

# Hard Real-Time: C++ versus RTSJ

Daniel L. Dvorak and William K. Reinholtz

Jet Propulsion Laboratory  
California Institute of Technology  
4800 Oak Grove Drive  
Pasadena, California, 91109 USA  
1-818-393-1986, 1-818-354-6419

{daniel.l.dvorak, william.k.reinholtz}@jpl.nasa.gov

## ABSTRACT

In the domain of hard real-time systems, which language is better: C++ or the Real-Time Specification for Java (RTSJ)? Although standard Java provides a more productive programming environment than C++ due to automatic memory management, that benefit does not apply to RTSJ when using `NoHeapRealtimeThread` and non-heap memory areas. As a result, RTSJ programmers must manage non-heap memory explicitly. Although that's a common practice in real-time applications, it's also a common source of programmer error, regardless of language. In an ironic role reversal, this paper shows that C++ is able to provide a safer programming environment than RTSJ (or C) for managing memory in a hard-real-time producer/consumer pattern. C++ accomplishes this via a reference-counting pointer. RTSJ (and C) cannot provide an equivalent mechanism because it lacks the necessary language features. Despite other attractive features of RTSJ, the relative simplicity and safety of the C++ programming model for this common pattern suggests that C++ will be a strong competitor to RTSJ in the domain of real-time mission-critical systems.

## Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *classes and objects, control structures, dynamic storage management, frameworks*

## General Terms

Design, Reliability, Languages.

## Keywords

Programming model, architecture, concurrency, real-time.

## 1. INTRODUCTION

Since its emergence in 2000, the Real-Time Specification for Java [1] (RTSJ) has generated considerable interest because it enables real-time applications to be programmed in the popular Java programming language, with no syntactic changes to the language. Real-time facilities are provided via APIs whose real-time properties are provided by a modified JVM. Given Java's popularity and productivity advantage relative to C++, plus its attention by both the research community and tool vendors, the RTSJ has the potential to

become the language of choice for real-time applications, displacing C/C++.

To practitioners, the appeal of one programming language versus another depends in part on the ease and naturalness of designing and developing code for common tasks. A "programming model" includes the things that a programmer must know and consider at design time and what he/she must actually code. In the case of RTSJ and C++ there is a notable difference in programming models for hard real-time applications. By "hard real-time" we mean that timing requirements must *always* be met; execution must be *predictable* in that real-time tasks are released on schedule and complete within their deadlines. Failure to do so can be as serious as an error in program logic. Hard real-time is not the same as "real fast", though that's sometimes necessary in order to meet performance requirements.

This report illustrates differences between RTSJ and C++ in how each language implements a real-time producer/consumer pattern. This pattern is common in real-time control systems and it happens to expose fundamental language differences that affect program simplicity and safety. Our comparison shows that the C++ programming model is simpler and safer (in this particular case) because C++ contains features that relieve the programmer from worrying about when to release objects back to a memory pool. Neither Java nor RTSJ provide the necessary features, so RTSJ programmers are forced to explicitly program the release of such objects.

It's important to note that the RTSJ design in this comparison is based on the use of `NoHeapRealtimeThread` and non-heap memory. Such threads achieve guaranteed timing behavior since garbage collection (GC) activities cannot interfere with them. However, the emergence of real-time garbage collectors for Java [2,3,4] offers an attractive alternative for real-time applications, provided that they are able to tolerate both the GC-induced preemption latencies and the GC processing cost.

