Run-Time Reallocation of Heterogeneous Computing Resources

A submission to Navy SBIR N01-79
by The Open Group Research Institute
10 January 2001

A. Identification and Significance of Problem or Opportunity

A.1. Problem Description

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 built 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.

* * *

We begin with separate overviews of the various types of heterogeneity that occur in modern systems and of some design issues that must be considered in the design of fault tolerant systems.

A.1.1. 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 GUI's, 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 to 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.

A.1.2. 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 a failure of

individual components were allowed to cause failure of the entire system, the overall system would be so 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.

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 addressing). 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 DBMS's.

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 might be better to duplicate the processing at the second node, or there might even be other alternatives that are better.

A.2. 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 DD-21,

A.2.1. 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 the DD-21 system, for example, must process incoming radar target reports to monitor existing traffic and 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 DBMS's 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 fails, 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 can be 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 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.

A.2.2. Heterogeneity

Several technical approaches should be considered to address the issues introduced by heterogeneity. We will use the contemplated military systems as examples in order to provide a practical perspective. We will begin with a review of the current situation.

Separate Compilation

Most software today is written in a high-level language. Often the language is a standard, commercially supported language, such as C, C++, or Ada. Sometimes the language is special to the military. Usually, the source code is translated by a compiler into object modules that contain processor-specific instructions. These object modules are then link-edited, joined together and combined with object modules from common libraries, to form executable programs. Finally, these executable programs must be distributed to each ship and loaded on the appropriate computers. If a component is replaced in the field, the appropriate executable programs much be identified, located and installed on the replacement component.

Due to differences in the compilation and link-edit tools and environments, each distinct flavor of executable program must be specifically tested, both for individual correct operation and for interoperability with other flavors. Some characteristics that indicate that separate testing is required include:

- processor type, such as HP's PA-RISC and Intel's Pentium
- operating system type, such as HP's HP-UX and Microsoft's Windows NT
- operating system release, such as HP-UX 10 and HP-UX 11,or Windows NT and Windows 2000. Different Windows NT's service packs can also impact function.

These application executable programs also normally require common shared libraries, which are typically installed as part of the vendor's operating system. They can also require the presence of additional middleware, such as CORBA, DCOM, or Web servers. The need for this additional software must also be identified and the appropriate software must be located and installed on each system. (Variances in versions of this software can also be indicators of the need for additional testing configurations.)

Fat Binaries

So-called *fat binaries* are executable programs that contain machine code for multiple different processor types. Perhaps the best known instance of this deployment strategy was NeXT, which created executable files that contained machine code for both Motorola 680x0 and Intel ix86 processors. Typically these files were produced by a vendor-produced compiler, which could use a common front-end and then separate compiler back-ends to translate the intermediate code into the multiple machine-specific instruction sets.

NeXT deployed these files in order to simplify distribution of its own software. Several problems remained, however. First, there were significant differences between the

hardware platforms. NeXT had built special computers with the 680x0 processors with specialized graphics and printer support components. The ix86 platforms were IBM-PC-compatibles. The processors were also significantly different: the 680x0 was big-endian, the ix96 was little-endian; alignment requirements were different, such that one machine type would fault in situations that the other would not. In the end, while this strategy might have reduced NeXT's distribution costs, it did not seem to reduce the necessity of separately testing multiple system flavors. In addition, third party suppliers did not widely adopt this strategy. Thus, users had to separately identify, acquire and install packages for each processor type. Finally, many customers were surprised and complained about the need for significant amounts of additional disk storage.

Processor Emulation/Simulation in Software

The machine code instructions for one processor can be interpreted and executed by another. Many such emulators exist, particularly in the open source world for various historical systems. Only a few commercial versions, however, seem to have been integrated into the standard execution environment. One example was Apple's conversion from the Motorola 680x0 to the to the IBM/Motorola PowerPC. When the PowerPC version of the Mac operating system detected that it had been told to execute an executable program in 680x0-form, it instead invoked a software interpreter for the 680x0 instruction set. This appears to have been an attempt to ease the development process at Apple as successive releases of the OS have eliminated more and more of the 680x0 code. This migration strategy seems to have worked well for Apple, but they benefited by having moved to a significantly faster processor. Still, user benchmarks indicated that 680x0 code executed between 5 and 100 times slower.

Digital Equipment Corporation used a similar strategy in converting from the VAX to the Alpha processor. Digital, however, applied proprietary compiler technology to the problem and translated the VAX instructions to equivalent Alpha instructions. Because the Alpha processor was significantly faster than the VAX, the emulated VAX code often ran faster than the original. Digital also applied this compiler technology for its port of Microsoft's Windows NT to the Alpha processor. In that case, the emulated ix86 code was reported to run at rates from half speed to parity.

Processor Emulation/Simulation in Hardware or Microcode

Whereas the emulation strategy described above emulated other processor types using ordinary programs usually running with no special privileges, some machines perform execution using special hardware or using specially privileged code. Early examples include IBM's emulation of its earlier 709x and 1401 processors on various models of its newly introduced 360 processor family. The particular 360 models had been specially designed to provide the emulation, and the migration strategy seemed to have worked well.

