **A *soft deadline* is a completion time constraint, such that**
 - **if the deadline is met**
 - **then the time constrained portion of the thread's execution is more timely**
 - **otherwise**
 - **that portion is less timely**
- ❑ **A hard deadline is a special, simple, case of a soft deadline**

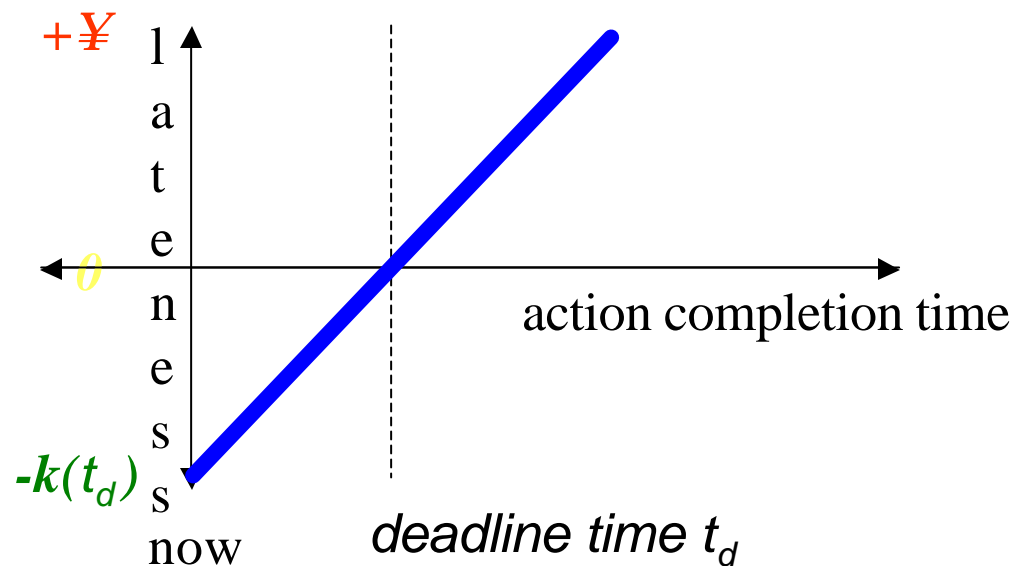
A soft deadline may have any pair of values on each side of the deadline time

- ❑ A soft deadline time constraint may be bi-valued with any utility values on each side of the deadline time
- ❑ These values are one way of expressing action importance



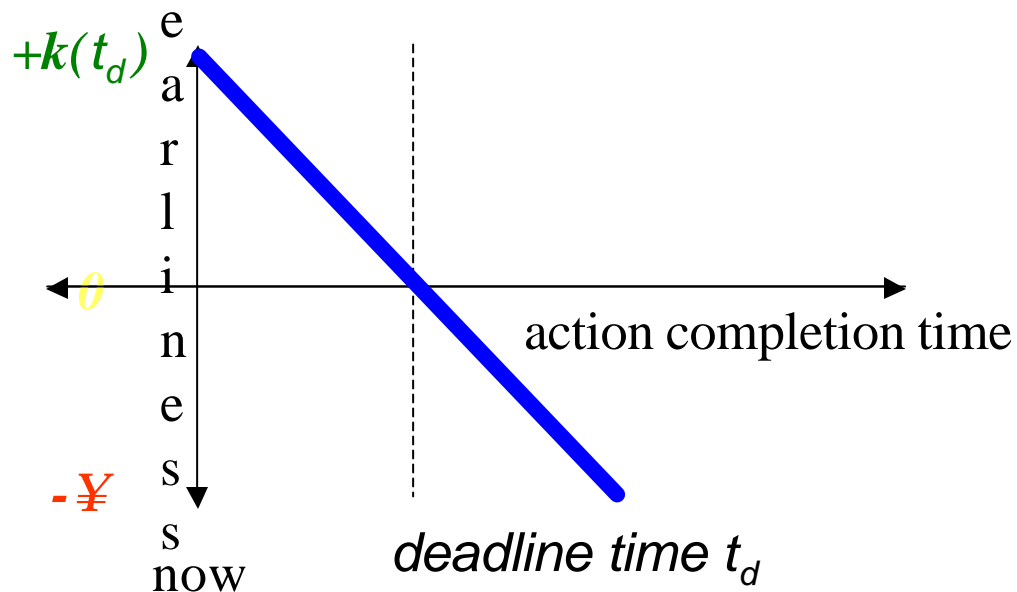
A soft deadline often specifies timeliness in terms of lateness with respect to the deadline time

