Run-Time Reallocation of Computing Resources in a Heterogeneous Networked Computing Environment[1]

Final Report[2]

by The Open Group.
Douglas Wells, Principal Investigator
<d.wells@opengroup.org>

Report Abstract

Report developed under SBIR contract for topic N01-079. This SBIR investigated the use of COTS real-time OSs, real-time Java, the Open Group's real-time group communications facility, and a real-time node failure detector to address the needs of Navy computer systems for real-time, fault-tolerant, distributed systems in heterogeneous environments based on COTS hardware and software components. These real-time facilities were evaluated and found suitable for use in real-time, fault-tolerant systems with sub-second deadlines. In addition, the work identified a requirement for a development framework and tool-kit that would provide common facilities for real-time, real-fast communication, failure detection, instrumentation and visualization, and distributed clock management. This framework would be based on existing standards, including UNIX, POSIX, CORBA, UML, and would support applications written in Java as well as other languages. Discussions with several potential customers and vendors indicate that this technology would be useful in multiple military and civilian applications, including battle management, telecommunications, securities industries, and plant control. This document also describes a technology transition plan for evaluating future development on an ongoing basis.

Introduction

This document describes the results of an SBIR Phase I effort that explored the development of a product to enable fault-tolerance within distributed, real-time systems using the Java language. This effort investigated the use of recent technology to address the needs of distributed real-time, fault-tolerant, mission-critical systems in heterogeneous environments (referred to as mission-critical systems herein). Emphasis was placed on the interaction of two particular components: The Open Group's CORDS/GIPC group communication protocol framework and tool-kit; and "real-time" Java Virtual Machines (JVM) based on the recently developed Real-Time Specification for Java (RTSJ).

Our exploration of the technical aspects of CORDS/GIPC and Real-Time Java indicate that they can effectively address the problems that were identified in the original SBIR solicitation (N01-079). Analysis and demonstration experiments indicate that both components can meet the timeliness requirements of a significant number of potential

[1] Sponsored by Naval Sea Systems Command under contract N00178-01-C-3032

[2]This version of the final report contains only the technical sections of the original document.

commercial and military applications, and that each component can be used in junction with other components to build mission-critical systems. Furthermore, each component is at an early stage in its development cycle and can be expected to improve significantly.

Concurrently, we discussed the use of such technology with a number of potential military users. A common theme has been that they have an interest in the Java and group communication capabilities as proposed, but that the components were needed within the context of a development environment that would support the needs that are common to many developers of distributed, real-time, fault-tolerant mission-critical systems.

Recommendation

The investigation performed as Phase I of this effort has shown that it is technical feasible to develop a framework and tool-kit for building distributed real-time, fault-tolerant, mission-critical systems in heterogeneous environments using other COTS components. The investigation has also identified commercial interest for commercializing and marketing the results in a form that would be available to the government at COTS software. This commercial product would address both the initial development and the continuing development and life-cycle maintenance of evolving mission-critical applications, which would be written primarily, but not exclusively, in Java.

The proposed framework would be designed to:

- Support distributed, real-time, mission-critical applications,
- Interface to other systems, including other languages, hardware platforms, and distributed system technologies (e.g., Ada, C/C++, single board computers)
- Interface to development tools for real-time systems (e.g., scheduling tools, UML)
- Adhere to standards where possible, e.g., CORBA, POSIX, UNIX, Java
- Allow multiple, alternate implementations of each functional area in order to support applications with diverse requirements

The tool-kit would include capabilities for supporting real-time systems, such as:

- Real-time JVM
- Real-time communication interfaces, including group and publish/subscribe
- Real-time, real-fast networking interface
- Failure detection and management
- Instrumentation collection and visualization
- Distributed clock management

Particular components that we propose to include (but not exclusively) are:

- TimeSys's JTime real-time JVM
- The Open Group's CORDS/GIPC group communication capability
- The Open Group's FFD real-time failure detector
- The Open Group’s ETAP instrumentation collection facility
- UML modeling tools, such as Rational Rose or Rhapsody

There are a number of other tools that could utilize the common capabilities of the framework. We expect that some of them will evolve into useful products:

- Characterization support, e.g., timing benchmarks, maximum latency tools

- Test environments, including load stressers and component fault inducers

Finally, we note that the framework product would need to include:

- Documentation of the framework model
- Sample worked cases

Problem Statement

Heterogeneity

Modern mission critical systems are complex and distributed, involving multiple components on a heterogeneous mix of computer system platforms. The existence of multiple environments inserts additional complexity into all life-stages of the system: the original design must accommodate the processing model of each environment, separate programming is often required for each environment, each such environment much be individually tested, and separate distribution packages are required for each environment. The result is increased development time and costs, and a higher error rate during system operation.