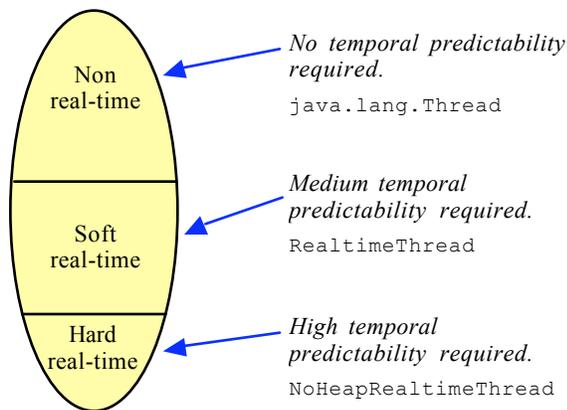
## 2. RTSJ BACKGROUNDER

Ordinary Java technology is not suitable for real-time systems for several reasons: no scheduling control over threads, unpredictable synchronization delays, run-anytime garbage collection, coarse timer support, no event processing, and no safe asynchronous transfer of control. The real-time specification for Java, known as "RTSJ", addresses these limitations through several areas of enhanced semantics.

The RTSJ was shaped by several guiding principles. Foremost among these is the principle to “hold predictable execution as first priority in all tradeoffs”. Another principle is that the RTSJ introduces no new keywords of other language extensions. Also, the RTSJ provides backward compatibility, meaning that existing Java programs run on RTSJ implementations. Importantly, the RTSJ APIs support modern scheduling policies, such as Earliest Deadline First, in addition to conventional priority-based scheduling.

It’s important to understand that “real time” doesn’t mean “real fast”. The guiding principle of predictable execution places more importance on specifying and meeting timeliness constraints than on raw throughput. Real-time applications must respond to periodic and sporadic events, and the RTSJ provides facilities for informing a scheduler of such constraints and determining if a set of constraints admits a feasible schedule. The net result in RTSJ, in contrast to purely priority-based systems, is that scheduling and dispatching can be based on explicit timeliness information.

Most real-time applications are a mixture of “hard real-time”, “soft real-time”, and non real-time parts, as shown in Figure 1. In this report we use the term “hard real-time” to mean that temporal correctness criteria must always be met. For example, if a hard real-time computation misses a deadline, the system goes into an abnormal state. By “soft real-time” we mean that temporal correctness criteria are almost always met, so an occasional missed deadline (for example) is tolerated. By “non real-time” we mean that there are no temporal correctness criteria. A key point here is that a single RTSJ-compliant VM can support systems that mix hard, soft, and non real-time parts.



**Figure 1. Most real-time systems are a mixture of hard real-time, soft real-time, and non-real-time, all of which can be supported by a single RTSJ-compliant VM.**

The RTSJ extends Java semantics in several areas, as summarized below. This background information is intended to provide readers with a broad understanding of how the RTSJ supports various aspects of real-time programming. Some features of the RTSJ have been omitted for brevity.

## 2.1 Threads

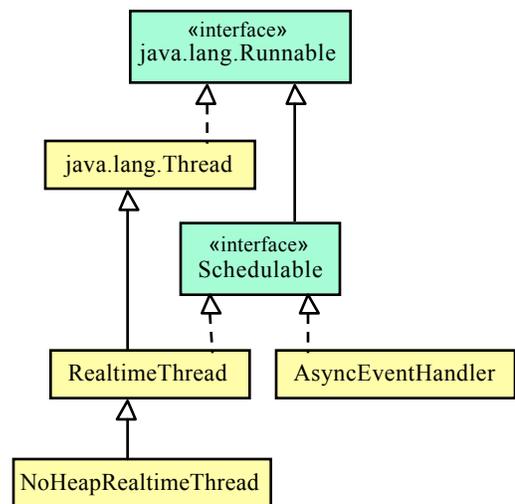
The RTSJ introduces two new types of thread that have more precise scheduling semantics than `java.lang.Thread`. Parameters provided to the constructor of `RealtimeThread`

allow the temporal and processor demands of the thread to be communicated to the system. `NoHeapRealtimeThread` (“NHRT”) extends `RealtimeThread` with the restriction that it is not allowed to allocate or even reference objects from the Java heap, and can thus safely execute in preference to the garbage collector. Such threads are the key to supporting hard real-time execution because they have implicit execution eligibility logically higher than any garbage collector.