A modern instance is IBM's AS/400. There is no "native" form of this processor type. Instead, the defined instruction set is very high-level (very CISC), and IBM has interpreted the AS/400 instruction set using various native processors. At least one

current version of the AS/400 uses a variant of the PowerPC processor and pre-translates the AS/400 instructions into PowerPC instructions before execution. The result of the translation is not visible externally, and the user of the system has no access to the PowerPC level. To that user, the AS/400 executes its instructions natively, and the user can move his programs from one member of the family to another without undue difficulty.

Another modern instance is Crusoe from Transmeta. The Crusoe processor behaves in a similar manner to the AS/400. The chip appears to the user to be a clone of Intel's Pentium line. In fact, the Crusoe chip "compiles" the Pentium instructions to its native format and actually executes those native instructions. The Crusoe chip is relatively new and Transmeta has only partially revealed its marketing strategy, but it appears that users will have little or no access to the native level of the Crusoe chip.

Interpreted Code

Some languages can be executed from source, or from a minimally translated version of the source. Examples include BASIC, Lisp, Smalltalk, FORTH, Perl, and several other scripting languages. These languages are usually 2 to 10 times slower than equivalent compiler code. Some these languages have developed large followings in specific areas. BASIC remains popular for managing Windows, Lisp and Smalltalk remain very popular in the fields of Artificial Intelligent and fast prototyping. Perl is very popular for common system administration tasks. The usual problem with these languages is that their overall popularity is limited and there is limited support for third-party packages and middleware.

Most of the interpreted languages have associated compilers, which can boost their performance to near that achievable with languages with separate compilation. In this case, however, the compilers usually produce separate executable programs, and the same problems arise as are encountered with today’s separately compiled executables.

Virtual Machines

Another category is languages that require a significant compilation step, but which produce instructions for a standardized virtual machine. An early example was p-code, as produced by UCSD's Pascal compiler. The resulting module was machine-independent and could be interpreted in many different environments. A more recent example is Sun's Java. Also, Microsoft seems to be positioning its recently announced C# as an alternative to Java. Interpreting Java byte code seems to operate at a similar speed to Apple's 680x0 emulator, but because the instructions used by the virtual machine were designed specifically to simplify emulation, the worst cases have been eliminated, and interpreted Java code appears to execute from 5 to 10 times slower than equivalent native code.

Java also includes a capability for a flash compiler (a so-called Just-In-Time compiler). This compiler works from the virtual machine instructions and produces native processor instructions. Such code seems to execute only a factor of 1.5 to a factor of 5 slower than equivalent native code. Recently, several organizations have announced significant

progress in "hot-spot" compilers. This dynamic compiler technology produces code that promises to run with minimal speed differences from compiled native code. Each of these compiler technologies works from machine-independent Java byte code files. Thus, there need be only one form of executable module, which can then be run at near-native speeds on multiple processor types.

There also exist standard compilers for Java, which compile Java source or Java byte codes to native executable file format. This process produces executable programs similar is most respects to the separately compiled programs described originally. It should also be noted that several groups have produced or are creating Java processors, which execute Java byte codes as their native instruction set. So far, most of these processors have targeted small embedded system environments, such as set-top boxes or robotic systems.

A.2.3. Fault Tolerance

As noted previously, many commercially available systems that support fault tolerant operation do not provide adequate support for systems that are likely to encounter severe physical damage. There appear to be only a few systems that would support fault tolerant operation for systems separated by a significant distance.

CORBA Fault Tolerance

Members of the Object Management Group (OMG) recently completed the Fault Tolerant COBRA (FT-CORBA) specification, which implements an active replication model and will allow the replication of objects as managed within the CORBA model. Although several implementations are underway, including a few commercial versions, none are yet available.

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 DBMS's. 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 earliest implementations of group communications was 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 were designed to support group communications in real-time applications. A prototype demonstration systems that uses CORDS and GIPC to manipulate a ball sorting apparatus can detect and recover from a node failure in less than 400 msecs[1]

A.3. Proposed Solution

A.4. Technical Approach

The Open Group Research Institute believes that an opportunity exists to provide middleware support for real-time, fault-tolerant distributed systems and proposes to investigate this opportunity using Navy combat systems as prototype applications.

A.4.1. Overview

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.

We describe below each of the major building blocks that are expected to be in the design.

A.4.2. 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

---

[1] L.M. Feeney, P. Bernadat, F. Travostino, Characterizing Group Communication Middleware for Real-time Distributed Systems, IEEE 1997 Real-Time Systems Symposium, Kyoto, Japan.

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, ranging from 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 recently published[2], 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 IBM, 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.

A.4.3. CORDS/GIPC

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 has been 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 concepts we then developed GIPC, which was specially designed to support group communications in real-time applications.

A.4.4. ACE

The Adaptive Communication Environment (ACE) is an object-oriented framework and tool-kit for implementing distributed systems in heterogeneous environments. ACE implements patterns commonly used in such systems and simplifies development. In addition, the ACE API is generally independent of the underlying OS. Thus, use of ACE alleviates or eliminates several problems that arise due to the existence of a heterogeneous infrastructure. ACE was originally developed by the Distributed Object Computing group at Washington University at St. Louis.

