

Composability Problems and Mitigation in Real-Time Software Defined Radio (SCA) Systems

David F. Carlson
Chronolytics Inc.

Raymond Lindenmeyer
Harris Corp.

Abstract

A goal of the Software Communication Architecture (SCA) is to allow software radios to be built from "components" supplied by multiple vendors. The SCA Core Framework (CF) provides the software backplane which is designed to accommodate plug-in components and whole applications called "waveforms". A key property of software components is that they be "composable". In this paper we will show that reasonable implementation of the CF specification contributes to the situation that each integration of real-time components may require a large amount of non-recurring engineering (NRE) to "fix" the Core Framework/Operating Environment (and applications) in order to meet real-time requirements. The lack of real-time composability may make the use of multiple vendors for software components more difficult and more costly for the integrator. In particular, how a reasonable implementation of the CF using synchronous method invocation (SMI) CORBA and the "thread & lock" model effectively prevent composability.

This paper will present an alternative model for a POSIX-based application framework, which, in part, addresses real-time composability in component design. This paper will present a justification for building software radios with design-for-latency, asynchronous method invocation (AMI) and deadline monotonic scheduling (DMS). We will examine state-of-the-art practice in designing large real-time systems. We claim by adopting and adapting these practices, SCA compliant software radios can be built with greater composability, which decreases the risk and cost to the integrator of multi-vendor/reusable software radios.

Extremely Brief Introduction to SCA

The JTRS (Joint Tactical Radio System) SCA (Software Communication Architecture) is a US government specified CORBA-based specification for building software radio systems that allow SCA-defined Digital Signal Processing (DSP), Security, Devices, and Services to be configured and used by communication applications called "waveforms". The purpose of SCA is to allow radio applications to be made "standard" and therefore portable and reusable. The SCA CORBA interface specifications are called the Core Framework (CF) which is the application view of the Operating Environment (OE).

Although SCA was not designed with composability of the OE in mind, the SCA nevertheless endeavors to define and negotiate for finite resources within the OE with "capacities". The nature of these external capacities assumes superposition, that is, that resources are consumed in a manner that is additive (or exclusive). Although some computation resources are additive, certain resources are not subject to superposition. For example, the throughput rate of A component-based operating environment and waveforms allows the system integrator to build a "component toolkit" to be used to provide functionality in both the OE and waveforms. As will be shown, the non-scalability of conventional implementations make the development of

a device with physical latency, such as disk head seek time, may increase service latency in a means not easily represented by "capacity".

Ideally, software components are designed such that superposition of unrelated software holds. This is a very difficult problem because there are many resources that are shared unbeknownst to the software component. From the devices shared through an operating system to the mutual exclusion mechanisms the standard libraries use to keep unrelated threads (and plug-ins) from corrupting shared data structures, there exists a great deal of shared state external to the component. In the absence of some priority-inversion avoidance mechanism, any shared state is an opportunity for unbounded priority inversion (which is obviously not subject to superposition.) Describing real-time task completion latencies as a "capacity" with superposition of the time-independent (sporadic) task completions yielding CPU utilization is useful for building composable real-time systems – however, as will be shown, a conventional implementation of SCA makes this impossible. flexible toolkits difficult and/or hard-wired to the locking/thread model of the system. A second benefit of a component based architecture would be to allow flexible integration of multiple waveforms within a single radio. This is particularly advantageous with the approaching ubiquity of

