An Example Real-Time Command, Control and Battle Management Application for Alpha

David P. Maynard, Samuel E. Shipman, Raymond K. Clark,
J. Duane Northcutt, Russell B. Kegley,
Betsy A. Zimmerman,
Peter J. Keleher

TR #88121

Archons Project Technical Report

December 1988

An Example Real-Time Command, Control and Battle Management Application for Alpha

David P. Maynard[1], Samuel E. Shipman[2], Raymond K. Clark[2],
J. Duane Northcutt[2], Russell B. Kegley[3],
Betsy A. Zimmerman[3],
Peter J. Keleher[3]

Department of Computer Science[2]
Department of Electrical and Computer Engineering[1]
Carnegie Mellon University

Fort Worth Division[3]
General Dynamics

December 1988

Archons Project Technical Report #88121

This research was supported in part by the USAF Rome Air Development Center under contract number F33602-88-D-0027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

Table of Contents

List of Figures

Abstract

This report describes the design, implementation, and evaluation of a demonstration command, control and battle management (C^2/BM) system developed jointly by Archons project researchers at Carnegie Mellon University (CMU) and by technical staff members at General Dynamics Corporation (GD). Over the course of a few months, this team defined a realistic air defense scenario, designed a distributed, real-time C^2/BM system for this scenario, and developed a prototype implementation of the design using an early version of the Alpha operating system. In addition, a support environment was constructed to exercise the application. The primary goals of the application effort were to demonstrate the decentralized, real-time technology incorporated in the Alpha operating system and to evaluate that technology in a realistic context. To that end, the application exercises the distribution, real-time, and survivability mechanisms provided by Alpha.

By transferring Alpha technology from a research environment into an industrial setting, Archons researchers were able to obtain feedback from professional designers and system builders. This feedback validated the belief that the programming model and key support mechanisms provided by Alpha are well-suited to the design and construction of large, distributed, real-time control applications.

1 Introduction

This report describes the design, implementation, and evaluation of a real-time command, control, and battle management (C^2/BM) application developed to run on an early version of the Alpha operating system. This was an experimental effort aimed at investigating the efficacy of using Alpha technology as a basis for developing large, distributed, real-time systems. Over the course of a few months, a small team of people from the Archons project at Carnegie Mellon University (CMU) and from the Fort Worth Division of General Dynamics Corporation (GD) designed and developed both an air defense system prototype (ADSP) and a support environment in which to exercise it. Using this experimental prototype, we have demonstrated that Alpha's basic abstractions and resource management techniques can be used in the development of the complex real-time systems that will be required in the future.

Alpha is an adaptable, decentralized operating system for real-time applications. It is being developed as part of the Archons project's on-going research into distributed real-time systems. Alpha is a new kind of operating system that is unique in three significant ways. First, Alpha is decentralized, providing reliable resource management transparently across physically dispersed nodes. This allows distributed application programming to be done as though it were centralized. Second, Alpha provides comprehensive, high-technology support for real-time systems. In particular, it supports supervisory control applications (*e.g.*, industrial automation, combat platform management) which are characterized by predominately aperiodic activities that have critical time constraints (such as deadlines). Third, Alpha provides a unique combination of mechanisms for supporting the construction of highly survivable applications. Chapter 2 provides a brief introduction to Alpha. A book [Northcutt 87] and a series of technical reports (listed in the references) describe Alpha in more detail.

1.1 The Alpha Application Effort

Early in 1987, the Alpha research group at CMU began an effort aimed at evaluating and demonstrating the potential of Alpha technology in the context of a realistic, distributed, real-time application. Although Alpha implementation was still in its early stages (Pre-Release 0.2), we felt that building a realistic application would provide useful feedback so we could evaluate and, if necessary, redirect our research and development efforts.