---

[2] ..., *The Real-Time Specification for Java,* Addison Wesley, ISBN 0-201-70323-8

Although ACE was originally created as a research project and continues to be developed by groups as several universities, ACE is also available from and is supported by at least two commercial organizations. ACE has been successfully applied in several real-time projects, including at least one aerospace application. We anticipate using ACE in support of enhancements to the real-time Java and CORDS/GIPC components.

A.5. Benefits of Approach

A.5.1. In our approach, only one form of executable object exists, and that object can execute on any node regardless of the underlying platform.

The Java class object is machine independent: the exact same executable object can run on any Java virtual machine. Each object runs in a context that provides identical behavior for almost all machine characteristics, including integer ranges.

A.5.2. Our approach enables efficient fault-tolerant applications for systems with real-time constraints.

The application programmer can leverage the power of the group communications programming model to eliminate the need for brute force fault tolerance techniques, such as binary copies of data bases, that would introduce intolerable delays in system with real-time constraints. Instead, the programmer can easily take advantage of application specific characteristics that result in a viable system.

A.5.3. Our approach minimizing training costs for developers by leveraging the Java programmer base.

The use of Java is becoming very widespread, and Java products now exist for many environments, from constrained embedded systems, such as set-top boxes, to World Wide Web server applications. Many developers are learning Java, attracted by its ability to provide applet services via web browser frameworks.

A.5.4. Our approach leverages the existing base of Ada applications and developers in the U.S. military.

The Java virtual machine is not limited to executing programs written in Java. In particular, there is an existing commercial product that compiles Ada to the Java virtual machine instruction set. In addition, there is an open source project to create such a compiler adjunct for the Free Software Foundation's gcc compiler.

Thus, real-time, fault-tolerant applications could be written in either Java or Ada. This software would be compiled to Java class objects as an end-result of the development process. In either case, the resulting Java classes are processor-independent and will run on any machine with an interpreter and/or a compiler for the Java virtual machine. Optional optimization on each target machine would provide execution speeds approximating that of native compiled code.

A.5.5. Use of Java fosters creation of correct code.

Standard Java is a *safe* language in that a programmer can not overwrite portions of the auxiliary implementation. Thus, Java applications are not subject to many of the exploits that have been applied to C-based applications, such as sendmail or the Internet DNS servers. This safety has been provided as a side effect of Java's use of garbage collection as the model for dynamic memory management. Traditionally, languages that utilize garbage collection memory management have not succeeded in real-time applications because the garbage collection process interfered with the real-time execution and prevented successful adherence to deadlines.

B. Phase I Technical Objectives

The technical objectives of the Phase I effort will be to:

1) identify the requirements for middleware support of complex, real-time, fault-tolerant distributed systems in military and civilian applications,
2) evaluate and qualify the components slated for use in such as system,
3) create a technical design that addresses the identified requirements, and
4) create a development project plan for Phase II (option).

C. Phase I Work Plan

For phase I, we propose the following activities over a six month time span to create a software architecture:

C.1. Requirements Definition

C.1.1. Commercial: Research and analyze commercial real-time, fault-tolerant systems. Identify and characterize time constraints during normal operation and during fault recovery. Identify strategies for efficient fault-tolerant operation.

C.1.2. Military: Research and analyze military real-time, fault-tolerant systems. Identify and characterize time constraints during normal operation and during fault recovery. Identify strategies for efficient fault-tolerant operation.

C.1.3. Identify target operating systems and execution environments, including any additional software that is required for application execution. Verify that target operating environment can meet time constraints required by identified applications.

C.2. Evaluate Components

The application requirements must be refined into specific evaluation criteria for each potential component. Then each component must be evaluated in light of those criteria.

C.2.1. Evaluate Real-Time Java implementation

Expected evaluation criteria include:

- Is the throughput performance of the real-time Java implementation sufficient? Is hot-spot optimization required?
- Does the Java garbage collector interfere with meeting real-time deadlines?
- Can the real-time heaps defined in real-time Java be effectively used?
- Which, if any, of the various implementations of real-time Java should be selected?

C.2.2. Evaluate ACE

Expected evaluation criteria include:

- Should the use of ACE be replaced by JACE, which is a new version of ACE currently under development and targeted specifically at Java applications?

C.2.3. Evaluate CORDS/GIPC

Expected evaluation criteria include:

Can CORDS/GIPC meet the required performance and recovery time requirements for the applications?

C.3. Design Development

C.3.1. Define overall architecture

C.3.2. Identify necessary extensions to existing components needed for read-time and/or fault-tolerant application support.

C.3.3. Develop example application implementations for identified applications for selected target environment, including specific examples of data flow and fault recovery.

C.3.4. Review and characterize needs for instrumentation in system.

C.3.5. Production of design document and feasibility report.

C.4. OPTION: Phase II preparation

C.4.1. Prepare work plan for Phase II development

Prepare Phase II Transition/Marketing Plan