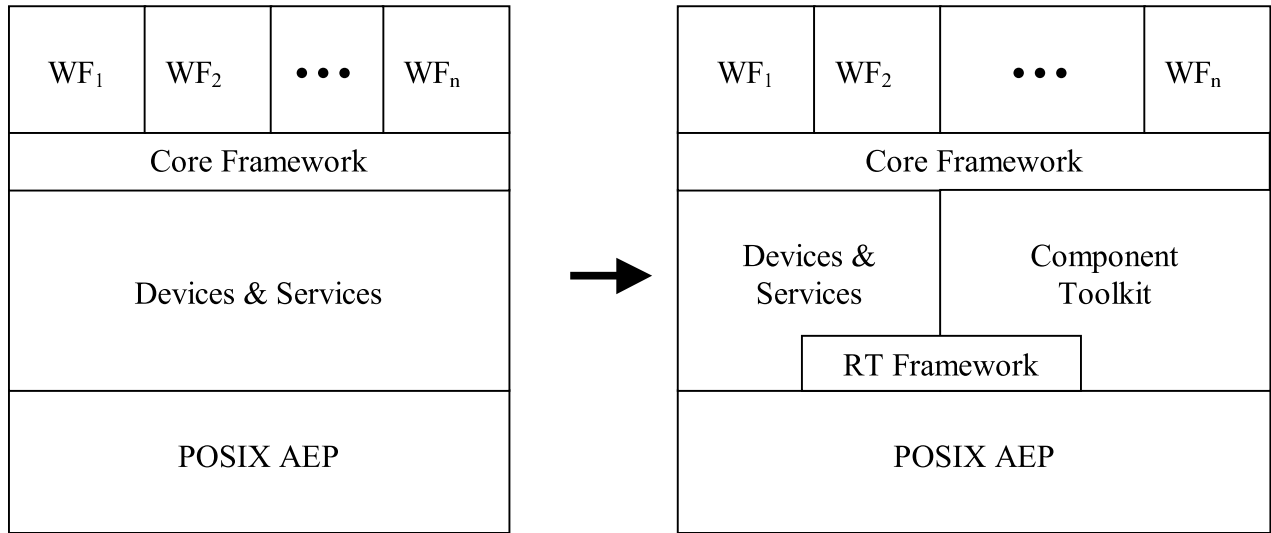


Figure 1. By introducing a RT framework and a Component Toolkit to an SCA implementation, the intent is to increase composability of the OE as well as provide OE/WF developers a set of real-time components whose properties are known and do not rely on “global system knowledge” for their correct use.

networking waveforms and the need for support of conventional line-of-sight waveforms. The authors believe that SCA will migrate over time to a component-based architecture. However, in the near-term there is a desire to reduce integration cost by means of a more composable operating environment with mitigation of current SCA-derived behaviors and “natural” implementation.

The SCA Specification / CORBA Synchronous Method Invocation Issue

The CORBA specification of SCA uses a very high percentage of two-way calls, that is calls that return values and/or out parameters. This call structure maps very well in procedural languages, such as C++, to making synchronous client calls to the CF servants. The SCA specification of using the POSIX AEP as the basis of waveforms leads to the “natural” implementation of a pthread-per-client and a multi-thread per servant model. In order to protect non-reentrant class internals from the methods called by “other” threads, some method of mutual exclusion is required. Particularly in the case of C++ exceptions, each class protects much of its code with a lock or mutex (rather than simply protecting the shared data) thus increasing the preemption latency for even high priority tasks (that share the lock) even with some means of priority inversion avoidance. This thread model, described pejoratively as “thread & lock”, is the subject of the Sutter Litmus Test: “Any programming model where a

thread is a central abstraction is too low-level (e.g., uncontrolled blocking / reentrancy / affinity).” [Sut 05]

Although “natural” as an SCA implementation, the “thread & lock” model is recognized in other parts of the real-time industry as non-scalable because it requires the caller to know and understand the “global” thread locking model, classes must have internal knowledge of the other classes methods. In C++, the locking scheme may vary between the base and derived classes, which the caller would prefer not to have to discern. In a typical radio application where control and data flows are bi-directional, lock hierarchies, lock recursion and lock order must be exactly followed or race-conditions and deadlocks will occur. Servant recursion requires integration time analysis and a worst-case allocation of servant threads within the target POA or servant recursion deadlock will occur.

Moreover, since thread priority is a global attribute (ie., only the highest priority runnable thread is running), priority is itself not composable. The real-time behavior of a servant depends not only on the priority of the callers and the servant, but on all the priorities of all the other tasks on the system.

Priority is also a weak-specifier of latency. In real-time systems, service latency is the desired design parameter. The system requirements state that a buffer needs to be received,

processed and enqueued with, say, 20mS not at priority “20”. And yet “thread & lock” model attempts to use thread priority to “bound latency. Priority bounds latency for only the highest priority runnable thread. The “thread & lock” model fails for large real-time systems of periodic or sporadic (stochastic) inputs/outputs. Many SCA software radio waveforms and operating environment implemented as “thread & lock” fail to scale and do not lead to a composable operating environment and waveforms.

Active Object Mitigation Strategy

A mitigation strategy to the “thread & lock” strategy is the active object [Lav 95]. The active object implements a class as a queue server. The public methods of the class queue work to be handled by the “active” thread of the object. By sequentializing access to the class internals by servicing requests on the work queue, the active object moves the

classes where a pthread-per-class would be considered overkill.

Rate Monotonic Alternatives to “thread & lock”

Other real-time fields have avoided the “thread & lock”. For many years the flight simulation community used “frame based scheduling” which allocates time to highly defined activities. Complete knowledge of the input/output/runtime of each of thousands of activities is required for the globally scheduled frames. The advantage is that by scheduling within one heavy-weight task 1000’s of small non-blocking light-weight tasklets, control is centralized and the tasks need no mutex to protect shared resources – all resource reads and writes are scheduled. This technique has been in use, in particular within the flight simulation community, for 40 years. It requires global knowledge, it is rigid and brittle to change, but it works.