On the other hand, the heterogeneous nature of the systems is due to the value inherent in specialization. Some computer systems are optimized for operating user interfaces; others include special capabilities for real-time environments. Some network components provide world-wide access; others provide high-bandwidth delivery that is uneconomical except in local area networks.

One particular benefit of a distributed, multiple component system is that the multiple components can be configured to provide redundancy in support of providing highly available, fault-tolerant system services. Designing and implementing fault tolerant systems is a specialized skill, however, and most successful fault-tolerant applications are build upon system tools that were designed and implemented by developers experienced in fault-tolerant systems.

Identifying and eliminating unnecessary heterogeneity pays off in simpler, more effective system operation. Identifying and enabling useful heterogeneity results in higher performance and a more efficient system. The problem is to determine which is which, and then to accomplish both. A middleware solution that accommodates necessary heterogeneity while supporting fault-tolerant applications would be useful in both military and civilian applications.

Origins of Heterogeneity

Multiple Processor Types

The issue of dealing with heterogeneous systems arises in several forms in modern military (and civilian) systems. The first form is the existence of computers with different instruction set architectures (ISA). Most source code for large software systems is written in a high-level language. Compilers then translate that source code into the low-level machine code that can actually be executed by the processors within those computers. Still, most languages and almost all implementations of those languages allow differences to occur in the execution of the resulting programs. C and C++, for example, do not require that all variables be initialized and do not specify the value that is assigned to those variables if they are not initialized. Almost all implementations reuse space on a common stack and simply allow a variable to take on whatever value happened to previously exist in the memory location occupied by the variable. Some compilers provide special modes to check for such uninitialized variables, but it is not possible, in general, to detect all such cases.

Multiple Operating Systems

Another form of heterogeneity occurs due to the use of multiple operating systems. The primary current example is the co-existence of both Microsoft Windows and UNIX operating system families. Both systems have approximately equivalent capabilities and both have similar execution models. Still, system code that attempts to operate on both types of systems is riddled with special cases, particularly *#ifdef*'s, in the case of C and C++. On the other hand, the windowing models are significantly different, and graphics-oriented code, such as GUIs, is almost impossible to make similar while adhering to each system's native windowing model.

Even common operating system families have significant differences. While a core set of common support routines exists on all versions of Microsoft Windows, numerous procedures that are useful for system programming tasks exist only on Windows NT and not on Windows 95/98/ME, and vice versa. Also, there are variances in various subsystems. For example, Microsoft Windows 95/98/ME only supports 8-bit code sets (primarily ASCII), while Microsoft Windows CE only supports the 16-bit UNICODE code set.

A similar situation exists for UNIX and UNIX-like systems. Although POSIX defines a useful common set of functions, many functions relative to systems programming tasks were purposely omitted in order to enlarge the set of systems that could comply with the standard. Even the more extensive and stringent UNIX standards (which are maintained by the Open Group) do not cover all aspects of systems programming and do not attempt to address differences in implementations. For example, some common UNIX implementations are limited to using only 256 file descriptors in executing select calls, while others allow over 10,000. Some UNIX implementations make it easy to receive raw Ethernet packets (which might be used to effect automatic network configuration); others make it difficult. Finally, the Linux community is developing a set of standards that largely conforms to the UNIX standards, but is slightly different.

Multiple Operating Versions

All of the above problems with heterogeneous systems are exacerbated by differences between versions. The commercial chip and hardware manufacturers find it necessary to continually upgrade components to reduce manufacturing costs. Silicon chips are so complex that the hardware bugs have begun to resemble software bugs. Some Intel chips, for instance, now include a capability to load a patch file into the processor at boot time, in a manner very similar to what was used on 1960's mainframe computers. As peripheral chip manufacturers introduce new products, or phase out certain products or as those manufacturers go out of business altogether, board manufacturers must track those changes, often replacing multiple components with one new chip, or a specially designed ASIC.

System vendors must modify their operating systems to include the changes that are required by the modified hardware. Simultaneously, they are fixing bugs and otherwise enhancing the operating system. Rarely are those changes, other than the security-relevant ones, retrofitted into older versions of the operating systems. Middleware vendors must maintain compatibility with the upgraded operating systems, and even though they might attempt to not depend on new operating systems features, inevitably some change will slip by.