We use the term "realistic," as opposed to "real," because experimenting with real applications of the kind that Alpha is designed to support would be difficult and expensive. Such applications are typically embedded—controlling vehicles, factories, or other complex machinery that interacts with the physical world. Real applications are also very complex. This complexity often impedes experimentation by making it difficult to isolate individual effects so that the results of various changes can be observed. To overcome these problems, we decided to build an experimental prototype containing the important components of a real application, but with few of the extra features that might limit the visibility of critical effects.

Our specific objective was to create an application that was reasonably representative of supervisory-level, real-time C^2/BM systems. It was important that the application have sufficiently complex and stringent requirements to benefit from Alpha's support for distributed processing, time-driven scheduling, and system survivability. Some of the specific requirements were:

- an application that could take advantage of the potential performance and survivability benefits offered by Alpha's transparent distributed processing,

- an application that could demonstrate system survivability and graceful degradation through automatic reconfiguration after node failures and through (re)integration of new or repaired nodes,
- an application that could demonstrate the scheduler's ability to ensure that: 1) when possible, application time constraints are satisfied, and 2) when overloaded, application-specified policies are used to control the allocation of the limited resources available,
- an application with dynamically varying time constraints that could exercise the flexibility and power of the best-effort scheduling techniques.

Finally, we required an application that could graphically illustrate these capabilities during a fairly short presentation that involved running the application software while explaining its behavior to an audience in real time. This final requirement allowed us to demonstrate our results to our research sponsors and other interested parties.

Since we needed a realistic application, we decided to team with an industrial partner. This partner would provide expertise in real-time command and control applications and share the effort of implementation. The Alpha group would gain insight into the requirements of the application and of the application programmers (in terms of development and debugging tools). In return, the industrial partner would benefit from the transfer of technology from the Archons project. Both groups would gain valuable experience in developing decentralized real-time systems.

1.2 Application Selection

Bearing in mind the objectives stated in Section 1.1, the Alpha group began the search for a suitable application and partner. We considered a wide range of application types—vehicle control (ship, airplane, spacecraft), process control (factory automation), command and control (air traffic control), and spoke with several organizations (*e.g.,* General Dynamics, Jet Propulsion Laboratory, IBM).

In March 1987, a group at General Dynamics approached the Archons project and indicated their interest in advanced technology for future C^2/BM systems. We told them of our interest in an application for Alpha, and together settled on the idea of building a prototype for a coastal air defense system inspired by the requirements of the Air Defense Initiative (ADI)—a U.S. Air Force defense program designed to protect against airborne attackers of the 1990's and beyond.

In the chosen scenario (described in detail in Chapter 3), the system coordinates the defense of the northeastern United States against airborne threats approaching over the Atlantic Ocean. The major functions of the system are to locate, identify, track and intercept threats before they inflict damage. The primary threats are manned bombers, manned cruise missile carriers, cruise missiles, and jamming devices. Available defensive assets include over-the-horizon (OTH) ground-based radars, airborne early warning aircraft, manned interceptors, medium-range anti-missile missiles, and point-defense surface to air missiles (SAMs). The system uses these assets to gather data from sensors, track the targets, assess their threat potential, launch weapons, and control weapon flight paths until the threats are destroyed. The system achieves its goal if no bombers or cruise missiles reach their targets.

1.3 Application Development

After the Archons project and General Dynamics agreed to cooperate on building an application for Alpha, a technology development and exchange effort was initiated. The effort was carried out from April to November 1987, with approximately six months of this time being spent on the actual system implementation.

General Dynamics brought to the effort their knowledge of real-time C^2/BM applications, their implementation experience with air defense applications, their experience with environment simulators and scenarios, their expertise in operator console and user interface design, and their skills in producing software to meet industrial quality standards. The Alpha group contributed their expertise in designing distributed real-time systems and the Archons project's advanced technology for decentralized real-time operating systems embodied in Alpha.

Working together, personnel from the Alpha group and from General Dynamics designed the scenario and specified the functions of the application system. This cooperation ensured that the application scenario and implementation were reasonably representative of real-world systems, and that the system would demonstrate some of the most important features of Alpha.