Within the last 15 years, attempts have been made within the training systems community to use “frequency base scheduling” [Lucas 92]. That is, using operating system heavyweight threads and preemption as well as Rate Monotonic Theory [Liu 72] to allow high frequency update tasklets to preempt lower frequency response tasklets. Some method of state-vector sharing between data producer and consumer is employed to reduce the per-item locking.

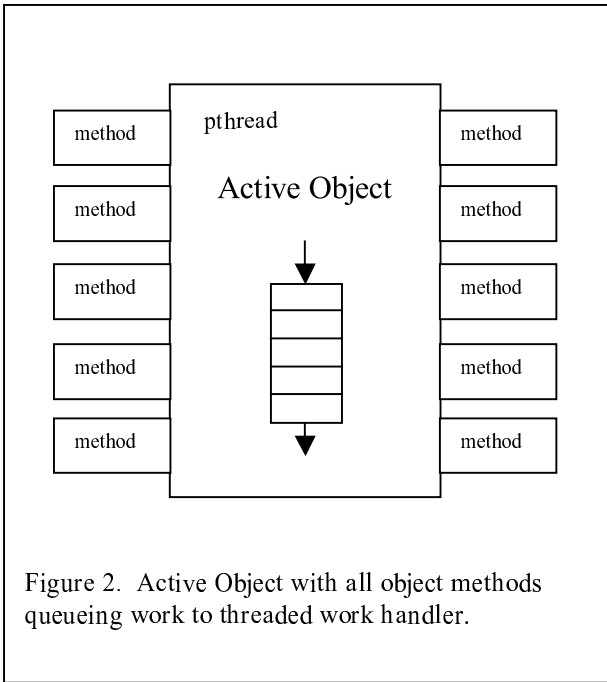


Figure 2. Active Object with all object methods queueing work to threaded work handler.

mutex from the individual methods to the work queue “put”/“get”. The active object abstraction is very useful to implement state-machine patterns.

The active object strategy is not without its downside. Although the priority of the active object thread can be determined external to the class, the synchronous CORBA calls and the design constraint of being non-reentrant effectively prevent the active object from making latency guarantees. It also fails to scale for “little” lightweight

"While RMA does require engineers to re-frame their understanding of scheduling issues to a more abstract level, only moderate training is required for people to be effective in using the technology" [Fow 93]

Deadline Monotonic Scheduling

In the early 1990s, rate monotonic theory was extended repeatedly [Sha 91] to include non-periodic real-time phenomenon such as sporadic tasks as well as generalized into latency-based Deadline Monotonic Scheduling (DMS)[Aud 92]. The authors have been part of the development of several large machine control applications using DMS as the basis of stochastic reactive tasking. This is particular advantageous when the bidirectional control/data flows are hard to predict a-priori such that “global knowledge” would be difficult to design into the system.

Like rate monotonic systems, deadline monotonic systems require that all lightweight tasks executing be non-blocking. If a large percent of CPU utilization is scheduled in accordance with rate monotonic theory and some means of priority inversion is present, the system will produce results for predicable schedulability as described by rate monotonic theory.

In several large non-periodic (sporadic) machine control applications the authors have worked on, deadline monotonic reactive tasking has been used with excellent outcomes. The software design is done on the basis of required latencies for correct operation. Software requirements for response latencies are allocated from the system requirements and derived for each function that composes the end-to-end requirement. The active object concept is expanded to a set of objects that implement latency “contracts” to the “application” software. The active object latency-servers are scheduled deadline monotonically, thus all the software that runs on the latency-server threads run in accordance to DMS theory. This framework allows application designers to focus on “design-for-latency”. That is, designed latency of each component/method using completion deadlines derived from the specification deadlines of components higher up and ultimately derived from the product specification. Whether automated design tools or integration failure analysis, latency-based designs allow tracability from requirements to components/objects that priority-based systems don’t.

The DMS-based framework also concentrates utilization/overrun metrics and debug within the latency-servers rather than distributed with application as well as operating environment codes. This allows the framework to “catch” violations of deadline monotonic utilization constraints and non-blocking constraints early in the development cycle. Like other active object codes, the serialization within a “latency contract” assures the application of exclusivity of all object components under the same contract (ie., scheduled on the same active object

serialized lockless design and can be used (with the addition of extra context switches) to eliminate mutexed access to shared data structures altogether.