As older hardware fails and the repair store supply dwindles, only the latest versions of the hardware are available as replacements. Often, in fact, the newer versions of the hardware have higher throughput and lower power utilization, such that there is a benefit in installing the newer versions. These newer hardware systems, of course, require newer versions of software. All of this leads to version creep and generally guarantees that any large distributed system will include multiple systems at different version levels.

Thus, large, complex systems, such as Navy battle management systems, will almost certainly operate in a heterogeneous environment. The heterogeneity will encompass disparate hardware, operating system, middleware and application systems. Successful system deployment requires that this diversity be accommodated in the system design.

Evolving Applications and Requirements

One source of heterogeneity that can be expected to increase over time is the evolving nature of mission requirements and application support. As applications increasing interact with each other and as software upgrades occur more often in fielded systems, it will become increasing difficult to test these mission-critical systems in exactly the configurations that occur in each installation. This situation will be exacerbated by the other, uncontrollable sources of heterogeneity described above. Thus, applications will have to operate correctly and efficiently in environments with varying hardware capabilities. Furthermore, they will have to share hardware resources with other applications that are fulfilling their own objectives.

Fault Tolerance

Normal Operation w/ no State Information

Military battle management systems are similar in many respects to the information systems in many commercial organizations in that the computer systems are mission-critical. Any lapse in operation of the computer system results in a failure of the organization to meet its objectives. Yet, these systems are complex and distributed, incorporating thousands of components, each of which has a limited lifetime and can be expected to fail according to a random, though statistical, pattern. If failure of individual components were allowed to cause the failure of the entire system, the overall system would unreliable as to jeopardize the overall mission.

Instead, the computation and communication infrastructure in such large, complex systems is designed to tolerate and isolate failures. If one computer node fails, other computers eventually recognize that it has failed, pass that information on to applications (perhaps as a communication failure), and continue to process other computation tasks. If one network fails, routers and gateways between the networks will notice that the network has failed, pass on that information to other parts of the system, and continue to process other communication tasks. It is possible that neighboring computers and networks will be affected by the failure, but generally, the farther away a failure is, the less likely it is to affect operation.

This mode of operation works because the separate components are largely self-contained and have few dependencies on other components. A network router might depend upon a nearby host to provide its initial parameters (such as the types of communications lines and the static routes associated with the local network address ranges). Typically, however, this information is only needed at bootload time, and the supporting host can go down and back up multiple times in the interim without interfering with the operation of the router.

In fact, this ability to tolerate simple failures is leveraged to support normal maintenance, such as backups or system upgrades. Computer nodes can be taken off-line for normal maintenance, either software or hardware. While the services provided by the node are no longer available, other parts of the system continue otherwise normal operation. When the maintenance is completed, the computer node is reintroduced into the network and normal functioning can proceed.

Maintenance of Distributed State Information

The maintenance of information associated with multiple components is more difficult and is typically left to the applications. Consider a web server with mostly static content, for example, providing on-line access to recently released opinions from the U.S. Supreme Court. Such a web server can be provided with multiple DNS addresses (so-called round robin addresses). In this case, if a web browser fails to make a connection to one the original address, it will eventually timeout and attempt to connect to the next address, which will succeed. Assuming the second host has been supplied with the same files as the first, the user of the web browser often will not notice the switchover, other than the one-time delay due to the timeout.

When dynamic state must be supported, however, the problem becomes harder. Consider the same web browser, but now assume a banking application. Assume that a user has used the web interface to transfer money from another bank to this bank. Now, if the computer node goes down and another node replaces its function, the user will not be pleased to find that his money is nowhere to be found. Thus, it is necessary, for many applications, to maintain state across multiple machines in a manner similar to transaction processing as provided by commercial DBMSs.

Maintenance of the banking application data is relatively straightforward. The money transfer activity can be described by 100 or 200 bytes of data, and the user is willing to delay his next activity for normal "human response" times of a few seconds. In fact, the user probably won't even notice if his account is "frozen" for several seconds while the information is transferred to a second system. Even if several thousand users are performing such transfers simultaneously, the information can be transferred to the other machine and acknowledgements can be returned within those few seconds that the user is willing to wait.

Applications with larger data sets and/or shorter time constraints present a more difficult situation, however. Consider a typical radar tracking application, for example. Assume that there are 10,000 targets being tracked (as might happen in TBMD), with new reports being received for each target every 10 seconds. Further assume that each report can be characterized in 100 bytes and that the application maintains a history of the most recent 10 reports. If we design a solution that blindly copies over the entire data set (perhaps as stored in a track file object), we find that even a dedicated gigabit network will introduce a significant delay in the processing. Also, processing of the target reports is limited to the capabilities of the one machine.