General Dynamics dedicated three full-time employees to the task of application design and implementation during the period of the joint effort. General Dynamics implemented the application system running under Alpha. They also designed and implemented the experimental support environment running under UNIX. The support environment (described in Chapter 4) includes the Scenario Simulator, the Experimental Control Console, the Operator Console, and the Communication Interface.

At CMU, four people worked full-time on the design and implementation of Alpha. CMU provided GD with pre-release software, culminating in Alpha 0.5. They provided advice on the proper use of the Alpha object-oriented programming model and other Alpha facilities, and assisted in implementing some portions of the application code. CMU personnel also assembled a duplicate of the Archons distributed systems testbed (See Section 2.5) at General Dynamics.

The Alpha group dedicated a fifth person to act as coordinator and liaison. The liaison spent approximately half-time working in Pittsburgh at CMU and half-time in Fort Worth at GD for the duration of the project (travelling between cities at two-week intervals). Work proceeded in a loosely coupled fashion, with the Alpha group at CMU concentrating on the operating system, and the application group at General Dynamics concentrating on the ADSP and its support environment.

The defense system prototype and its support environment was initially demonstrated to our research sponsors in December 1987. In March 1988 it was shown to our sponsors, DoD representatives, members of industrial technical staffs, and academic researchers. Both demonstrations were well received.

1.4 Caveats

It is worth repeating that the work described in this report was directed toward a demonstration that previews technology still under development. The application is realistic in that it is representative of the kinds of processing that take place in real-world systems. Although inspired by the ADI environment, the threat, doctrine, and tactics in the application are purely notional. Finally, the application software runs under pre-release Alpha 0.5. This early version of the operating system lacks several features that would further simplify the design and implementation of large distributed systems.

1.5 Report Outline

The remainder of this report describes the application in more detail and analyzes the results of the effort. Chapter 2 provides a system context by briefly describing the Alpha operating system and the Alpha distributed system testbed. Chapter 3 describes the application scenario and requirements in greater detail. Chapter 4 describes the experimental support environment in which the air defense system operates. Chapter 5 provides an introduction to the design of Alpha-based applications, and describes the high-level design of the ADSP. Chapter 6 describes each of the application object types, while Chapter 7 outlines the application threads. Chapter 8 explores the key technology issues of time-driven resource management and system survivability in the context of the ADSP. Finally, Chapter 9, evaluates our experience with the application and outlines ongoing Alpha research and development. An appendix at the end of the report provides a sample of the actual application code.

2 The Alpha Operating System

Alpha is an operating system designed to support complex, mission-oriented, real-time applications. These applications often have requirements that are very different from those usually considered by operating system designers. In particular, they require new types of support for distribution, real-time resource management, and system survivability and adaptability. Unlike other systems, Alpha has been crafted "from the ground up" to support the design and operation of such demanding applications in a reliable and maintainable manner.

Alpha is a *decentralized* operating system. It is oriented towards systems that may have anywhere from one to 100 physically dispersed processing nodes contributing to a single application or mission. By logically integrating these nodes into a single computer system, Alpha provides transparent access to distributed resources, and offers unified resource management (not just limited resource sharing) across node boundaries. By eliminating centralized facilities, Alpha allows applications to obtain the availability and survivability benefits of physical distribution without encountering the difficult problems associated with conventional distributed programming.

In addition to providing uniform resource management across node boundaries, Alpha supports *time-driven* management of vital resources such as processor cycles, communication access, and physical memory. Unlike static or synchronous scheduling techniques, Alpha's method of time-driven resource management supports supervisory-level real-time systems that may be dominated by dynamic and aperiodic activities with critical time constraints.

The prototype implementation (Release 1) of Alpha was created at CMU after more than ten years of experience with and research into the requirements of large, distributed, real-time systems. Further research and development continues at Concurrent Computer Corporation where Releases 2 and 3 of Alpha are being designed and implemented.