- In the scheduling theory context, "more" and "less" timely is usually measured in terms of
 - lateness: deadline minus completion time
 - or tardiness: $\max\{0, \text{lateness}\}$



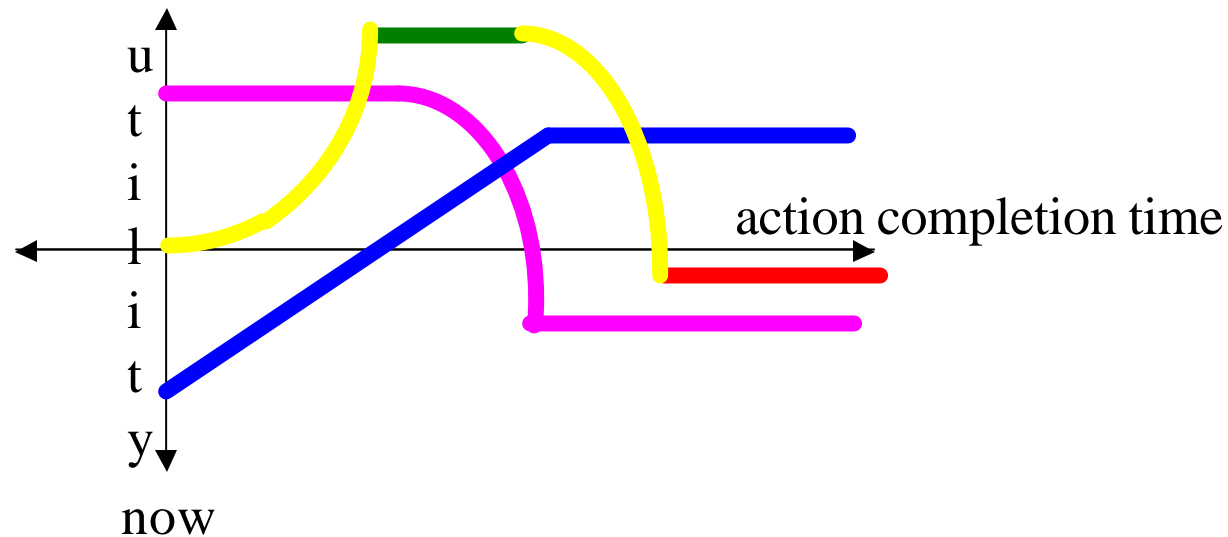
In the utility function model of timeliness, lateness is inverted to earliness

- To be consistent with the intuitive convention in this model that larger function values represent greater utility, lateness is inverted to earliness



A general time constraint may be any relationship between action completion time and utility

- A *soft (general)* time constraint is any relationship between
 - the time when the thread's execution point reaches the end of that time constraint's scope
 - and the utility to the system of when it does



In general, a real-time computer system has multiple time-constrained threads at once

- ❑ Multiple time-constrained threads generally contend for resources**
 - notably processor cycles**
 - but also other exclusively shared resources**
 - physical – e.g., communication paths**
 - logical – e.g., synchronizers**
- ❑ This contention must be resolved into a sequence of resource accesses**
 - e.g., thread executions**

Thread *scheduling* is deciding in what order they all will execute

- ❑ Each time thread scheduling is performed, it establishes a sequence – a *schedule* – for all threads ready at that time
- ❑ Scheduling may be performed
 - statically (prior to execution time) by a person or a program
 - dynamically (at execution time) by a user or system software

Thread *dispatching* is granting resource access – e.g., launching – the currently most eligible thread

- ❑ When scheduling is employed, dispatching occurs in schedule order**
- ❑ Thread scheduling is not always necessary or computationally feasible – dispatching alone may be sufficient**
- ❑ Dispatching when scheduling is not employed establishes a thread resource access – e.g., execution – sequence one thread at a time**

A sequencing event may cause the sequence to change

- ❑ **A *scheduling event* or *dispatching event* (in general, a *sequencing event*) is an event in time or space (code) that may cause the execution sequence to change – e.g., a thread**
 - becoming ready or blocked
 - contending for a resource
 - violating its time constraint
 - completing within its time constraint

Contention among time-constrained threads should be resolved by a timeliness-based criterion