## 2.2 Scheduling

The scheduling area in RTSJ provides classes that allow the definition of schedulable objects, manage the assignment of execution eligibility of schedulable objects, assign “release characteristics” to schedulable objects, and perform “feasibility analysis” for sets of schedulable objects.

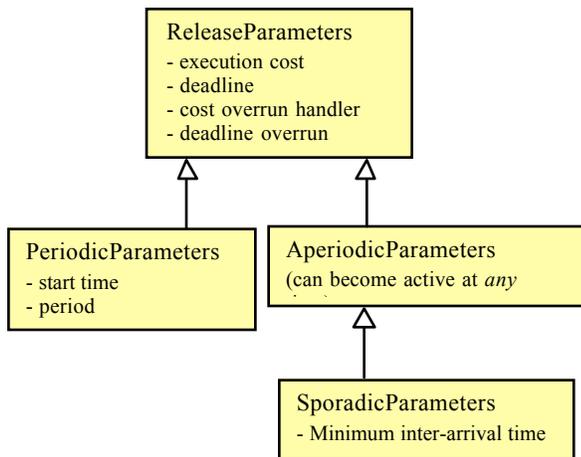
As seen in Figure 2, schedulable objects are instances of `RealtimeThread`, `NoHeapRealtimeThread`, and `AsyncEventHandler`. Each of these is assigned processor resources according to its release characteristics and execution eligibility. As shown in Figure 3, there are three types of `ReleaseParameters` to support periodic, aperiodic, and sporadic execution. Each of these subclasses contains parameters needed to determine whether a feasible schedule can be found for a set of schedulable objects.



**Figure 2. The RTSJ introduces RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler as new types of Runnable.**

## 2.3 Memory Management

The RTSJ contains classes that allow the definition of regions of memory outside the traditional Java heap. These new memory areas—called `ImmortalMemory` and `ScopedMemory`—are not managed by a garbage collector. This means that instances of `NoHeapRealtimeThread` can use such memory to communicate results within hard real-time areas as well as between hard real-time areas and soft- or non real-time areas.

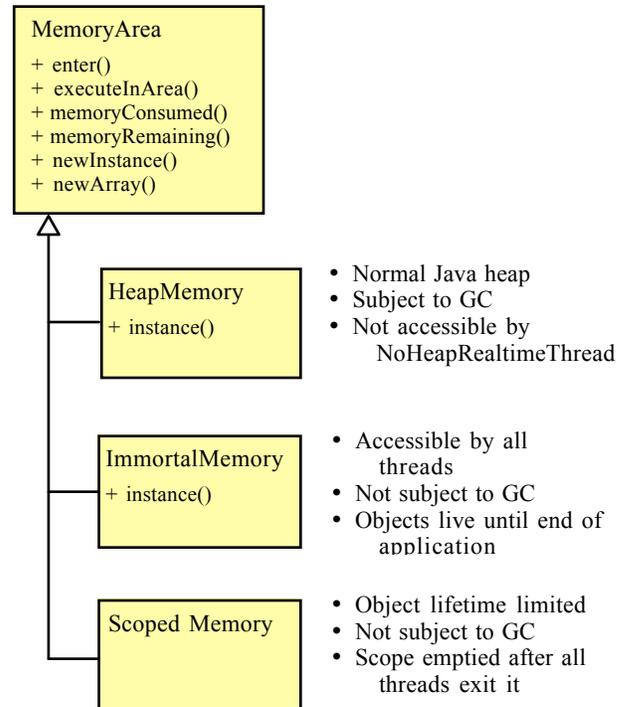


**Figure 3. Release parameters supply processor and temporal demands needed to determine schedule feasibility.**

ImmortalMemory is a single memory area that is shared among all threads. Objects allocated in the immortal memory live until the end of the application. In fact, unlike standard Java heap objects, immortal objects continue to exist even after there are no other references to them. Importantly, objects in immortal memory are never subject to garbage collection.