2.1 Alpha Programming Model

The Alpha programming model is based on the concepts of *objects* and *threads*. This model differs from process-based models by distinguishing data and executable code (objects) from the points of control that "animate" the code (threads). By making this distinction, Alpha is able to offer greater concurrency, better data encapsulation and more uniform resource management.

In Alpha, objects can contain both code and data. At the highest level of abstraction, objects correspond closely to abstract data types. Each object has a well-defined interface and is accessible only through the entry points or *operations* which it defines. Figure 1 shows a sample object with three operations (*Initialize, Insert,* and *Remove*) that define its external interface. Alpha enforces the object interface by placing every object in a separate, hardware-protected address space, and by controlling entry into objects with an *operation invocation* mechanism.

Unlike some object-oriented systems, objects in Alpha are passive. For computation to occur in an object, there must be one or more threads active in it. Alpha threads are the schedulable points of control that actually execute instructions. Intuitively, they are similar to "processes" in other systems. Each thread consists of an execution context that includes thread-specific data (*e.g.,* register values and stack information) and a set of attributes. The thread attributes

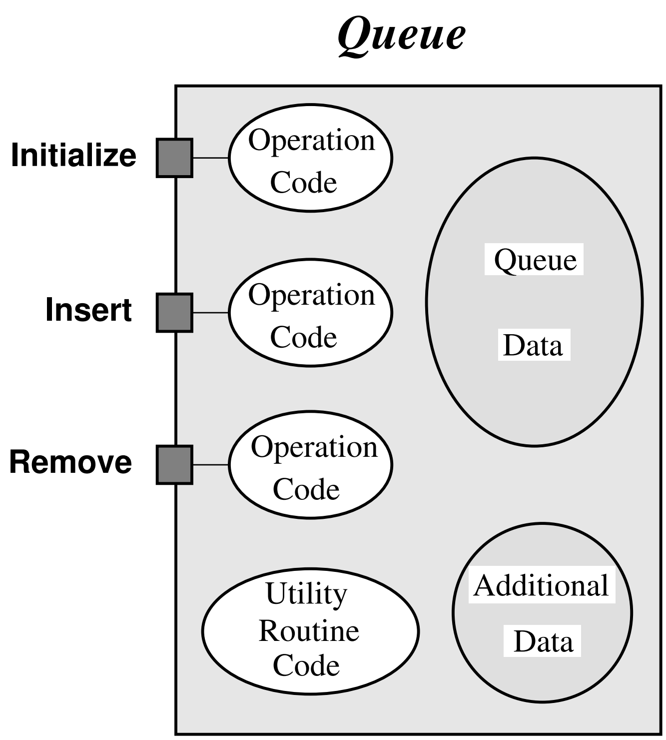


Figure 1: Example Object

include information, such as time constraints and reliability requirements, that is used by the system resource managers (*e.g.,* to ensure that deadlines are met).

When a thread is created, it begins executing at an application-selected operation in a specified object. Once running, it can travel to other objects by invoking operations on them. Alpha's threads are unique in that they are not constrained to remain on a single node. If an operation is invoked on an object at a different node, the thread transparently travels to the remote node, carrying its thread attributes with it. These attributes allow resources to be managed consistently throughout the entire decentralized computer.

Alpha does not artificially limit the level of concurrency within objects. At any given time, there may be zero, one, or several threads active in a particular object. The operating system provides concurrency control mechanisms such as locks and semaphores that the programmer can use to synchronize thread execution when required by the application.

Object-oriented programming offers many benefits including simplified design, improved modularity, and increased reusability of software components. Several books, such as [Cox 86], discuss the general merits of object-based systems. Alpha's object/thread model has specific advantages that make it well-suited for distributed, real-time systems. A technical report on the requirements and rationale for Alpha [Northcutt 88a] explains these advantages in greater detail.