- ❑ **Contention for execution (and all other physical and logical resources) should be resolved**
 - **whether scheduling**
 - **or dispatching alone**

according to an application-specific policy that seeks maximal usefulness as measured by a sequencing optimality criterion
- ❑ **In real-time computing systems, this criterion has two timeliness dimensions**
 - **collective timeliness optimality**
 - **predictability of collective timeliness optimality**
- ❑ **(Usually there are also non-timeliness components in the optimality criterion, such as relative importance, precedence constraints, resource dependencies, etc.)**

The first dimension of sequencing optimality criteria is optimality of collective timeliness

- ❑ **Collective timeliness optimality is the merit of a sequence of time-constrained thread executions**
 - i.e., the resulting timeliness of all those threads, with respect to an application-specific criterion
- ❑ **Every possible sequence of thread executions has an optimality value according to this metric**
- ❑ **Optimum collective timeliness is a rare special case**

Hard and soft real-time are distinguished by their sequencing optimality criteria

- ❑ **Hard real-time sequencing has only one timeliness component in its optimality criterion**
 - **i.e., always meet all hard deadlines**
 - **which specifies that only maximum optimality is acceptable**
- ❑ **Soft real-time sequencing includes all other possible timeliness components in sequencing (usually scheduling) optimality criteria – very common examples include**
 - **minimizing the number of missed deadlines**
 - **minimizing the mean tardiness**
- ❑ **Hard and soft real-time are not distinguished by optimality**
 - **hard real-time sequences do have maximum optimality**
 - **soft real-time sequences may have maximum optimality**

The second dimension of sequencing optimality criteria is predictability of collective timeliness

- ❑ 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 $C_n = \text{variance}/\text{mean}^2$
 - 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 $C_n = \infty$

These two dimensions of sequencing optimality criteria are orthogonal

- ❑ The collective thread execution sequence optimality may be
 - deterministically optimal (hard real-time)
 - deterministically sub-optimal (soft real-time)
 - non-deterministically optimal (soft real-time)
 - non-deterministically sub-optimal (soft real-time)

Sequence optimality and predictability generally must be traded off

- ❑ The two dimensions of collective timeliness – optimality and predictability of optimality – generally must be traded off against one another on an application-specific basis, to adapt to circumstances
- ❑ For example, it may be necessary to choose between
 - one sequence which results in
 - better optimality (e.g., lower mean tardiness)
 - with worse (higher) coefficient of variation
 - another sequence which results in
 - worse optimality (e.g., higher mean tardiness)
 - with better (lower) coefficient of variation
- ❑ Hard real-time is a special case where this trade-off is not necessary

A scheduling algorithm or dispatching rule is selected to satisfy the optimality criterion

- ❑ Given a collective timeliness sequencing (e.g., a thread scheduling) criterion, a suitable sequencing algorithm is chosen or devised
- ❑ When scheduling is employed, the algorithm is usually called a *scheduling algorithm*, and when only dispatching is employed, the algorithm is usually called a *dispatching rule*
- ❑ There may be more than one algorithm for a given criterion – e.g., the hard real-time criterion is met by the earliest deadline first (EDF) algorithm, and by the least laxity first algorithm, among others
- ❑ Conversely, a specific algorithm may be suitable for different criteria – EDF meets the hard real-time criterion, and also satisfies the soft real-time criterion “minimize maximum lateness”

Outline

- ❑ Java and “real-time Java”
- ❑ Sun’s Java Community Process
- ❑ JSR-1 the Real-Time Specification for Java
- ❑ JSR-50 the Distributed Real-Time Specification for Java
- ❑ A real-time lexicon
- ❑ **Categories of “distributed systems”**
- ❑ DRTSJ: the initial approach
- ❑ The *activity* abstraction
- ❑ Activity-based programming models: past, present, and future
- ❑ Conclusion

A distributed system has application entities that exhibit trans-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 trans-node behaviors
- ❑ Each of these trans-node behaviors has end-to-end 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 trans-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 may provide more than one programming model, but usually only one programming model is the first class abstraction, and the 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, and an optional data flow programming model as a service layered on top of the control flow model**
- ❑ 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**

Outline

- ❑ Java and “real-time Java”
- ❑ Sun’s Java Community Process
- ❑ JSR-1 the Real-Time Specification for Java
- ❑ JSR-50 the Distributed Real-Time Specification for Java
- ❑ A real-time lexicon
- ❑ Categories of “distributed systems”
- ❑ **DRTSJ: the initial approach**
- ❑ The *activity* abstraction
- ❑ Activity-based programming models: past, present, and future
- ❑ Conclusion

