Challenges for Building Complex Real-time Computing Systems

Michael W. Masters

U.S. Naval Surface Warfare Center, Dahlgren Division

MastersMW@nswc.navy.mil

Lonnie R. Welch

School of EECS

Ohio University

welch@ohio.edu

Abstract

In the early years of computing, real-time systems consisted of single computers interfaced to a few sensors and effectors, and perhaps to a user control device as well. While such systems still exist, and have become almost ubiquitous in a myriad of everyday devices, the world of real-time computing has expanded in scope and complexity in ways comparable to the proliferation of computers for many other aspects of business, economic and social life. Perhaps the most recent evidence of this expansion is the employment of distributed processing to achieve the objectives of real-time system designers. With the expanded use of computers, particularly in distributed real-time embedded systems (DREs), issues of complexity have emerged that were not faced by early real-time system designers. Ccomplex aggregations of computers operating as a distributed system have assumed a distinctly different character from early real-time systems, exhibiting far more variation in form as well as in the underlying technology base. Dealing with this complexity has become a major driver for innovation in both system architecture and computing technology. The challenge designers face today is, how to design, build, deploy and maintain ever larger and more capable distributed real-time embedded systems to meet the demands of a changing world. Modern real-time systems are likely to be more than just systems; they are rapidly becoming large-scale systems of systems. Because of their long life and high cost, complex real-time systems cannot be recreated as new starts when new requirements emerge. They must evolve and undergo changes and upgrades in place. Furthermore, since the technology base on which DREs are built is changing rapidly, complex real-time systems must allow for the convenient insertion of new computing equipment and supporting software products. For these reasons, flexibility of design has become a paramount concern.

This paper will present major challenges and promising solutions for the problem of building real-time open architectures. Based on our experience within the Navy shipboard computing systems domain, we will consider technical characteristics, change management, system flexibility and dynamic resource management.

Introduction

In the early years of computing, real-time systems consisted of single computers interfaced to a few sensors and effectors, and perhaps to a user control device as well. The computer program in this processor almost certainly performed a cyclic closed loop control function, monitoring sensor inputs and commanding effectors in accord with a fixed and rigid control algorithm to achieve designer objectives such as managing process flow, vehicle motion, automatic response, etc.

While such systems still exist – and indeed have become almost ubiquitous in a myriad of everyday devices, including automobiles, appliances, spacecraft and video games – the world of real-time computing has expanded in scope and complexity in ways comparable to the proliferation of computers for many other aspects of business, economic and social life. Perhaps the most recent evidence of this expansion is the employment of distributed processing to achieve the objectives of real-time system designers.

With the expanded use of computers, particularly in distributed real-time embedded systems (DREs), issues of complexity have emerged that were not faced by early real-time system designers. It is true that small cyclic closed loop control systems are still built according to principles developed early in the history of real-time systems. Furthermore, in recent decades rate monotonic analysis (RMA) has provided a theoretical underpinning for such systems, but the actual architectural structure has changed little.

On the other hand, complex aggregations of computers operating as a distributed system have assumed a distinctly different character from early real-time systems, exhibiting far more variation in form as well as in the underlying technology base. Dealing with this complexity has become a major driver for innovation in both system architecture and computing technology. The challenge designers face today is, how to design, build, deploy and maintain ever larger and more capable distributed real-time embedded systems to meet the demands of a changing world.

Furthermore, the traditional problems of functionality and performance are seriously compounded because such systems can no longer be optimized exclusively for performance but instead must find a suitable niche in a complexity, multi-dimensional trade space involving many design factors, including real-time performance, survivability, secure operation, large size, complex, long service life, continuous evolution, distribution, mission criticality, and cost consciousness. In summary, modern real-time systems are likely to be more than just systems; they are rapidly becoming large-scale systems of systems.

Because of their long life and high cost, complex real-time systems cannot be recreated as new starts when new requirements emerge. They must evolve and undergo changes and upgrades in place. Furthermore, since the technology base on which DREs are built is changing rapidly, complex real-time systems must allow for the convenient insertion of new computing equipment and supporting software products. For these reasons, flexibility of design has become a paramount concern.

Technical Characteristics

Technically, real-time systems encompass a wide range of characteristics that fundamentally impact their design and operation. No longer are they relatively straightforward cyclic hard real-time programs. They must deal with and accommodate a wide variety of processing objectives. These include:

- Driven by external world events
- External environments that cannot be characterized accurately a priori
- High volume throughput of continuously refreshed data
- Hard real-time deadline aperiodics
- Asynchronous, event-based low latency responses
- Soft real-time processing requirements
- Reaction time paths across multiple hardware and software components
- Wide dynamic range of processing loads
- Operator display and control requirements
- High availability and survivability requirements
- Stringent certification and safety requirements

Open Systems

Given the rapidity with which commercial computing technology is evolving, a major consideration for any long-lived system is the problem of efficiently changing the underlying COTS (commercial-off-the-shelf) computing technology at frequent intervals to keep it current and easily maintainable. This process is often referred to as *technology refresh*. A fundamental problem arises in performing this process when application code is tightly coupled to the underlying technology base and that base changes frequently in ways that impact application code.