2.2 Time-Driven Resource Management

One of Alpha's primary goals is to support highly complex and dynamic real-time applications. A major way that Alpha supports such applications is by resolving contention for system resources in a global, dynamic, and (most importantly) time-driven manner. Large real-time systems often consist of several concurrent activities, each of which may have critical timeliness requirements. In Alpha, these activities are embodied as threads. The thread abstraction not only serves as a means of capturing the notion of application activities in a system context, but provides a mechanism for notifying the operating system of any application-specific time constraints. Activities that have timeliness requirements specify those constraints dynamically as part of the attribute information that accompanies a thread.

2.2.1 Time-Value Functions

An essential prerequisite for being able to satisfy an application's timeliness requirements is providing a way for the system designer to express those requirements both concisely and accurately. In Alpha, time constraints are specified using the concept of *time-value functions*—functions that express the time-dependent value to the system of completing a specific phase of a computation [Jensen 75]. Time-value functions allow the designer to express requirements with much greater fidelity than the simple priorities or hard deadlines that other systems employ. Time-value functions are based on the idea that completing a computation has a certain value to the system. In real-time applications, this completion value often varies with time. For example, the classical concept of a hard deadline describes a task that has a constant positive completion value before its deadline, and zero value after the deadline (see Figure 2). Not all real-time activities have hard deadlines however. Other tasks may have critical times when their completion is *most* valuable, but may have non-zero completion value even after that time has passed. Such constraints are often known as "soft" time constraints since they do not have a single time, or hard deadline, when the completion value instantaneously drops to zero. It is also possible that some tasks are less valuable if completed too quickly (*e.g.,* when their input data improves in quality as time progresses).

Due to the dynamic nature of Alpha's intended applications, Alpha allows time constraints to be computed and specified at run-time. This allows applications to react to changes in the available resources or the environment. Several technical reports [Northcutt 88b] [Shipman 88] explain the features and usage of time-value functions more thoroughly.

2.2.2 Best-Effort Resource Management

Simply being able to express an application's timeliness requirements to the system would not be very useful if there were no way of utilizing the information to manage system resources.

