Testability of Complex, Middleware-Based Systems

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

Robert E. Bernstein
Lockheed Martin Corporation
robert.e.bernstein@lmco.com

Amarendranath Vadlamudi
System/Technology Development Corporation
amar.vadlamudi@stdc.com

Abstract

The use of object-oriented middleware technologies in mission critical systems has enabled the creation of complex, adaptive distributed applications that incorporate fault-tolerance strategies. A redesign of the Aegis Weapon System will operate on hardware distributed throughout a ship¾ both for scalability and fault tolerance. Critical applications will need to maintain backups in separate damage compartments while communicating over a shared infrastructure. However, any resulting unpredictability might cause the system to miss hard real-time deadlines. Certification of current weapons systems is based on design analysis, simulation, and extensive testing. Extrapolation of this method would require testing of huge numbers of individual configurations, an intractable problem. One promising alternative involves the use of an intelligent resource manager that can compare the performance of an application to a system model, and effect configuration changes to improve system performance. This paper discusses the results of that investigation and explores the use of this capability in the context of systems, such as the Aegis Weapon System.

1. Introduction

The use of middleware technologies, based on distributed objects, has enabled the creation of complex, distributed systems. Multiple component objects, acquired from many sources, can be combined to form useful applications. In turn, multiple separate applications operating on a shared infrastructure can then be combined to form complex systems. In addition, the use of object-oriented design techniques provides an opaqueness over the internal operations of individual objects such that individual objects can often be replaced by enhanced object components with additional properties, such as fault tolerance, without requiring system redesign. The result is that complex, adaptive, distributed systems using multiple, independent fault tolerance strategies are now being developed for mission critical applications.

The Aegis Open Architecture provides an interesting, but not unique, example of the issues facing these systems. The Aegis Weapon System is a shipboard (cruisers and destroyers) system that both protects the ship and controls offensive weapons. Lockheed Martin is reimplementing this system using distributed object middleware. The goals of this effort parallel those of analogous commercial projects: reduced software life-cycle costs, enhanced system extensibility, simplified design, improved performance, and potential reuse in other systems. Yet this redesign represents a major change in development strategy: a switch to the use of an open architecture, based on COTS (Commercial Off The Shelf) hardware and software, using commercial best practice techniques. This conversion is exposing weaknesses in current system design and development methodologies that limit the potential payoffs of the above goals. This paper explores several issues related to system certification in this context.

2. Existing System

The current Aegis Weapon System is an excellent example of a purpose-built system. There is a single, coherent architecture that was created specifically for Aegis. Operating on computer hardware unique to the military (the AN-UYK 43), the cyclic executive scheduling in the operating system, also unique to Aegis, supports hard real-time applications by providing deterministic behavior via allocation of specific CPU scheduling slots to specific element applications. The system provides fault tolerance via dual processors operating in a warm backup configuration. Like many other systems, it has been extended and enhanced over

Presented at: International Conference on Dependable Systems & Networks, Washington, D.C., June 23-26, 2002.

many years such that the architecture has been stressed in ways not contemplated by the original designers. Still, the Aegis system is recognized as a system that effectively performs its system mission.

By definition, weapons systems are dangerous. There are numerous ways in which a system failure could result in loss of life on the Aegis ship. One obvious case is failure to close the missile control loop, resulting in the self-destruction of an outgoing missile whose target is endangering the ship. Thus, these systems must be certified both for safety and mission effectiveness. Certification is based largely on design analysis, mission simulation, and functional testing driven by established disciplines in the industry, such as reliability modeling, safety cases, and fault injection testing. Each application is analyzed to verify that it conforms to the resource allocation envelope assigned to it within the overall system architecture. Resource usage patterns of the applications are subsequently confirmed in simulation runs. Finally, the complete system undergoes an extensive testing regime whereby the system performs numerous mission scenarios while being subjected to various input loads and simulated faults. One estimate is that testing currently comprises 40% of overall system development costs, and each major testing cycle take 6 months to a year.

3. Aegis Open Architecture

The goals for the redesign include aggressive performance goals, emphasizing significant improvements in both load scalability and reduced latency in recovering from component failures. Although the design process is still underway, the expected hardware architecture includes multiple processor pools supporting application use of parallel algorithms. Digital signal processing (DSP) for the phased radar system will execute on COTS single board computers. Other processing will be based on COTS server systems, such as are sold by Sun Microsystems and SGI. The communication infrastructure will be based on COTS routers and switches.

The software architecture is equally aggressive. The operating systems will be COTS real-time OSs, such as LynxOS and Solaris. COTS middleware includes CORBA, publish/subscribe messaging, and database management systems. Software applications are being designed using the UML-based Rational Unified Process (RUP) methodology from Rational Software.

4. The Testing Challenge

Although the use of COTS hardware and software technologies provides significant improvements in system performance and maintainability, it also significantly alters several assumptions underlying the existing testing philosophy by introducing additional sources of variability. These changes become more apparent by considering the ability of the Aegis system to meet its hard real-time deadlines, such as those in the weapons launch execution path.