A smarter solution is to take advantage of application-specific knowledge. The first thing to note is that the historical track reports are redundant. Since they are not going to change, there is probably no purpose in retransmitting them. The tradeoff for only transferring the new track reports is that the receiving system must repeat the original processing in order to regenerate state information, such as the Kalman error residue. The choice of the more effective solution, however, depends on the resource constraints of the processing system: it might be more effective to transmit the entire state-or it more be better to duplicate the processing at the second node, or there might even be other alternatives that are better.

An Analysis of the Problem in the Context of Navy Application

Although many aspects of military combat systems are similar to commercial systems, there are numerous differences that can affect the selection of solution patterns for Navy battle management systems, such as are planned for future Navy ships,

Complex, Real-Time Systems

One distinguishing feature is the real-time nature of the systems. Generally, commercial systems can be real-time or they can be complex, distributed systems. There are few commercial systems that are both. Military systems, on the other hand, are often both real-time and complex, distributed systems. The hallmark of future military systems will be the capability of correctly operating in more complex contexts than the enemy can.

The purpose of military weapons is to disable the enemy's fighting capacity, which is usually performed by destroying the enemy's weapons and often by killing enemy forces. Because the enemy has similar but opposing goals, military combat systems are often life-critical. The AAW system contemplated for future Aegis systems, for example, must process incoming radar target reports to monitor existing traffic and to detect new threats, follow the progress of several in-flight missiles, and manage the targeting of illumination radar for each in-flight missile. If the real-time constraints are not met, the threats will not be detected and/or the missiles will not destroy the threats. Successful incursion of the missile into the ship will result in millions of dollars worth of damage and the loss of human life. Perhaps even more importantly, the ship could be disabled and its defensive capabilities eliminated. This would imperil the next ship, and then the next, and then the next.

Failure Characteristics

Most commercial systems are designed to recover from limited failures. Many commercial DBMSs, for example, log data to a special disk in support of transactions. If the computer goes down for any reason, it can use the data on the disk when it comes back up in order to determine the state of computation and continue from that state. RAID disk controllers provide protection from disk driver failures. Still, these systems typically only protect against one failure at a time. If the computer goes down and the logging disk fails, the transactions are lost. If the RAID disk controller itself fails, access to the data is unavailable until it is replaced. If two disks fail, a subset of the data is also lost.

Even commercial fault tolerant systems, such as Tandem and Stratus, do not protect against certain situations with multiple failures. Stratus' VOS system, for example, does not (by itself) protect against physical damage to the system processor. Tandem's Guardian system does not protect against concurrent failure of two (physically collocated) processors that happen to contain any pair of primary and the backup processes that are part of an application.

Navy combat systems, however, are expected to encounter such situations. Indeed, they are also expected to maintain rated performance. Any incoming weapons that manage to hit the ship can be expected to destroy most or all of the processing capability within the immediate battle damage confinement area. Thus, for any particular application task, it is useful to place the primary and backup processing capabilities in physically distant locations, preferably separated by one or two bulkheads. These damage confinement bulkheads, by their very physical nature, are likely to limit the incursions presented by communication cables. In practice, this will likely mean that communication between primary and backup processors will occur via shared, high-speed, routed networks, such as Ethernet and its higher speed cousins, including gigabit network over fiber optic cables.

Technical Approach

During the Phase I effort, the Open Group investigated its proposed solution for the use of middleware support for real-time, fault-tolerant distributed systems. The proposed solution combines recent developments in the Java and CORBA communities with the real-time capabilities of the CORDS/GIPC group communications system. By enhancing those basic technologies with components that provide easily accessible fault-tolerance methods, the resulting system would be useful in military combat systems as well as commercial environments, such as factory floor automation and stock exchanges.

The technology base of the proposed system comprises an implementation of real-time Java combined with the CORDS/GIPC group communication system. The capability of CORDS and GIPC would be made available via Java objects. In addition, additional objects would be created that would utilize the basic GIPC capabilities to provide simple methods for implementing object replication, and active and passive backup services. The combination provides a productive environment that eliminates many of the problems associated with mixing multiple hardware platforms and/or multiple operating environments. It also provides an effective tool base for implementing efficient, fault-tolerant applications.