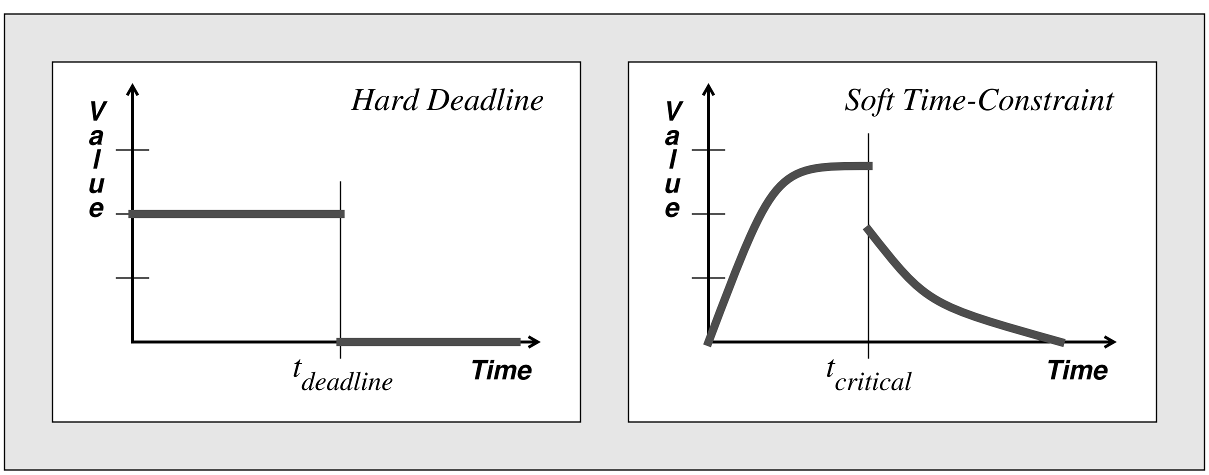


Figure 2: Example Time-Value Functions

Alpha supports a wide range of scheduling policies that may use all, part, or none of this information. One particular class of policies is being developed as part of the Alpha group's research into advanced real-time systems. This class of policies uses a technique known as *best-effort* resource management to satisfy the application time constraints whenever possible, and to facilitate graceful degradation (as defined by the application) when resource demands exceed the available supply. Best-effort policies evaluate the time-value functions of all contending threads collectively. If sufficient resources are available, the threads are scheduled in order of their critical times (the time after which the value of completing the activity diminishes). If the system is overloaded, threads are (temporarily) discarded from the schedule until enough resources are available for the activities that are scheduled to satisfy their time constraints.

Several policies may be used for deciding which threads to remove. The load reduction policy used for the ADSP attempts to maximize the total *value* accrued to the system during overloads. The value of completing time-critical tasks is specified by the application designer by means of time-value functions. When resource demands are too great, the best-effort policies ensure that time and effort are spent on activities that are potentially the most valuable. Equally as important, the policies ensure that resources are not wasted on activities that have little or no chance of making a positive contribution to the application.

Alpha's mechanisms for time-driven resource management are described more thoroughly in many of the technical reports listed in the references. More details on the best-effort algorithms are available in [Locke 86], [Clark 88] and others.

2.3 Support for Survivability

The mission-oriented systems for which Alpha is designed often need to continue operating even if parts of the computer system fail. Alpha provides many facilities that support the construction of systems that can survive and adapt to node failures and other sudden or gradual changes in the environment. These facilities are complementary to, but independent of, hardware mechanisms that enhance the reliability or fault tolerance of the computer hardware. Alpha does not impose policies on how failures should be handled. Instead, the operating system provides mechanisms that system designers can use separately or in concert to construct a system that includes the performance *vs.* survivability trade-offs that are appropriate for the application. The survivability tools that Alpha provides include methods for containing the effects of faults, mechanisms for handling exceptions in a uniform manner, facilities for ensuring the availability and consistency of vital data and services, and mechanisms for ensuring graceful degradation and the timely distribution of resources under overload conditions.

When hardware or software faults occur, it is important to contain their effects insofar as possible. Alpha's object-based model limits fault propagation by placing each object in a separate (hardware-enforced) address space, and by separating software components into system-enforced protection domains. These protection domains are defined using a capability mechanism that controls interactions between objects. Threads are only allowed to invoke operations on objects for which they have explicit capabilities (which cannot be forged). This protection of data and control over thread propagation helps the system designer prevent faults from spreading unchecked throughout the system.

For detecting failures, Alpha provides a uniform exception handling mechanism that supports a wide range of user- and system-defined events. Because of the distributed nature of Alpha, processing nodes may fail while threads span nodes. This means both that invocations can fail because an object no longer exists and that portions of a computation can become detached (or *orphaned*) because a thread spans a node that fails. Alpha provides mechanisms for indicating the failure of operation invocations and for detecting and eliminating orphan threads. Alpha also supports the notion of time-constraint exceptions that occur when a computation cannot meet its timeliness requirements (*e.g.,* because of a system overload). Since timeliness requirements are considered part of the correctness requirements for the system, such exceptions may be as vital as those caused by node failures. In Alpha, both types of exceptions are handled by the same mechanisms.

In order to respond effectively when a failure does occur, vital services must remain available and vital data must remain both available and (in some cases) consistent. The Alpha kernel has built-in functionality that supports object replication and atomic transaction mechanisms. By using *exclusive* replication (replication without data consistency between replicas), objects that provide vital services can be made available even if one or more of the replicas are destroyed. *Inclusive* object replication provides a method of ensuring that essential data will remain available, even after a node failure. By using the mechanisms that support atomic transactions, one can ensure the consistent behavior of actions, and can perform atomic, permanent, and serializable updates on data. The programming model technical report [Northcutt 88b] explains these Alpha robustness facilities more completely.