A real-time distributed system has acceptable timeliness of end-to-end application behaviors

- ❑ The defining characteristic of any real-time distributed computing system, whatever its programming model, is that**
 - the end-to-end timeliness (optimality and predictability of optimality) of trans-node application behaviors**
 - is acceptable to that application**
 - under the current circumstances**

A trans-node behavior'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 trans-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.)

consistently on each node involved in that application trans-node behavior

Trans-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 must propagate timeliness properties when RMI's take place

- In real-time distributed Java systems,
a trans-node application behavior's end-to-end timeliness
(and other) properties must be**
 - acquired**
 - propagated**
 - and deposited****if and when 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 consistently 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 proposes to first apply the RMI enhancement to a control flow programming model

- ❑ Another objective of JSR-50 is apply the RMI enhancement mechanism to support a control flow programming model**
- ❑ More specifically, control flow in this proposal means**
 - an individual program can be distributed in the sense that one or more of its computational entities can execute across physical node boundaries**
 - one or more causally ordered sequences of constituent operations occurring on multiple nodes**
 - each via a sequence of method invocations/returns**
 - instead of being confined to a single node or a single object instance on a node, as a conventional thread is**
- ❑ Well known control flow programming models include CORBA and DCOM/COM+**

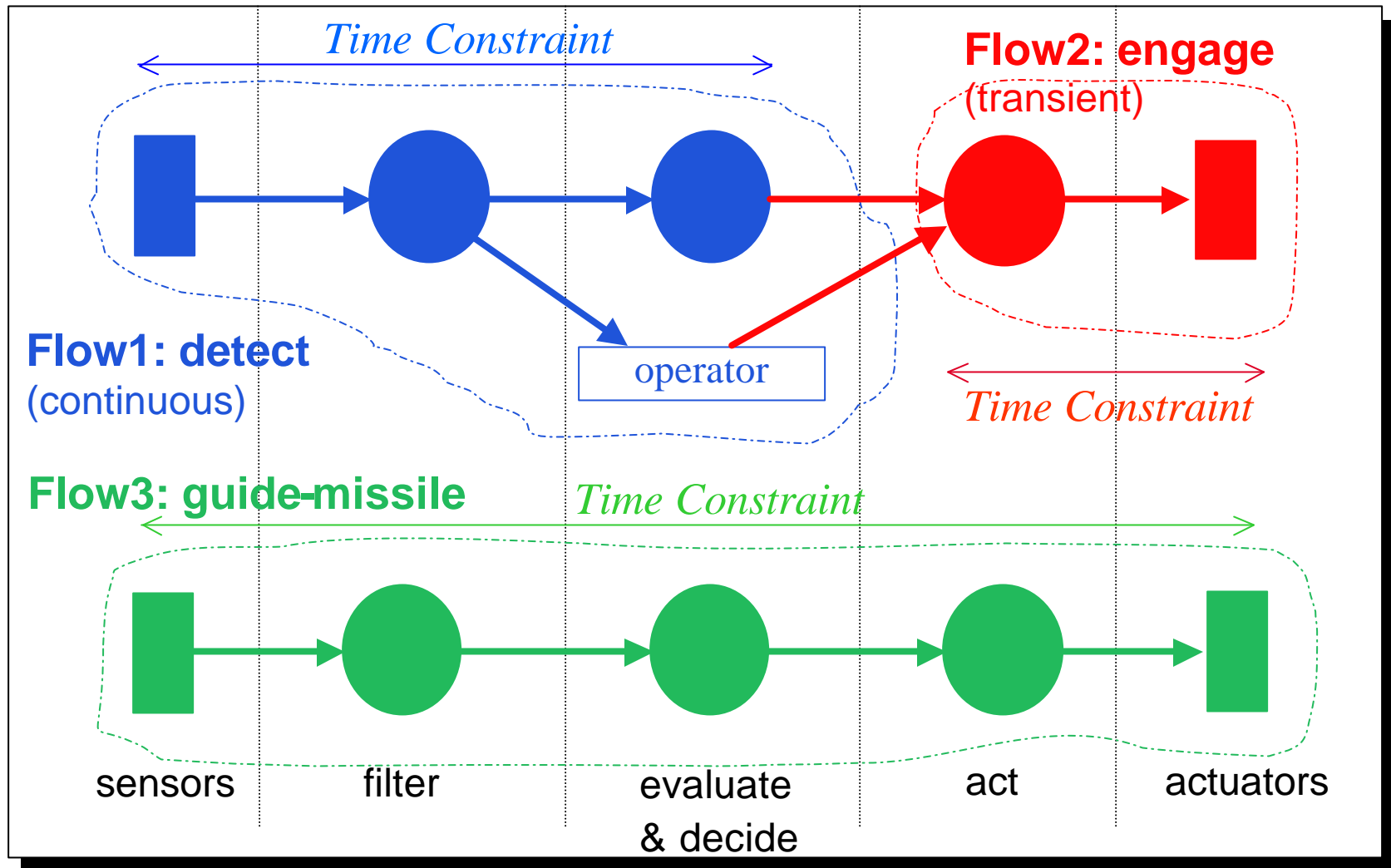
Control flow has compelling advantages as a native model for distributed real-time software