In such circumstances, the implementation of a technology refresh also necessitates revision and retesting of the application computer programs. In worst case circumstances, the time and cost to accomplish this exercise may constitute a significant percentage of the original development cost of the system – while providing no increase in functionality or performance. Clearly, this is an unacceptable situation, especially given the sums of money involved. Methods for reducing this cost are very much in demand.

The method that has evolved within the DRE community involves a combination of using standards-based hardware and support software combined with extensive layering of modest sized componentized software products, each one of which isolates the layers above from changes below via standard or pseudo-standard application programmer interfaces (APIs). Systems built according to this approach are called *open systems*. Such systems are easy to upgrade and to refresh because hardware and support software components on which applications reside are, by design, interoperable and interchangeable. In its POSIX documentation, IEEE defines an open system as:

> *A system that implements sufficient open specifications for interfaces, services, and supporting formats to enable properly engineered applications software: (a) to be ported with minimal changes across a wide range of systems, (b) to interoperate with other applications on local and remote systems, and (c) to interact with users in a style that facilitates user portability.*

Open systems – and architectures built according to open system principles – possess a number of common characteristics. While not every open system possesses every possible characteristic, most open systems tend to possess most of these characteristics. The attributes of an open system include the following:

- Use of public, consensus-based standards
- Adoption of standard interfaces
- Adoption of standard services (defined functions)
- Use of product types supported by multiple vendors
- Selection of stable vendors with broad customer bases and large market shares
- Interoperability, with minimal integration
- Ease of scalability and upgradability
- Portability of application(s)
- Portability of users

The result of incorporating open system principles into a system architecture is that such systems fulfill the following relationship:

Open Systems = Standard Interfaces + Defined Capability → Interoperability + Ease of Change

While devising rigorous metrics that reflect abstract concepts such as open systems and open architecture is difficult, it is not impossible – especially as long as one realizes that such metrics may themselves be somewhat qualitative. For example, standards compliance may be approximated by first-order estimation of the ratio of standards-based APIs to total APIs. Time and work effort needed to incorporate new functions may be tracked and compared to historical data about similar activities. Numbers of interfaces, especially multi-client server interfaces, may be compared to previous implementations. Early process-oriented system engineering efforts should address this question explicitly.

Managing Change

In virtually all long-lived systems, change is inevitable, and usually occurs at many levels. Application functions change, paced by threats and missions. Computing technology changes, driven by technological innovation and market pressures. Even standards change, albeit at a slower pace. Managing this change and isolating its effects is the key to successful implementation of DREs. The primary techniques for attaining isolation are use of standards, and use of a category of support software known as middleware.

Middleware comes in two major types, adaptation middleware and distribution middleware. A third type of support software, resource management, provides system management functions that are not strictly middleware in nature but that, like adaptation and distribution middleware, provide vital services in the composition of large systems.

Together, these three types of support software provide a synthesizing function that allows applications in a distributed system to interface with each other and with the underlying computing equipment and operating systems.

System Flexibility

In the past, systems, both distributed and otherwise, have been designed with a fixed and rigid allocation of processing tasks to processing resources – i.e., in a manner somewhat analogous to the fixed and rigid time division multiplexing approach of real-time cyclic computer programs and executives. This approach has many advantages, not the least of which is its analyzability and predictability. It is, in a relative sense, easy to design, build and test.

However, this approach also has weaknesses, just as the cyclic approach to real-time computation has weaknesses. In the case of a fixed allocation of processing to processing resources, the weaknesses include a similar brittleness of design, a fault tolerance approach that generally requires at least double the amount of equipment as that required to simply provide inherent mission functionality, and an inflexibility in operation that substantially inhibits run time adaptation to the dynamic conditions of the moment.

Such systems are also frequently constrained in their performance by design choices made at system inception and early design. In particular, many systems exhibit a performance "sweet spot" where they perform well. When pressed to operate beyond this range in terms of capacity requirements, they tend to degrade in performance, sometimes catastrophically. In effect, such designs are not scalable with respect to system load. The application code needed to manage load shedding in such circumstances adds significantly to development cost, test time and software defects.