ScopedMemory is an abstract base class for memory areas having limited lifetimes. A scoped memory area is valid as long as there are real-time threads with access to it. A reference is created for each accessing thread when either a real-time thread is created with a ScopedMemory object as its memory area, or when a real-time thread runs the enter() method for the memory area. When the last reference to the object is removed, by exiting the thread or exiting the enter() method, finalizers are run for all objects in the memory area, and the area is emptied. Objects in scoped memory are never subject to garbage collection.

The memory management enhancements in RTSJ also include facilities for access to physical memory, facilities for non-heap memory allocation in linear time, and facilities for obtaining information about the temporal behavior of the garbage collector, such as its preemption latency.



**Figure 4. The RTSJ introduces two kinds of non-heap memory that are not subject to garbage collection.**

## 2.4 Synchronization

The RTSJ contains classes that allow application of the priority ceiling emulation algorithm to individual objects; allow the setting of the system default priority inversion algorithm; and allow wait-free communication between real-time threads and regular Java threads. This strengthens the semantics of Java synchronization for use in real-time systems by mandating priority inversion control. The wait-free queue classes provide protected, concurrent access to data shared between instances of java.lang.Thread and NoHeapRealtimeThread.

## 2.5 Time

The RTSJ contains classes that allow description of a point in time with up to nanosecond accuracy and precision (dependent on the precision of the underlying system), and allow distinctions between absolute points in time, times relative to some starting point, and rational time, which allows the efficient expression of number of occurrences per some interval of relative time.

The time class relationships are depicted in Figure 6. Instances of AbsoluteTime represent absolute time expressed relative to midnight January 1, 1970 GMT. Instances of RelativeTime encapsulates a point in time that is relative to some other time value. Instances of RationalTime express a frequency as an integral number of cycles per an amount of relative time.

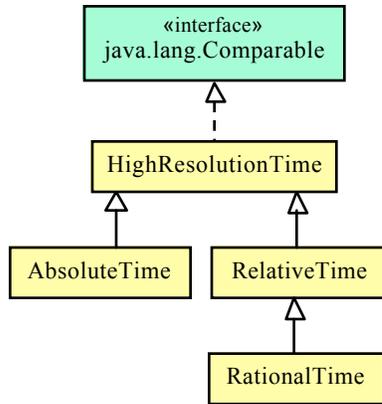


Figure 6. High resolution time supports timing with nanosecond accuracy and precision, subject to the underlying system’s accuracy and precision.

## 2.6 Timers

The RTSJ contains classes that allow creation of timer whose expiration is either periodic (`PeriodicTimer`) or set to occur at a particular time (`OneShotTimer`). RTSJ also defines an abstract base class for clocks, recognizing that real systems often have other kinds of clocks (e.g. simulation clocks, user time clocks), and allows timers to specify such a clock in place of the default system clock.

## 2.7 Asynchrony

The RTSJ contains classes for binding the execution of program logic to the occurrence of internal and external events. Specifically, an asynchronous event is represented as an instance of class `AsyncEvent` or a subclass. An event occurrence may be initiated by application logic (by invoking the event instance’s `fire()` method) or by the occurrence of a “happening” that is external to the JVM, such as a hardware interrupt.

Each instance of `AsyncEvent` may have one or more instances of `AsyncEventHandler` associated, as shown in Figure 7. The converse also holds: every instance of `AsyncEventHandler` may have one or more instances of `AsyncEvent` associated. Every time an event occurs, the associated handlers are made eligible to run; dispatching of the handler is subject to its release parameters.

<code>java.lang.Thread</code>	Yes	No	No
<code>RealtimeThread</code>	Yes	Yes	Yes
<code>NoHeapRealtimeThread</code>	No	Yes	Yes

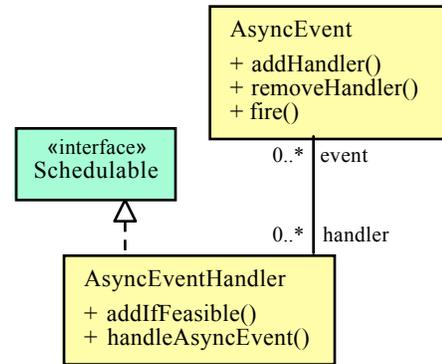


Figure 7. Asynchronous events and their handlers can have a many-to-many relationship in the RTSJ.