4.1 Variability Due to Shared Infrastructure

One major change and a significant advantage introduced by distributed object systems is the use of a shared infrastructure. Various applications use resources differently at various times during their execution. The radar tracking system will use more resources when there are more targets visible in the sky. Weapon launch systems will use more resources when actually launching weapons. UAV monitoring applications will use more resources when a UAV is actually in flight. Other applications will vary based on time of day or will be triggered by events initiated by off-ship activities.

These changes in resource usage will inevitably result in changes in the interaction patterns between applications. Environmental noise will cause hardware errors, such as disk read errors, or memory ECC errors, or communication checksum errors. Generally, these operations will be automatically retried and will often succeed on the second or subsequent operation. Eventually someplace in the system, as a result of the cumulative effects of such retries, message A will be presented to an event-driven application after message B, when it had previously always arrived prior to message B. This will result in a change in the operation of that application. If this situation never occurred during system tests, we can't know if it will impact the successful achievement of a system deadline.

4.2 Variability Due to COTS components

While the individual components of the legacy Aegis system have been designed to adhere to the resource usage envelope provided by the overall Aegis architecture, COTS components have not. They have been designed and developed independently, and will not interact with the Aegis applications until well after the component design and qualification process is complete. The use of multiple COTS hardware components will result in slightly different execution rates on different processors, or on different networks. Different instances of operating systems will have slightly different memory allocations patterns, resulting in slightly different

execution times for system call execution. DSP algorithms will execute using slightly different times based on the input date. While purpose-built hardware and software can be specially designed to use constant execution time algorithms, COTS components typically will not be, opting instead for the fastest execution time in each individual instance.

4.3 Variability Due to Configuration Evolution

Variability can also be introduced due to differences in system configuration. Although a system might start out in the same configuration each time, it will eventually evolve in an unpredictable way in response to the stimuli it encounters. This results in different resource loading and different interference patterns between even unrelated applications. It was earlier noted that many hardware components automatically retry low-level operations that encounter errors, such as memory ECC errors or noise encountered during communication. The fault tolerance capability of the system significantly amplifies this effect. If a computer node fails, the applications executing on that node will automatically switch to another node, resulting in different loading of that node. Even if the backup node had been operating in a "hot" backup mirroring mode, the system operating will be different because the failed node is no longer transmitting communication packets over the shared network. In addition, repair operations can introduce additional loading. An example would be the initial loading of a newly repair disk drive.

This variability is increased in the case of the Aegis Open Architecture due to the particular requirements of the system. Shipboard weapons systems are expected to continue operation even in the presence of battle damage. An incoming missile would be expected to flood at least one watertight hull compartment, resulting in the failure of all components in that compartment. To protect against this situation, the system must be designed to maintain application fault tolerance configurations with primaries and backups in separate hull compartments. Since there is a physical design requirement to limit the number of physical components that penetrate the bulkheads between compartments, the communication between the application primary and backup will occur via a shared communication link, resulting in increased interference between unrelated applications.

4.4 Variability Due to Application Evolution

Another major change in the Aegis Open Architecture is the goal of allowing more dynamic application evolution. The introduction of a new application or a major change to an existing application would obviously alter the resource usage patterns within the system. This situation is even more complicated, however: while it might be possible to use constant execution time algorithms and/or hot backup techniques to reduce the impact of the changes described above, it will is impossible to have predicted the resource usage patterns of a completely new application.

If each application upgrade required a complete system recertification, the introduction of a new application would be delayed by at least six months just due to the time required for executing the test cycle. The cost of this additional testing would rapidly overcome the cost benefits anticipated in the switch to the use of an open architecture.

The switch to COTS hardware and software components also affects system performance. The lifetime of a particular computer model produced by a computer vendor is typically a year or less. Even that period is deceptively long in that even a particular computer model will usually ship with different internal components at the end of its lifetime than at the beginning. Consider that the lifetime of a particular disk model is only 3 to 6 months. The result is that the performance characteristics of a computer that is deployed in an actual system is likely to be different from the one that underwent certification testing. (Note that the life cycle of a system hardware component is comparable to the length of the system test cycle.)

COTS vendors generally assume that their customers don't care about the specific performance characteristics of the components as long as those components are faster than their predecessors. Unfortunately, this is not always true for systems operating with specific physical time constraints. One telling example, which has been independently rediscovered in numerous environments, involves an upgrade to network interface cards (NICs). Consider the situation where one NIC is sending a large data frame to another system via TCP, resulting in multiple back-to-back messages addressed to the NIC on the other system. When the two NICs are identical, as is likely to be the situation in a newly deployed system, everything is likely to work as expected. If, however, the transmitting NIC is replaced by a faster version, it can be the case that the two packets arrive at the destination NIC with a lower inter-packet latency, perhaps before the slower NIC can handle the first arrival. The result is that the second packet will not be received. Because communication protocols are fault tolerant, the second packet will be retransmitted, either by the NIC or by the TCP protocol software. This transmission will typically occur automatically, without intervention by or even notification to the application. Everything works

fine—except that the application now encounters an additional, unexpected delay of one tenth of a second to a few precious seconds. The cumulative effects of one or more such delays is likely to result in the eventual failure of the application to meet its deadline.