One of the best examples of the impact of lack of scalability is that of the popular Internet service provider, America Online. Years ago, AOL changed its pricing structure from a use-time basis to a fixed fee per month. Not unexpectedly (except to AOL management), users thus freed from the cost burden of time-based charges began staying online much longer. AOL had not anticipated this sudden increase in required capacity. Furthermore its architecture was not particularly scalable at that time. The result was a highly visible nation-wide gridlock for AOL users.

Dynamic Resource Management

It is interesting and significant that, at least in the web service market place, new technologies have since been developed to deal explicitly with the need to dynamically adapt to wide variations in service and capacity requirements placed on computing resources. These technological innovations encompass both hardware solutions and software solutions. In either case, their purpose is to provide system designers with a *dynamic resource management* capability.

Dynamic Resource Management, or DRM, is a computer system management capability that provides monitoring and control of network, computer, sensor, actuator, operating system, middleware, and application resources within a distributed computing environment. DRM is a critical component for exploiting the new computing capabilities emerging from the rapidly evolving COTS technology sector. When combined with a standards-based and modular approach to system design, DRM provides an ability to readily change system configurations to support not only rapid insertion of new functional algorithms but also rapid and inexpensive COTS technology refresh.

Without DRM and without the portability that an open systems approach fosters, each insertion of new functional capability and each change out of COTS computing equipment must be accomplished via a lengthy redesign and development process, with their inherent cost and schedule impact.

The primary unique capability provided by Resource Management is its ability to maintain a user specified level of performance for each application computer program as it executes on the available computer system resources.

DRM accomplishes this function by allocating – and reallocating as necessary – the application computer programs to the computer resources based on the ongoing observed performance of the application system as it operates and performs its designed functions. Once the application system has been initially allocated and initiated, DRM maintains the user specified level of performance in the presence of not only changing a computer system configuration (e.g., due to computer and/or network failures) but also varying external stimuli experienced by the application system and changing mission requirements as designated by the application system's users.

Because of this flexible dynamic allocation capability, DRM confers a number of benefits on real-time systems. DRM is particularly valuable for mission critical systems that must maintain a specified level of performance, the absence of which may lead to drastic, even catastrophic consequences.

DRM provides three major system benefits. First, it allows the DRE system to sustain a level of performance that is invariant with respect to external load. By allocating computing resources to critical computations that would otherwise be overloaded, a steady level of performance is maintained. Second, since the system resource allocation can be rebalanced during system operation, the system can be tuned to the mission at hand. Thus, mission flexibility is conferred, a capability which is virtually unknown today.

Finally, because the DRM capability treats the entire computing system as a pool of computers, any one of which can run any program, any computer can serve as a backup or spare for fault tolerance. Thus, rather than a primary-backup fault tolerance model, the proper view in a DRM-managed system is an N + M redundancy scheme. This approach has the potential to drastically increase operational availability following casualty events.

For systems where maintenance of the deployed system is a concern, DRM offers an additional benefit. DRM acts as a manning reduction and operational cost saving enabler. Use of the pool of computers approach means that any computer can do any function (with certain specialized exceptions). Thus if a sufficiently large pool of computers is deployed, computers that fail during operation do not have to be repaired or replaced immediately, but can be serviced at a convenient time. This in turn means that personnel do not have to be dedicated to computer system maintenance, thus reducing maintenance staffing and operational cost.

Conclusions

In broad and general terms, architecture is defined as "the structural design of an entity." Adding "openness" to the list of architectural characteristics implies that the "structure" of the architecture explicitly promotes interoperability, both internally and externally, as well as ease of modification and extension.

It is an engineering truism that what is achievable in system design (architecture) is a function of not only the task to be accomplished but also the technologies that are available. In the early days of real-time computing, the technology base included custom or niche market computers, point-to-point connectivity, real-time executives and a variety of proprietary support packages tailored to the specific characteristics of individual applications.

However, the evolution of high performance COTS provides an opportunity to design architectures capable of meeting the demands of ever more complex and demanding applications. The need for evolution of architecture toward an open and distributed approach is motivated by both performance and supportability considerations. Commensurate with this dual set of motivating factors, the goals of open architectures are as follows:

- DREs that continue to meet their mission needs through upgrades and changes
- System design that fosters affordable development and life cycle maintenance
- System design that reduces upgrade cycle time and time-to-deployment for new features
- Architecture that is technology refreshable despite rapid COTS obsolescence

Finally, note that system requirements include not only capability and performance goals but also engineering "-ility" goals as well. In addition to traditional "-ilities" such as reliability and survivability, the "-ilities" of open architectures include metrics such as *portability, affordability, extensibility* and *flexibility of use*. These objectives can be met through a combination of careful design, combining open systems principles and standards, and dynamic resource management.