## 3. PROBLEM DOMAIN

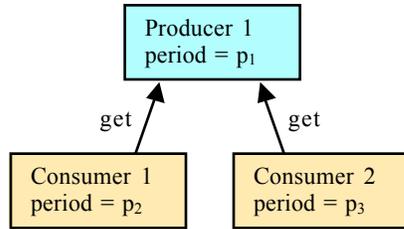
Many real-time applications are “control systems”, i.e., they interact with the real world through sensors and actuators to control some physical system, whether it be a rover on the surface of Mars or the powertrain in your automobile. In either case, these systems are designed for continuous operation and employ “closed-loop” (feedback) control, usually with time-critical requirements. For example, control of the driving motors on one of the experimental Mars rovers runs at a 200 Hz rate. Such control systems are termed “hard real-time” to the extent that any failure to satisfy the timeliness requirements puts the system into an abnormal state.

Control systems contain many real-time producer-consumer relationships. For example, one task acquires data from sensors; another task gets that data and computes new state estimates; another task gets a state estimate and computes a control response; another task issues the control commands to actuators. In each such relationship data must flow from producer to consumer, and the data is often a structure or object, not a primitive type. Also, much of the data is of fleeting importance, to be replaced by newer data on the next cycle of execution.

Efficient use of memory is a hallmark of many real-time systems. Usually they cannot afford the runtime overhead of general-purpose memory allocation, such as `malloc` or `new`, so they manage pools of pre-allocated objects, recycling the objects continually via code inside consumers that release objects back to a pool after using them. In a perfect world this works very well, but history has shown this to be a common source of error. Programmers may neglect to release an object, particularly in a rarely used branch of code, or may release an object but continue to use it. Either way, the results can be disastrous. For this reason, the programming model for working with memory pool objects is a major point of comparison in the following sections.

## 4. PRODUCER/CONSUMER PATTERN

This paper compares RTSJ to C++ in terms of how simply and safely each language handles a common task in real-time applications, namely, data-handling in a real-time producer/consumer relationship, as shown in Figure 8. Several requirements and constraints apply: there is one producer and multiple consumers; both producer and



**Figure 8. This real-time producer-consumer pattern has one producer and multiple consumers, each running with potentially different periods. A consumer gets data from a producer.**

consumer(s) are periodic threads; the producer and its consumer(s) may have different periods; all threads must satisfy temporal correctness requirements; the data generated by the producer is immutable (or at least it is not supposed to be modified by any consumer); and producers may generate data structures (not simply a built-in type) that must be conveyed to its consumer(s). This is a “data pull” pattern, meaning that the consumers actively pull the produced data, as needed, whenever needed. In the larger context, a thread can be both a producer and a consumer and therefore may participate in multiple producer-consumer relationships.

In ordinary object-oriented programming a multithread-safe design for conveying data from producer to consumer(s) involves locks. Java is particularly elegant here, not only for the simplicity of its `synchronized` methods, as shown in Figure 9, but also for its automatic memory management that frees the programmer from thinking about when to delete objects that are no longer needed. The programming model is simple and foolproof, but the use of locks for this purpose is undesirable in real-time systems for several reasons. First, locks incur a non-trivial amount of overhead that can slow down high-frequency control loops, even in the absence of contention. Second, when contention does occur, the application incurs the cost of context switching, raising the worst-case execution time of the contending thread by the amount of time needed for the holding thread to complete its

```
// Java code fragment for producer.
// Uses synchronized set and get methods.

class Producer {
    private X latest;

    private synchronized void set(X value) {
        latest = value;
    }

    public synchronized X get() {
        return latest;
    }

    // Producer code that calls set
    ...
}
```

**Figure 9. Synchronized methods in ordinary Java provide an elegant multithread-safe solution for the producer-consumer problem, but not for real-time.**

work and release the lock (this makes it hard to determine worst-case execution time). Third, locks are a source of potential priority inversions and deadlocks, either of which can lead to system failure.