Real-time Java

A fundamental part of the strategy is the use of a real-time version of Java. Java provides an execution environment that is largely independent of both the underlying hardware platform and of the operating system. The Java class object is machine independent: the exact same executable object can run on any Java machine. Each object runs in the context of a virtual machine that provides identical behavior for almost all machine characteristics, including integer ranges. The use of Java is becoming very popular, and Java products now exist for many environments, including constrained embedded systems, such as set-top boxes, to World Wide Web server applications.

The creation of a version of Java capable of supporting real-time applications with specified time constraints was the first enhancement proposed for Java. The result was published in June, 2000, and several implementations of the real-time specification are currently under development by members of the specification expert group. Substantial portions of the reference implementation, which is being developed by TimeSys, have already been released and the rest is expected soon.

Real-time Java incorporates an innovative method for dealing with the garbage collection problem. Certain real-time threads can be declared to be higher priority than the garbage collector and can, therefore, defer the execution of the garbage collector. To prevent deadlock, these threads must allocate space only from certain reserved heaps, which are not subject to garbage collection. A method is then provided to allow information from the non-garbage collected heaps to normal processing space. Although this method is highly promising, the ability for real-time application programmers to grasp the concepts and use it effectively remains to be proven.

Group Communications

Group communications is a model of communicating between multiple computers. Simplistically described, group communications provides the ability to do an atomic broadcast within a group. Such an atomic broadcast in a distributed environment can be compared to a transaction for traditional DBMSs. A group is a set of cooperating processes that want to communicate with each other. An atomically broadcast message is directed toward a group, and is either delivered to all group members or none. Applications can then leverage this basic function in order to easily implement application-specific fault tolerant strategies.

One of the earlier implementations of group communications is the ISIS system, which was developed at Cornell University. ISIS was spun off into a commercial company and was widely used on Wall Street for financial transactions systems. The commercial product appears to have been discontinued. Cornell later developed Ensemble, a research system similar to ISIS with additional capabilities. Ensemble is available for general use.

The Open Group has developed CORDS and GIPC. CORDS is a general-purpose framework for developing communication protocols. GIPC is a Group IPC system that supports group communication. A distinguishing feature of CORDS and GIPC is that they have been designed to support group communications in real-time applications. A prototype demonstration system that uses CORDS and GIPC to manipulate a ball sorting apparatus can detect and recover from a node failure in less than 400 msecs on normal PC hardware interconnected using 10Mbps Ethernet.

CORDS was developed by the Open Group Research Institute in the mid-1990's. It was originally adapted from the x-kernel, which is a communication framework that was created at the University of Arizona. A version of CORDS was acquired by DASCOM and shipped commercially as part of a security router product created by DASCOM. Subsequently, the Open Group Research Institute conceived and implemented the concept of *paths*, which reserves CPU and buffer resources for certain threads and enables operation in real-time applications. Using the *paths* concept, we then developed GIPC, which was specially designed to support group communications in real-time applications.

Phase I Technical Objectives

The technical objectives identified for the Phase I effort were to:

- Identify the requirements for middleware support of complex, real-time, fault-tolerant distributed systems in military and civilian applications,
- Evaluate and qualify the components slated for use in such as system,
- Create a technical design that addresses the identified requirements, and

Requirements Identification

We discussed the proposed system with several military system integrators. They expressed interest in the particular products, but they also identified a larger problem. Their interest in these products derives from a goal to develop more comprehensive, network-centric weapons systems. They want to build systems that are more flexible, more adaptable, and more maintainable. They are being encouraged to build components that can be composed into multiple systems, which can support many diverse missions. In short, they want to abandon stove-piped systems.