- ❑ **Distributed control flow is a natural, well-understood, incremental extension to local control flow (procedure calls)**
- ❑ **Familiarity is an especially important factor in attaining adoption in real-time application domains, such as industrial automation, defense, and telecommunications**
- ❑ **Data flow per se typically doesn't include resource management for end-to-end properties, such as timeliness**
- ❑ **Data-flow abstractions can often be built cost-effectively using control flow abstractions, while the converse is usually semantically difficult**
- ❑ **The network model leaves too much work for the application programmers**

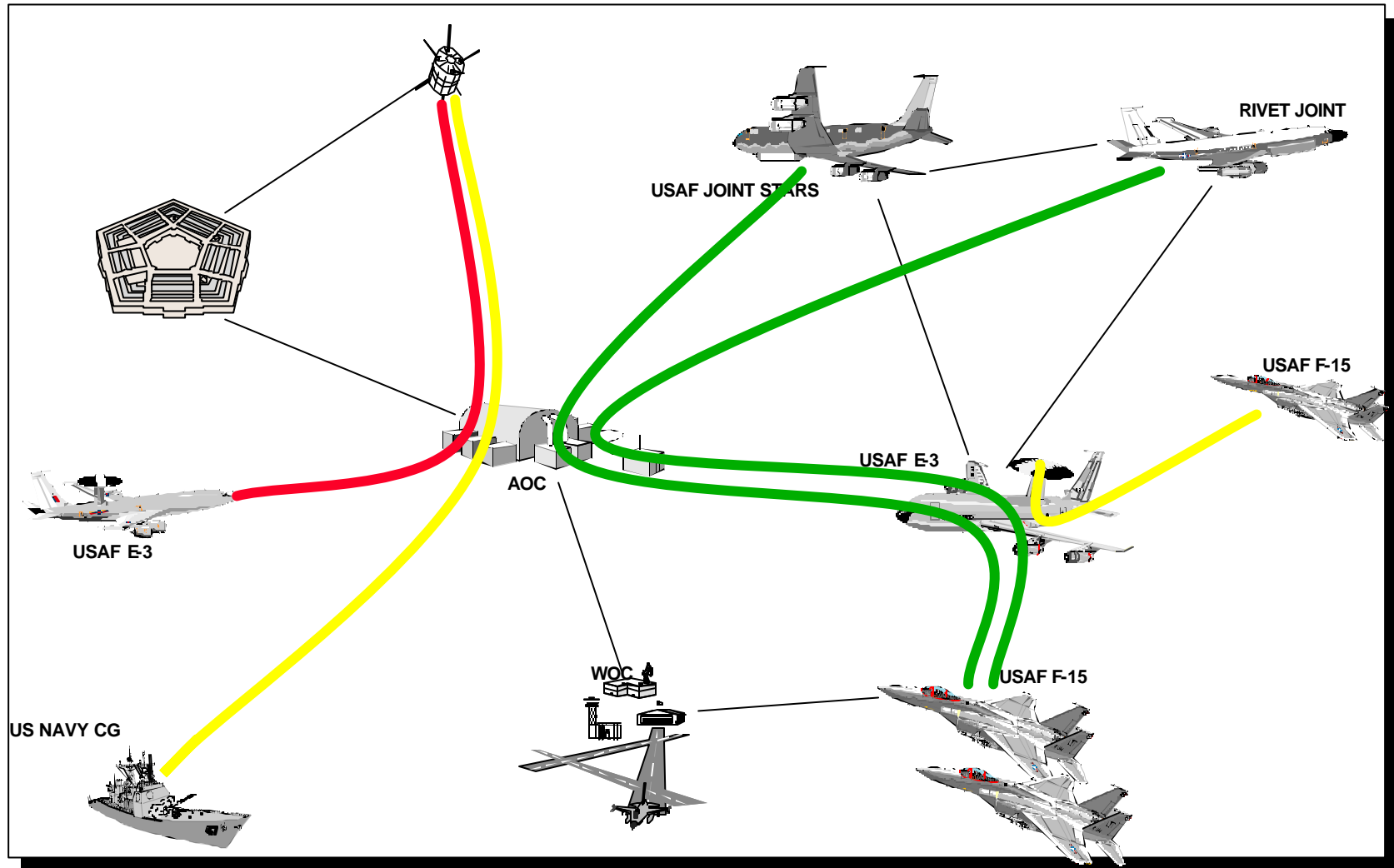
Control flow method invocations (and returns) are location-independent, and other code is not