5. Testing Methods for Distributed Object Systems

Even ignoring the complications that derive from the evolution of the hardware and software components over time, we encounter difficulties with the sheer magnitude of the testing task if we simply extrapolate from the current strategy. A system based on distributed object technology can be expected to include many more components than the legacy Aegis system. The inclusion of these additional components increases the magnitude of the number of different configuration combinations that must be separately tested—by one to two orders of magnitude. It is obviously impractical to test that many distinct configurations.

Consequently, we must alter our approach to system certification. The primary goal behind the conversion to an open architecture is to improve system effectiveness, so we'll ignore any changes that result in a lower level of certification.

One strategy would be to limit system reconfigurations to only those particular combinations that had been evaluated as part of the certification testing. This strategy is relatively simple to achieve. One way that it could be implemented would be by abstracting the entire configuration of a node to be the logical configurable unit. Each node would then be paired with an equivalent node in a different hull compartment. Then, if any hardware component or any application component on a node failed, the entire node would have to be taken out of service. This strategy would not allow the dynamic reestablishment of a fault tolerant capability by selection of a new backup node at run-time. A system could be designed for 2 or 3 or even *n* replicas, but that number would have to be fixed and would have to have been qualified as part of system certification testing. The use of this strategy would rapidly negate the earlier cost benefit goals and also severely limit the applicability of parallel execution of applications for scalability. In the long term, this strategy severely limits the utility of the open architecture approach, but it appears to be realistic and readily achievable. So it might be a reasonable and safe choice for initial deployment of the new system architecture.

A more capable strategy would to be identify "equivalent" configurations. The system would then allow system configurations that could be expected to perform identically to a configuration that had undergone testing as part of the certification process. In this strategy certification would need to be enhanced to also evaluate the assertion of equivalence. Configurations that one might expect to be equivalent would be fixed configurations operating on any of the identical computer nodes in a processor pool. Queuing theory also indicates that shared resources operated at low capacities (say less than 40%) might operate in an equivalent manner. The applicability of this strategy to an actual system requires significant testing, involving new testing strategies that articulate clear measures of equivalence. Thus, this strategy might be a useful target for a first or second system release with a plan for falling back to a simpler strategy if necessary.

A further step in the evolution of this strategy would utilize application performance modeling data to dynamically evaluate system execution for proper operation. In some ways, this strategy mimics the one used in the original, purpose-built design: the system architect assigns required performance characteristics to various components, which then adhere to that envelope. In the original design, each application would conform because of the overall determinacy of the system environment. This open architecture version would require active resource management that continually monitors system behavior and dynamically controls adaptive applications that would alter their operating modes to conform to the expected profile. This type of strategy offers the most potential over the expected life-cycle of the Aegis Open Architecture, but it appears to be somewhat beyond the current state of the art in resource management.

5. An Experiment Involving Intelligent Resource Management

As part of the QUITE[1] integration effort for the DARPA Quorum[2] program, we performed several simple experiments to investigate the potential use of an intelligent resource manager with adaptive applications in a context similar to that contemplated in this paper. One experiment scenario included an execution path of multiple processes that had been crafted to use CPU and communication resources in a way such that some configurations would not operate successfully. For example, in one possible configuration the processes would overload the system by requiring 125% of the available CPU cycles. Another configuration would require 150% of the available communication bandwidth on some, but not all, nodes. There were only a few configurations that would satisfy all the requirements, and only one configuration that was optimal.

We setup a test-bed for the experiment. We used the Remos[3] system to monitor the network behavior and then modified the DeSiDeRaTa[4] resource manager to analyze the CPU and communication performance metrics and then use that data to reconfigure the system as appropriate. After initializing the experiment in a stable configuration, we introduced an "unexpected" extra communication load, which caused the application to miss its established time constraints. We then expected the resource manager to identify a new configuration that would allow the application to meet its deadlines even in the presence of the reduced bandwidth availability.

Our results were sporadic. When the resource manager decided to reconfigure the system, it would select the "proper" new configuration. Unfortunately, the resource manager did not always decide to reconfigure the system even when deadlines were being missed. We were unable to completely analyze the problem in the limited time that was available. Our preliminary determination was that the problem was due to the limited visibility into interprocess communication bandwidth that was available via Remos's networking monitoring and our inability to disable a number of stabilization strategies built into DeSiDeRaTa. Still, we were encouraged by the partial results and believe that a strategy based on intelligent resource management can be made to work with suitable changes to the resource management components.

6. Conclusion

This paper has explored the use of resource management strategies in supporting the testing and certification of real-time, fault-tolerant, mission critical systems based on distributed object middleware. Relatively simple strategies can be used for systems currently under development. There are also promising longer-term strategies that should be investigated in both engineering and research contexts.

7. References

[1] http://quite.teknowledge.com
[2] http://www.darpa.mil/ito/research/quorum/
[3] http://www-2.cs.cmu.edu/afs/cs/project/cmcl/www/remulac
[4] http://desidrta.uta.edu/~project/

— end —