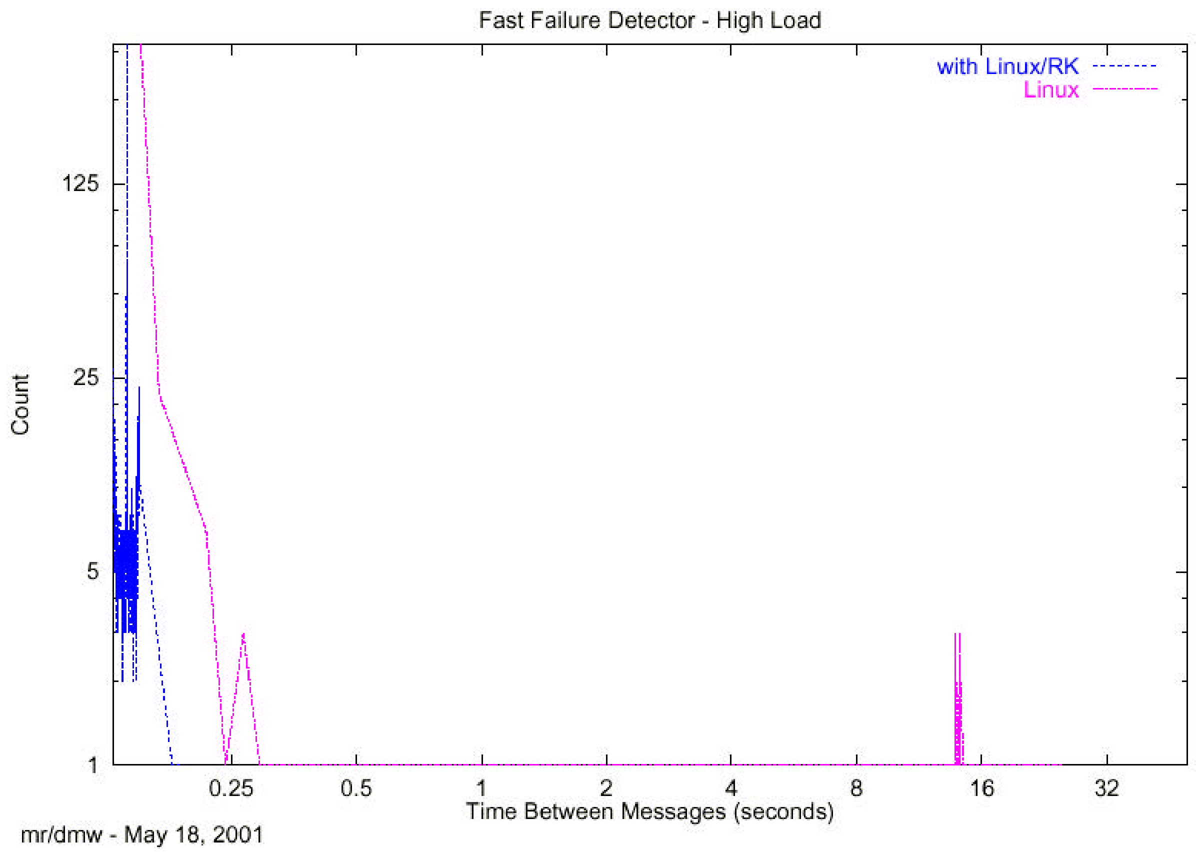
There is no corresponding reduction in system requirements, however. The developers must still satisfy stringent performance goals, including handling increased numbers of targets with higher availability. In addition, these flexible, adaptable, highly configurable systems must continue to be certified to perform properly in field conditions. These users confirmed that this larger need can best be met by a common development environment based on a flexible framework. This framework would be populated with a tool-kit that includes the Real-Time Java and CORDS/GIPC components from Phase I, as well as other, commercially available or privately developed tools that aided development of these mission-critical systems.

These users also emphasized that they need to reuse their own components. Having developed a capability for one application, they want to easily adapt it for another. When a commercial component is unavailable or unsuitable, they want to develop their own components and easily incorporate them into the current application as well as future ones. Thus, the proposed framework must be extensible.

There is a similar need in the commercial world. Although businesses do not normally encounter the same system certification requirements, the difference is one of formality rather than necessity. Failure of mission-critical systems in the business world can cause huge monetary losses, so the commercial world shares an interest in ensuring that its systems will behave properly, maintaining stable, predictable services even in previously untested configurations. Potential customers include telecommunications, factory automation, refinery operations, etc. We have also discussed these requirements with the Security Industries Automation Corporation (SIAC), which operates the New York and AMEX Stock Exchanges. The developers there emphasize the need for an effective group communication system as well as the requirement for precisely managing the hardware resources and software applications.

Component Evaluation

We began evaluating real-time operating system (RTOS) components. Since we have isolated many of the tightest time constraints into the Fast Failure Detector (FFD) component (see below), we were particularly interested in the behavior of the FFD component with various RTOSs. The first RTOS that we evaluated was Linux/RK, which is the research version of TimeSys's real-time Linux, which is the OS that will support the real-time Java implementation that we intend to use. We first configured FFD to generate messages at 100 msec intervals with a 500 msec timeout. FFD messages were typically delivered with less than 100 msec delay, and no messages exceeded the 500 msec timeout value. We then generated a very significant background load (load average > 126). Under this load, FFD messages were typically delivered with less than 250 msec delay, and again no messages exceeded the 500 msec timeout value (see Figure).