When failures occur in real-time systems, it is not sufficient to ensure that data and services remain available. It is also essential that the most vital tasks continue to execute within their time constraints. As described in the previous section, Alpha's time-driven resource management techniques are unique in their support of real-time systems where time constraints and available resources may change dynamically. In particular, the best-effort scheduling policies automatically adjust to overload conditions by ensuring that the most valuable threads continue to satisfy their timeliness requirements. This intelligent load shedding allows application and system performance to degrade gracefully in small increments. As with Alpha's other survivability mechanisms, the application designer controls (through time constraint specifications) how the system will respond to transient or permanent overloads.

Almost all of Alpha's mechanisms and services have been designed to facilitate the construction of survivable systems. The programming model, exception mechanism, and resource management techniques are carefully integrated so that techniques such as object replication and atomic transactions fit cleanly into the system. The result is an operating system that makes it easier for application builders to design systems that operate correctly under stress. As an added benefit, the same Alpha facilities that allow a system to adapt to failures also allow it to evolve as requirements change. This evolvability ensures that Alpha-based systems will not only be able to survive, but will be able to adapt to changing environments and scenarios—potentially increasing the useful lifetime of the systems while reducing the cost of system enhancements.

2.4 Pre-Release Alpha Version 0.5

The Alpha application effort was begun while the operating system was still at an early stage of development. When GD began implementing the application, the Alpha group at CMU was working on pre-release 0.2 of the operating system. The final version of the ADSP runs under pre-release 0.5 (and later) of Alpha.

Pre-release Alpha 0.5 supports most of the kernel-level functionality (*e.g.,* objects, threads, local and remote invocations, concurrency control, access control, and thread exceptions). In addition, an initial implementation of a best-effort algorithm is used for scheduling the application processor. However, the 0.5 system does not support many of the reliability mechanisms that would have simplified the implementation of the ADSP. In particular, the atomic transaction and inclusive replication facilities were not available.*

Pre-release 0.5 of Alpha also lacks many of the system-level services that are envisioned to be part of later implementations. Specifically, there are no user-level facilities for managing explicitly distributed services. Under most circumstances, distribution transparency is desirable; however, there are a few cases where the application prefers to manage distributed resources explicitly. It would be preferable to have the operating system provide services for managing these resources. Since Alpha 0.5 does not have a distributed systems services manager, an application-specific version was developed for the air defense system.

2.5 The Alpha Distributed Computer System Testbed

Alpha 0.5 runs on a special testbed that is designed to support distributed systems experiments. There are two copies of the Alpha prototype testbed—one at CMU, and one at GD. The sections below give a brief description of the testbed facility. A complete description of the design and construction of the testbed is given in *The Alpha Distributed Computer System Testbed* [Northcutt 88c].

The testbed facility at GD consists of two components—the *Distributed System Testbed* and the *Development and Control System* (See Figure 3). The Distributed System Testbed is the actual execution environment for the Alpha operating system and its applications. The Development and Control System is used for software development and for experimental monitoring and control.

* A specialized version of replication support was implemented for the ADSP; however, it was not available in time to be included as part of the public demonstration.