- 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



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 thus nodes)**
- ❑ **An activity can extend and retract its locus of execution point movement among a program's constituent methods in object instances that may be dispersed across a multiplicity of physical computing nodes, by location-independent method invocations and returns**
- ❑ **Normally a control flow program would be thought of as consisting of multiple concurrent activities**

An activity is a sequential entity and does not do “send/waits”

- ❑ The synchrony of a conventional operation invocation (or RPC) programming model is often cited as a concurrency limitation**
- ❑ But that criticism does not apply to the activity model**
- ❑ An activity is a sequential abstraction, 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 operation invocations**
- ❑ Remote invocations and returns are scheduling events at both client and servant nodes**
- ❑ Each node’s processor is always executing the most eligible activity there; the others there wait, as they should**

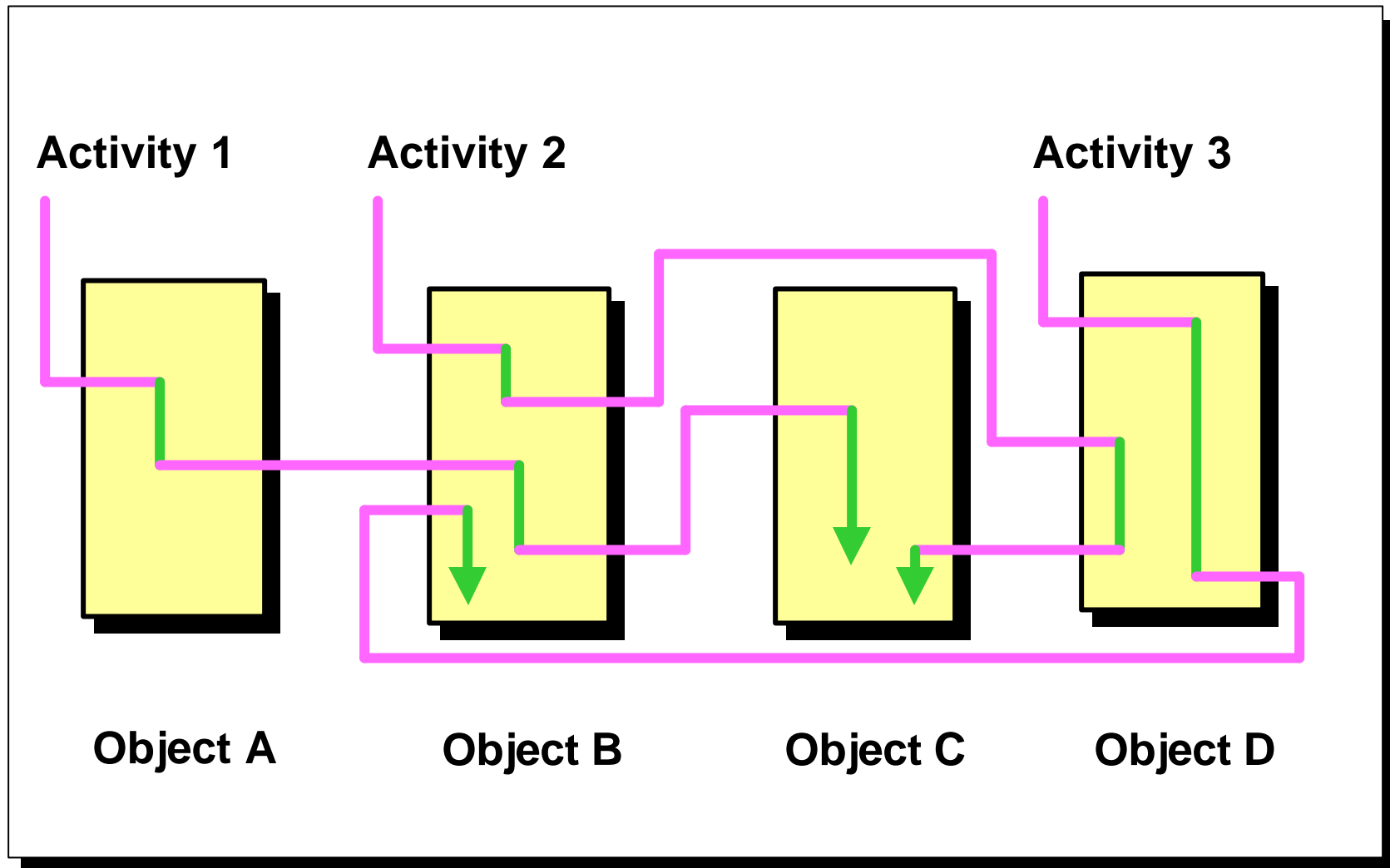
An activity always has exactly one execution point in the whole system