We also briefly tested Solaris and Windows NT, although the systems were supporting normal development loads, rather than being evaluated in a test-bed environment. Using the same FFD configuration operating at near-maximum real-time priority, we found that no Solaris messages exceeded the 500 msec timeout value, even over several days testing. Using the same FFD configuration operating at real-time priority, Windows NT would normally not exceed the 500 msec timeout value. However, certain usage patterns, usually involving the graphics display and/or the mouse, would cause the Windows NT FFD to exceed the 500 msec timeout value. In fact, in certain situations, the FFD component would declare itself to be down.

Although TimeSys's Real-Time Java JVM product is not available, they have released a reference implementation (RI) to the Real-Time Specification for Java (RTSJ) expert group, which is developing the specification. We do not yet have access to the RI, but we have been working with The MITRE Corporation, which does have access. MITRE has borrowed our ball sorter apparatus, and we assisted them in developing a demonstration application written in real-time Java. In June, 2001, the demonstration apparatus was shipped to the JavaOne conference in San Francisco, where it was highlighted in TimeSys’s booth.

(The ball sorter apparatus was developed as a demonstration of the real-time and fault-tolerance capabilities of our MK 7 and CORDS/GIPC components. Multiple colored golf balls are dropped through a tube, where they are detected and their color determined. At the end of the tube, flippers divert the falling balls into separate collection buckets based on their color. In the original setup, the color detection algorithm was executed on multiple, redundant computers that communicated using group communication. One or more computer nodes would be reset while balls were dropping, and the failed nodes would be detected and recovered within a bounded time (approximately 400 msec. In the JavaOne demonstration, the apparatus was used to demonstration that real-time Java applications could respond to the ball drops within required time periods.)

We also evaluated and improved a Fast Failure Detector (FFD) component, which we have also been developing under the DARPA Quorum Program. Originally developed for the Linux/RK operating system (which is similar to the TimeSys real-time Linux product), we ported the FFD component to Solaris and provided it to NSWC's HiPer-D project. They have installed the component in their test-bed and are reporting successful use with timeout periods on the order of 90 to 160 milliseconds in a test-bed based on Gigabit Ethernet. The HiPer-D project expects to use the FFD component in the AutoSpecialSM Doctrine AAW path of its simulated Aegis system in its upcoming 2001 demonstration.

Technical Design

Framework

A good framework is elegant. It comprises little code itself, leaving the bulk of its capabilities to the components that fit into the framework or tools that utilize those components. A good framework imposes low system overhead while providing powerful capabilities through both native tools and third-party plug-ins. The important principal, however, is that it be both useful and simple. It must be useful in the sense that the tools that are available via the framework are powerful and satisfy the functional requirements of the application. It must be simple in the sense that the developers understand the interfaces and find it easy to use the tools.

The particular details of the interfaces of are difficult to predict in advance. Most successful analogous frameworks have started with a few basic interfaces, and then experimentation leads to improved interfaces over time. UNIX, for example, began with a few well-proven interfaces (e.g., open/close/read/write), and one extensible interface ioctl. Vendors used this interface to define additional standardized functions. For example, the terminal interface evolved through various interim phases (e.g., *sgtty*, *termio*, *termios*) until it reached now standardized POSIX interfaces (e.g., *tcgetattr*, *cfgetispeed*).

We would expect the proposed framework to evolve similarly. Each functional area covered by the initial version of the framework would provide a few basic, common, well-proven interfaces and one or a small number of extensible interfaces. (Some examples are provided below.) As we populated the framework with tools, we would adapt the interfaces to aggregate common functions. In addition, we would expect trial use by external organizations, such as NSWC's HiPer-D group, to provide feedback on the usefulness and understandability of the interfaces.

Group Communication

The CORDS/GIPC group communication package has been proven in both demonstrations and commercial applications. We do not expect that it will need major restructuring. It will need to be ported to the target platforms, and we would expect to add additional, specialized protocols Also, Java interfaces based on the the newly developed RTSJ interfaces will need to be developed in order to maintain the real-time properties of the CORDS/GIPC delivery mechanisms.

We would also expect to use the CORDS/GIPC group communication interfaces to seed the framework. The real-time CORDS/GIPC group communication interface, for example, shares a common set of functions with other group communication packages. Examples include Create/DestroyGroup, Join/LeaveGroup, Post/ReceiveMessage. In addition, it requires a few additional functions to reserve the resources needed to satisfy its time constraints and to provide the visibility and control to low level functions as required by developers of real-time applications.