For the reasons just given, the RTSJ and C++ designs shown in following sections do *not* use locks. Instead, both rely on atomic operations to manage the exchange of data in a multi-thread-safe and multi-processor-safe manner. Although the mechanisms are different in the two languages, the differences are not material in this study. The important property is that a producer can safely update its data at any time and consumers can safely read it at any time, all without locks and therefore without the possibility of contention-induced context-switching.

Regardless of language, there are two basic operations that must be designed. First, a producer must generate new data and make it available to consumers without modifying any objects currently in use by any consumer. Second, consumers must somehow release data objects when no longer needed so that the memory can be reused. To minimize defects, such releases should occur implicitly rather than through imperative statements.

## 5. RTSJ IMPLEMENTATION

Given the requirement that producers and consumers must always satisfy temporal correctness requirements, both must use the RTSJ class `NoHeapRealtimeThread`<sup>1</sup>. Threads of this type guarantee highly predictable execution since they can always run in preference to the garbage collector. Such threads cannot reference heap memory, so all data generated by a producer and used by a consumer must be held in scoped memory or immortal memory.

Although scoped memory is intended for temporary objects, such as those passed from producer to consumer, it cannot be used in this situation for at least two reasons. First, the data generated by a producer must always be available for all consumers to read; it cannot be held in a scoped memory area since that scope will have to be emptied on some regular basis. Second, a thread may participate in multiple producer-consumer relationships in multiple control loops, and there is no ordering of scope ‘enter’ calls (in the general case) that satisfies the RTSJ’s single-parent rule for scope stacks.

Given the need to use immortal memory, coupled with the fact that all objects created in immortal memory are, well, immortal, it’s clear that such objects for transient data must be managed and reused. The design therefore relies on the concept of memory pools, where threads may obtain objects, initialize them, use them, and ultimately release them back to the pool.

### 5.1 Producer

A producer is a periodic `NoHeapRealtimeThread` that updates some data on each cycle. Since a producer must not modify an object currently in use by any consumer, the producer must obtain an unused object from the pool, initialize it, and then use an atomic instruction to expose the data to subsequent ‘get’ calls by consumers.

<sup>1</sup> This report focuses on RTSJ mechanisms that are independent of any garbage collector; it does not consider real-time garbage collectors in the solution space.

From the viewpoint of the programming model, writing a producer is familiar territory for Java programmers: call a factory method to obtain and initialize an object from the pool, then store the reference to that object in a field whose value is returned by `get()`. Since assigning a new value to a reference field is an atomic operation in Java, no special care is needed.

## 5.2 Consumer

A consumer is a periodic `NoHeapRealtimeThread` that gets the latest available data from one or more producers and uses it in some way, possibly even to generate new data as a producer to other consumers. In calling a producer's `'get'` method, a consumer obtains a reference to an object that the producer had obtained earlier from a memory pool. Other consumers will obtain a reference to the *same* object if they call `'get'` before the next producer update. Behind the scenes, each time that a consumer obtains an object, that object's usage count is incremented atomically, thus ensuring that the producer will not reuse that object until the usage count returns to zero. It is the responsibility of each consumer to eventually `'release'` the object so that it can be returned to the pool when all consumers are done with it, i.e. when the usage count becomes zero.

```
// RTSJ code fragment for consumer

class Consumer {
    private X data;
    ...
    // get latest data from producer
    data = aProducer.get();

    // access data
    doSomething( data.getValue() );
    ...
    // When done using producer's data,
    // must release object back to its pool.
    data.release();

    ...
}
```

It should be noted that reference counting could be avoided if the producer's `get()` method returned a *copy* of the producer's current data object. In RTSJ, the consumer (the caller) can arrange to be running in a scoped memory area when it calls `get()`. The copy would then be constructed in the consumer's (the caller's) scoped memory area, and used in normal ways. The consumer would be designed to exit the scoped memory area periodically, thereby emptying it. While this approach might be acceptable in some situations, it adds an undesirable runtime overhead, particularly for large objects.