Rate Monotonic Theory Results of Interest to Software Radios

The most conspicuous advantage of rate monotonic theory in deadline monotonic scheduling is that for systems conforming to RMA requirements, it is known to be optimal for fixed priority scheduling [Leu 82]. That is no amount of “global knowledge” or integration time tweaking will produce a better fixed priority scheduling of the system. This allows designers to build radios rather than tweak thread priority tables.

Another integration time benefit of RMA is that utilization can be used to determine schedulability.

The utilization, U, for n rate monotonic tasklets is given by,

$$U(n) = n(2^{-1/n} - 1)$$

Which converges for large n to ln(2) or 69%[Sha 91].

That is, a given load of sufficiently sporadic tasklets is schedulable in the “worst-case” when the processor utilization is less than 69%. The ability to determine “worst case” schedulability from empirical data allows the integration of multiple independent software components with a theoretical basis that the integrated product will be schedulable. For example, if two independent software components each run with a utilization bounded at 30%, the

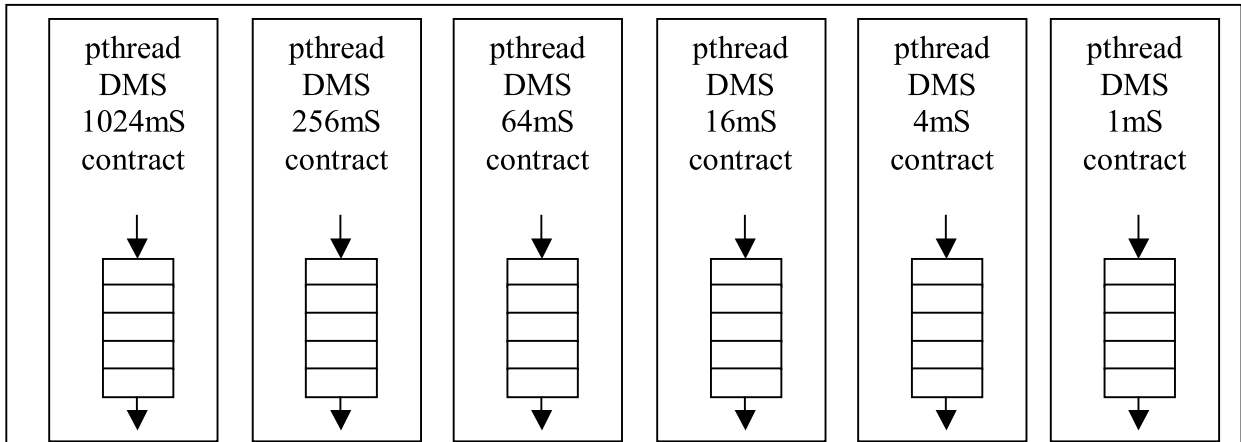


Figure 3. Active Object latency-contract servers effecting a Deadline Monotonic Scheduler. Tasklets are all scheduled sequentially according to their required response.

thread). This allows application designs to do de-facto

integration superposition would yield a total utilization of

60% which is guaranteed to be schedulable under RMT. Many of the components in a large real-time system may not, in fact, be independent. Yet, in the authors' experience to a large degree, utilization superposition holds. The ability to predict integrated software schedulability from component utilization is an extremely useful risk-reduction property of a DMS-based real-time system.

Implications of RMA to SCA

The requirement that a rate/deadline monotonic task be non-blocking requires that system concentrate blocking (physical) I/O into separate low utilization threads. This includes all devices with latency for which the calling task would wait pending an interrupt. These threads should be prioritized with the priority used for the deadline monotonic latency-servers that serve tasklets of utilizations similar to the utilizations of the I/O operations. The other blocking I/O of particular interest to software-defined radios is the CORBA Core Framework. On other real-time systems, the synchronous nature of the standard Synchronous Method Invocation (SMI) of CORBA and other RPC-type mechanism are made asynchronous by hand-coding a proxy where each caller queues a request to a interface-aware proxy-thread which makes the actual blocking call and upon return from the call generates a callback event. The proxy method is effective when only one caller per interface is present. This obviously does not scale well at all and many applications would not make real-time commitments with a 1-deep call interface. (On the other hand, if the servant side is asynchronous, the proxy thread will block for a very short time.)

A more satisfactory solution would be that use the IDL to generate the client proxy. Many of the ORB vendors consider their client-stubs proprietary and having IDL generate a compatible client-stub would violate licensing and non-disclosure agreements.

A better and more scalable solution for caller asynchronous behavior with synchronous interfaces is the CORBA

Asynchronous Method Invocation. Although supported by a small fraction of the ORB vendors, it provides the caller with IDL generated asynchronous callbacks.