Fault Management

As mission-critical systems become more complex, there are more interactions between applications, and any single application depends upon more hardware components. Large networks of COTS hardware components will experience hardware failures on a regular basis. When such complex systems run on large networks of COTS hardware components, hardware failures must be expected can planned for. A similar argument can be made about software failures. Thus, the detection, isolation, and recovery from hardware and software failures will be an increasingly important part of system design. Our initial component in this area will be node failure detection.

It is useful and convenient for multiple, cooperating applications to share a common view of the state of another component host. The group communication paradigm is one means of creating this common view. Unfortunately, detecting host failures with low values for timeouts requires large rates of network traffic with very low latency thresholds (at least for COTS real-time systems). The Fast Failure Detector (FFD) component isolates this function so that is can be provided an appropriate level of resources and then notify all other components that have registered their need for notification of host failures. The framework interface is quite simple, requiring interfaces only for registration and notification.

Instrumentation

Instrumentation is necessary to the development of a useful, high performance product. Many applications and many components already include instrumentation. What is often missing, however, is the ability to derive information about the interactions between components. The instrumentation logging formats are often incompatible, using different time bases and different formats. Usually, the differences are simply due to the ad hoc nature of the instrumentation. Sometimes, the differences are due to different operational environments. (For example, operating system device drives can not tolerate page faults and must use staging buffers of limited size.)

We would expect to build upon the ETAP (Event Tracing Analysis Package) instrumentation package. ETAP includes mechanism for managing and merging information from both operating system kernel internals as well as application components. In addition, ETAP is designed to allow merging of instrumentation information from real-time components on multiple nodes within a network, and its utility has been proven by its exposure in the Navy's HiPer-D test-bed at NSWC Dahlgren.

ETAP was originally developed by the Open Group as part of the MK 7 operating system. (MK 7 is a microkernel-based real-time version of UNIX also developed by the Open Group. All or portions of MK 7 have been shipped commercially by companies, such as DASCOM, Hitachi, and Apple).

CORDS/GIPC already includes "probes" to provide information to various instrumentation packages, including ETAP. In addition, we have been working closely with System/Technology Development Corporation and would expect to integrate the ETAP-based instrumentation capabilities with S/TDC's QMS (QoS Metrics Service) instrumentation package, which is based on XML and the CORBA event channel facility.

Real-Time Java Virtual Machines

We expect that there will be several real-time-capable JVM (Java Virtual Machine) products released over the next few years. Many of these products will be based on Sun's Real-Time Specification for Java (RTSJ). The draft specification was released more than a year ago, and the JSR-1 expert group, which is developing it, has addressed many of the concerns that we expressed in our original Phase I SBIR proposal. As a result, RTSJ seems to be a reasonable environment in which to develop real-time applications.

The JSR-1 group has limited itself to specification of real-time extensions to Java and to requirements of behavioral characteristics of the JVM. This has been a good strategy in that it will result in a usable product within a relative short period. Nonetheless, there will remain a substantial body of useful functions that will not yet exist. Some of these functions are being developed by other JSR expert groups and can be adapted to the needs of the proposed environment. Others will have to be specially developed. Notable examples include interfaces to group communications, to globally synchronized clocks, and to instrumentation.

Another area that will require development is networking for real-time applications. The JSR-1 effort has not addressed this area, leaving it to the JSR-50 expert group, which is developing the Distributed Real-Time Specification for Java (DRTSJ). (The principal investigator of this Phase I SBIR effort is a key member of this JSR-50 expert group.) Commercial realizations of the results of the DRTSJ effort can not be expected for several years. Nonetheless, there do appear to be useful subsets of Java network communication that can be utilized within time-constrained applications if appropriate protocols and Quality of Service (QoS) choices are selected. We would expect to develop these real-time networking capabilities as part of the proposed framework and toolkit.

ACE and TAO

ACE (Adaptive Communication Environment) is a portability layer than provides a common (“glue”) interface to numerous real-time, as well as non-real-time, operating systems. TAO (The ACE ORB) is a standards-compliant, real-time implementation of CORBA. While TAO is oriented towards C++ applications, the ACE component has proven to be particular useful for allowing application portability across a broad range of operating system implementations as well as operational environments. We would expect to include ACE, and possibly TAO, in our tool-kit as well as make use of it in developing toolkit components that were coded in C and/or C++.

— end —