## 6. C++ IMPLEMENTATION

In contrast to RTSJ, C++ has a uniform memory area. Any kind of thread can access any object in heap or static memory. There is no garbage collector to be avoided, there are no non-heap memory areas, and there are no memory area assignment rules. From this viewpoint the programming model is simpler.

A key mechanism that makes the C++ programming model even simpler is a lock-free, thread-safe, multi-processor-safe reference counting pointer (RCP) [5]. The RCP is a template

class that maintains a reference count associated with the pointed-to object (the referent). Each time that an RCP is constructed or assigned, it increments the reference count in the referent. Similarly, each time that an RCP goes out of scope, and is therefore destroyed, it decrements the count in the referent. Also, when an initialized RCP is assigned a new referent, it first decrements the count in the old referent. The net effect is that an object's reference count always equals the number of RCPs pointing to it. When the reference count goes to zero, the referent's `delete()` method is executed. For objects that are designed to live in a memory pool, the `delete()` method is defined to return the object back to the pool's freelist. The net result is a simpler programming model in which C++ programmers don't have to worry about manual memory management. The RCP's implementation depends on processor-atomic instructions, as detailed in [5].

## 6.1 Producer

A producer is a periodic Posix thread that updates its data on each cycle. Like the RTSJ producer, it obtains an object from a pool, initializes it, and then atomically exposes it as the object to be returned by subsequent calls to its `'get'` method. The only difference is that the C++ code manipulates reference-counting pointers rather than Java references. Otherwise, the two producers are identical in terms of simplicity and safety.

## 6.2 Consumer

A consumer is a periodic Posix thread that gets the latest data from one or more producers. The producer's `'get'` method returns a RCP to a `'const'` object that the consumer typically assigns to a local RCP. At some later time, when the consumer is done using the referent, it may either let the RCP go out of scope (i.e. let it pop off the stack) or it may reassign the RCP with a new value from the producer. In either case, the referent's reference count is decremented automatically and the object is returned back to the memory pool when the count goes to zero.

```
// C++ code fragment for consumer

class Consumer {
    private RCP<const X> rcp;
    ...
    // get ptr to latest data from producer
    rcp = aProducer.get();

    // access data by dereferencing the RCP
    doSomething( rcp->getValue() );
    ...
    // No explicit release needed. When RCP
    // goes out of scope or is reassigned,
    // it automatically decrements reference
    // count of referent.
    ...
}
```

## 6.3 Limitation

A well-known limitation of reference counting is that it's not suitable for circularly linked data structures. In our experience, such data structures don't appear in real-time control loops, so this has not been a limitation in practice.

## 7. COMPARING RTSJ AND C++

The two implementations of the real-time producer-consumer pattern highlight some interesting differences between the RTSJ and C++ — differences that affect the programming model. It is in the consumer that the C++ implementation can be seen as clearly simpler and safer than in RTSJ, where explicit releases must be programmed. First, the C++ programmer does nothing special to release objects back to the pool; it happens automatically and immediately because C++ destructors run whenever a stack object is popped, and it happens automatically and immediately when a RCP object is assigned, since it has an appropriately overloaded assignment operator. The RTSJ design is vulnerable to two well-known bugs: memory leaks due to failure to release, and consumer use of an object after it was released. It's ironic that C++ is able to provide an automatic memory management mechanism in this situation whereas RTSJ (and Java) cannot.