With the foundation of deadline monotonic scheduling and some means to make asynchronous CORBA calls in place, the formation of a scalable and composable real-time SCA design come into focus. As was true of the active object model, by means of running all accesses to protected data on the same reactor (latency contract active object), lockless mutual exclusion may be obtained. If it is required that explicit mutex locking be in effect, for example if successful negotiation of single reactive response time is impossible for all cases, then a priority inheritance mutex may be used within the constraints of the deadline monotonic scheduler [Sha 88]. However, adding mutexes to the DMS framework will have the same recursion and lock-order considerations as the "thread&mutex" case. The authors have worked on DMS systems designed with and without mutexes, and a minimalist approach to introducing mutexes into the framework is less error prone and easier conceptually than the orthodoxy of pure lockless design.

Conclusion

The conventional "thread & lock" model has problems with scalability and composability for real-time systems in general and SCA system in particular. Additionally, the CORBA SMI invites a "thread & lock" implementation and effectively inhibits rate monotonic theory-based designs such as deadline monotonic scheduling that is used to great effect in other complex real-time systems, both periodic and sporadic. Mitigation strategies, such as active objects, present realizable near-term advantages toward greater composability. However, to gain the advantages of RMA schedulability as well as the latency contract server thread model, the SCA Core Framework as well as the waveforms will need to be run using some form of asynchronous method invocation (AMI) in order to conform to the RMA requirements.

[Sut 05] Sutter, H. "C++: Future Directions in Language Innovation", TLN309, p11, Microsoft Professional Developers Conference, Sept. 13-16, 2005. Los Angeles, CA.

[Lav 95] Lavender, R. G., & Schmidt, D. C. "Active object: an object behavioral pattern for concurrent programming." Proc.Pattern Languages of Programs, 1995.

[Liu 73] Liu, C. L. & Layland, J. W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment." Journal of the Association for Computing Machinery 20, 1 (January 1973): 40-61.

[Leu 82] J. Leung & J.W. Whitehead, On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks, Performance Evaluation, 2(4):237--250, 1982.

- [Sha 91] Sha, Klein & Goodenough, J. "Rate Monotonic Analysis for Real-Time Systems," 129-155. Foundations of Real-Time Computing: Scheduling and Resource Management. Boston, MA: Kluwer Academic Publishers, 1991.
- [Aud 92] N. C. Audsley, A. Burns, M. F. Richardson & A. J. Wellings "Real-Time Programming", 127-132, ed. W. A. Halang and K. Ramamritham, Pergamon Press, 1992.
- [Luc 92] Lucas, L. & Page, B. "Tutorial on Rate Monotonic Analysis." Ninth Annual Washington Ada Symposium. McLean, VA, July 13-16, 1992. New York, NY: Association for Computing Machinery, 1992.
- [Fow 93] Fowler, P. & Levine, L. Technology Transition Push: A Case Study of Rate Monotonic Analysis Part (CMU/SEI-93-TR-29). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.
- [Sha 88] Sha, L., & J.B. Goodenough. "Real-time scheduling theory and Ada," Computer, April 1990, pp. 53-65.

Authors

David Carlson is the president of Chronolytics, Inc., a company which builds a scalable real-time framework called Chronolytix on its real-time enhanced Linux(tm). Mr. Carlson has been software practitioner for 20 years specializing design and integration of large real-time systems such as flight simulators and real-time embedded systems such as integrated circuit manufacturing systems and mid-volume / high-volume printing equipment. Mr. Carlson's research areas include exploring using "best-of-breed" industry techniques for building scalable real-time designs on symmetric multi-processor (SMP) systems running Linux(tm). Mr. Carlson has a BSEE and MSEE both from the University of Rochester.

Raymond Lindenmayer is a senior engineering manager at Harris corporation working on the Falcon III family of radios. Ray was a software development and product manager at Xerox for 15 years prior to starting with Harris Corporation. He was responsible for managing the embedded software on the current Xerox IGEN and Nuvera product families. Ray is also a Lt Col in the Air Force reserves. His jobs included Chief of Automated Logistic Technology for USCENTCOM at MacDill Air Force base where he was mobilized for 1 year in support of Operation Enduring Freedom. He was critical in the establishment of the forward headquarters for USCENTCOM in Qatar in November 2002 and he also led the deployment of advanced Radio Frequency Identification technology to establish in-transit visibility of critical cargo to support the War on Terrorism. Ray has a bachelor degree in Computer Science from Syracuse University, a Masters degree in Computer Science from American University, and a Masters in Business Administration from the Simon School at the University of Rochester.