- ❑ New activities can be created, or sleeping ones awakened, when needed – e.g., for branches of control flow**
- ❑ An application or system may consist of multiple activities**
- ❑ Multiple activities execute concurrently and asynchronously, by default**
- ❑ Activities synchronize through method execution – the writers of each object control activity concurrency in that object**

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

- ❑ The activity abstraction is implemented using local (RTOS or JVM) threads**
- ❑ In Java, an activity would be implemented by the concatenation of local (per node) threads sequentially performing RMI's when they transit nodes**

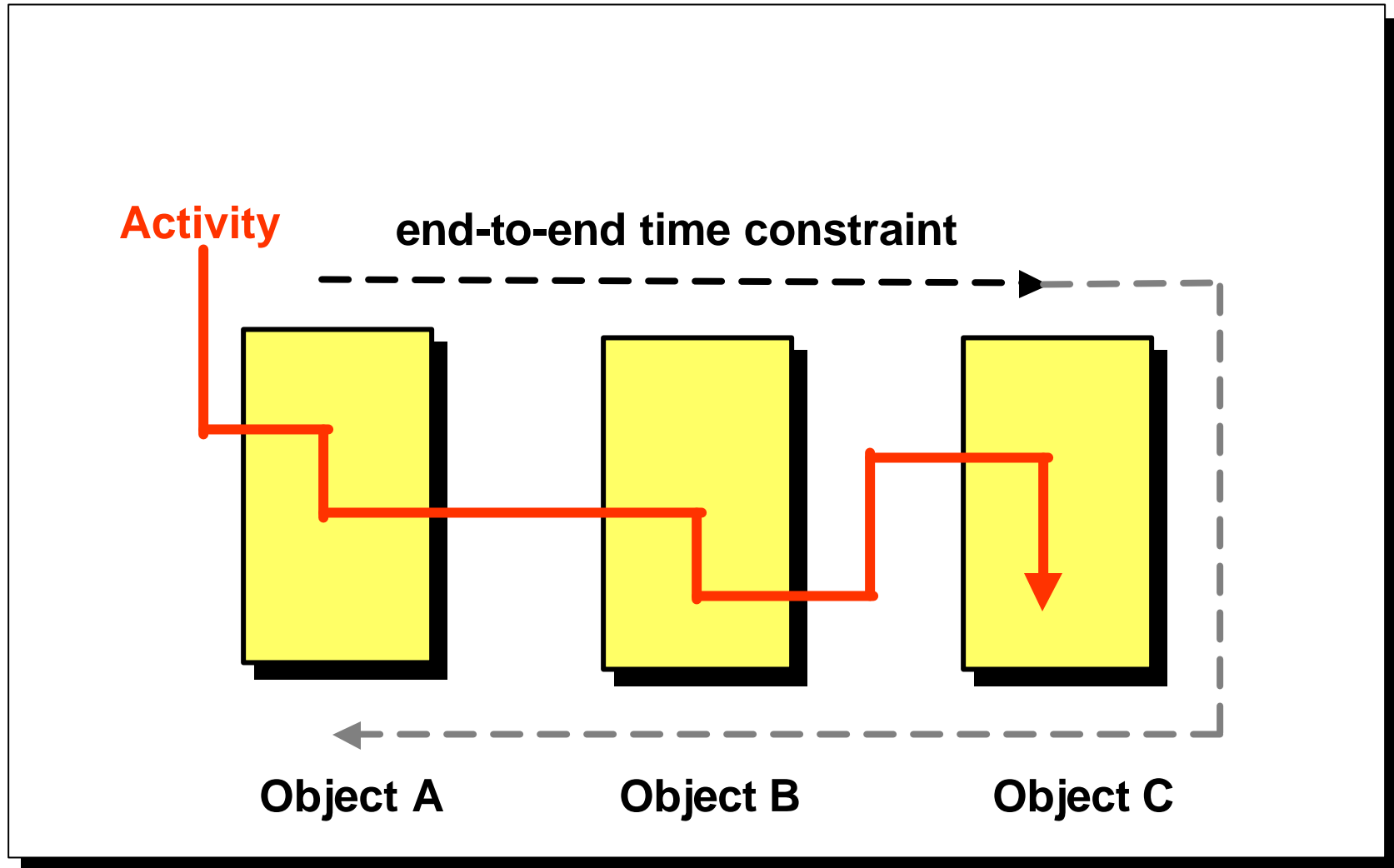
An activity is built using local (e.g., RTOS) threads and method invocations