Second, a producer in C++ is able to protect its data against illegal mutation by consumers since C++ is able to return a pointer to a `const` object. Since `'const'` is a type qualifier, any attempt to violate `'constness'` is detected at compile time. Java (and thus RTSJ) cannot return a "reference to a constant object". Again, the C++ consumer is easily protected against a kind of bug (illegal mutation of data) that can be extremely hard to debug.

To be fair, RTSJ has advantages of its own, relative to C++. Most important for real-time programming is the scheduling API, where temporal correctness criteria are specified explicitly. For example, RTSJ's *release parameters* specify start time, execution cost, period, deadline, and minimum interarrival time. These parameters are checkable at design time in a feasibility analysis, and run-time violations will trigger cost overrun handlers, deadline miss handlers, and minimum interarrival time exceptions. Another feature enables programmers to separate the execution costs of "normal case: do a little work" versus "error case: do a lot of work". This enables more effective use of CPU cycles since the scheduler does not have to reserve so much time for rare-but-costly error cases.

## 8. EPILOGUE

Although it's interesting to compare C++ and RTSJ, the longer-term debate shouldn't be about these specific languages. The bigger problem is that real-time applications coded in either language embed several 'commitments' that are hard to change in the middle of a project. These code-level commitments include choices such as: memory pools or not, scoped vs. immortal memory, partitioning of functionality among threads, locks that may be unnecessary (depending on execution model choices), and thread priorities. The problem is that these choices are incidentals, not essentials; they don't express or reveal the underlying requirements; they are a means to an end. What we *really* want is to specify the essentials explicitly and let an analyzer generate the language- and platform-specific incidentals that will satisfy the requirements — or tell us that no feasible solution exists. The essentials are: data structures for inputs and outputs, pure functions that perform state transformations, and required properties of sequencing,

timing, concurrency, etc. These concepts relate directly to the problem domain and are neutral with respect to language, software architecture, and hardware architecture. Consequently, they leave options open late in the development and testing cycle, rather than making early (and sometimes regrettable) commitments that can only be changed at great cost.

Do we still care about programming languages? Yes, because the vision described above is still a topic of research. Perhaps the most important "take away" message for practitioners here is to maintain a clear mental separation between essentials (things dictated by the problem domain) and incidentals (choices about how to choreograph the real-time execution of many pieces of functionality). The more you can do to keep the two concepts separated in the code, the more flexibility you'll have in responding to late-breaking changes in requirements and negative surprises in performance testing.

## 9. ACKNOWLEDGEMENTS

This work was performed jointly by the Jet Propulsion Laboratory of California Institute of Technology, by Sun Microsystems Laboratory, and by Carnegie Mellon University. The work at JPL was performed under contract with the National Aeronautics and Space Administration. The JPL team thanks the Office of the Chief Scientist for funding under the Research & Technology Development Program, and the strong support of the R&TD steering committee on the Software Techniques & Methods Initiative.

## 10. REFERENCES

- [1] Bollella, G. et al, The Real-Time Specification for Java, Addison-Wesley, 2001. <http://rtj.org>
- [2] Bacon, D., Cheng, P., and Rajan, V. The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems. Workshop on Java Technologies for Real-Time and Embedded Systems (Catania, Sicily, November 2003), in *Proceedings of the OTM Workshops, R. Meersman and Z. Tari, eds., Lecture Notes in Computer Science vol. 2889*, pp. 466-478.
- [3] Siebert, F. The Impact of Realtime Garbage Collection on Realtime Java Programming. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, (Vienna, Austria, May 12–14, 2004).
- [4] Nilsen, K. Doing Firm-Real-Time with J2SE APIs. Workshop on Java Technologies for Real-Time and Embedded Systems (Catania, Sicily, November 2003), in *Proceedings of the OTM Workshops, R. Meersman and Z. Tari, eds., Lecture Notes in Computer Science vol. 2889*, pp. 371-384.
- [5] Reinholtz, William K. A Lock-Free, Async-Safe, Thread Safe, and Multi-Processor Safe Reference Counting Pointer. To appear in *C/C++ Users Journal*