An activity has end-to-end timeliness attributes

- ❑ Each activity may have execution scheduling attributes – e.g., time constraints, 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 activity is governed by those scheduling parameters, according to the scheduling policy, regardless of the activity's execution point transiting nodes**
- ❑ Any of the three distributed scheduling approaches can be used**
- ❑ When it transits a node boundary, its timeliness parameters are propagated to the remote scheduling policy instance**
- ❑ When it returns, updated timeliness parameters are propagated back to invoker's scheduling policy instance**

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



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 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 activity abstraction supports application timeliness requirements everywhere in that two-dimensional predictability/time-frame space of distributed real-time Java systems**

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

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

Outline

- ❑ Java and “real-time Java”
- ❑ Sun’s Java Community Process
- ❑ JSR-1 the Real-Time Specification for Java
- ❑ JSR-50 the Distributed Real-Time Specification for Java
- ❑ A real-time lexicon
- ❑ Categories of “distributed systems”
- ❑ DRTSJ: the initial approach
- ❑ The *activity* abstraction
- ❑ **Activity-based programming models: past, present, and future**
- ❑ Conclusion

Activity-based programming models: past, present, and future

- ❑ The activity abstraction originated as *distributed threads* in the Alpha real-time distributed OS kernel at CMU CS (which included all the additional facilities)
- ❑ A relatively large experimental defense application at General Dynamics used it
- ❑ A second generation of Alpha was produced at KSR and Concurrent
- ❑ A B3 class multilevel secure version was created at SRI, and the technology was transitioned into a product by TIS
- ❑ Microsoft Research adapted Alpha's real-time and distributed thread technologies for their Rialto multimedia OS
- ❑ U of Utah transitioned distributed threads into the Mach 3 and Flex microkernels

Activity-based programming models: past, present, and future (continued)

- ❑ IBM and DEC merged Alpha's real-time and distributed thread technologies into OS/2 for PowerPC (not released)
- ❑ DEC used Alpha's real-time and distributed thread technologies in a proprietary middleware product and then a real-time DCOM, together with a Win32 RTOS (not released)
- ❑ The OSF/RI merged Alpha's real-time and distributed thread technologies into their Mach 3-based MK7 OS (which is the basis for a forthcoming commercial product from a major computer company)
- ❑ MITRE/OSF built a significant experimental application using MK7
- ❑ The DARPA/ITO Quorum program's *paths* and *utility functions/regions* are derived in part from Alpha concepts

Activity-based programming models: past, present, and future (continued)

- ❑ **OMG's Real-Time CORBA 1.0 specification includes activities as a design concept, but left the details undefined**
- ❑ **The current proposed Real-Time CORBA "2.0" specification is based on a *distributable thread* programming model (but the additional facilities are deferred to a subsequent spec)**
- ❑ **It also includes the Alpha real-time scheduling technology**

Conclusion

- ❑ **Real-time computing is about predictability of timeliness**
- ❑ **Distributed real-time computing is about predictability of timeliness of end-to-end behaviors**
- ❑ **Acceptable predictability of timeliness of end-to-end 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 end-to-end timeliness predictability needs**