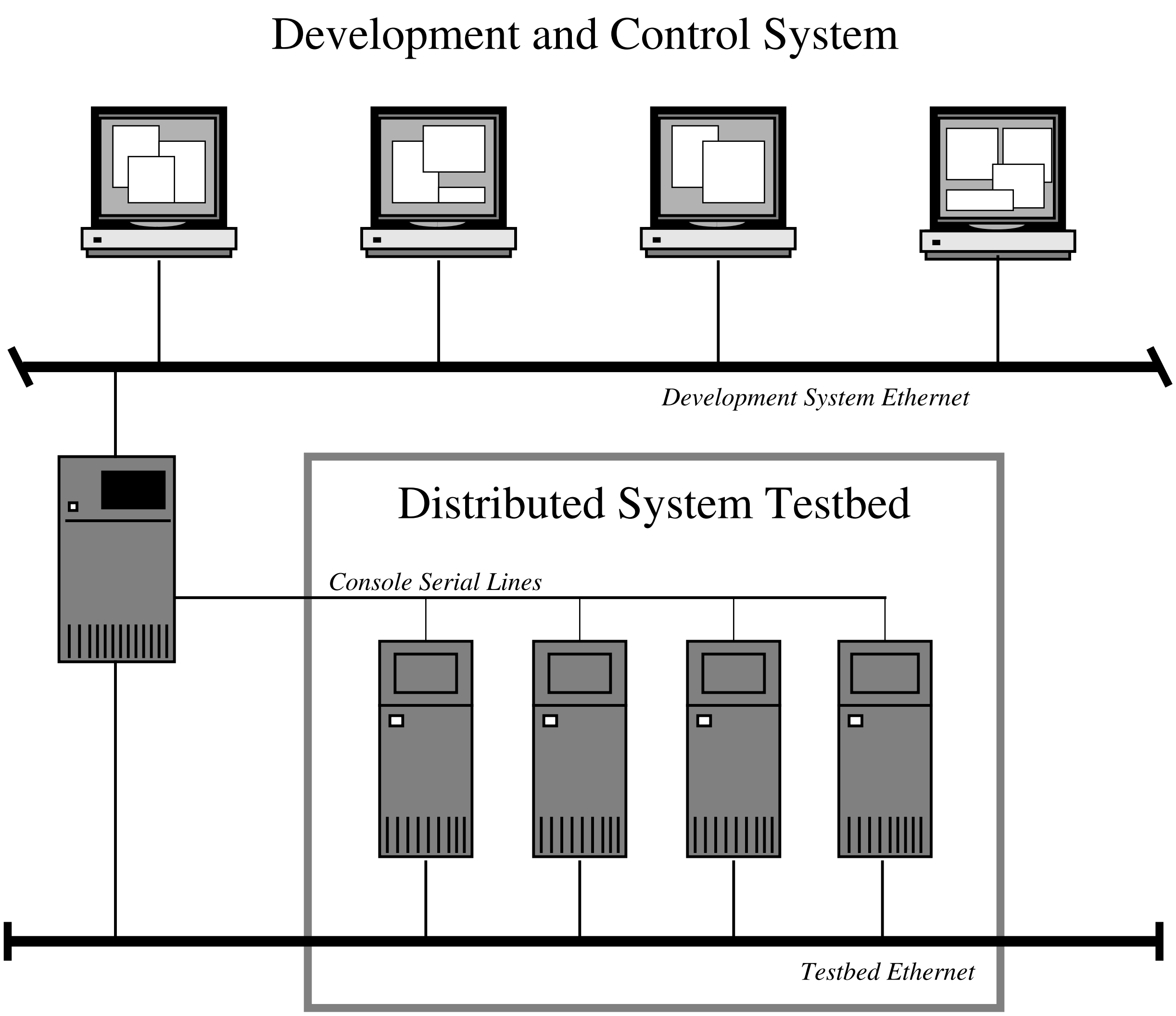


Figure 3: Alpha Distributed Computer System Testbed Structure

2.5.1 Distributed System Testbed

Alpha is designed to operate in a distributed environment containing one to 100 computing nodes connected by a local communications network. The distributed system testbed at GD contains four processing nodes connected by an Ethernet. An instance of the Alpha kernel executes on each of the nodes. The collection of nodes on which Alpha is running at any particular time constitutes the decentralized computer system on which applications execute.

The CMU implementation of Alpha has been designed to allow experimentation with hardware concurrency *within* the operating system. To support this type of experimentation, each node in the Alpha testbed contains multiple dedicated-function processors (see Figure 4). The Application Processor (AP) is a slightly-modified Sun Microsystems 2.0 CPU board that executes all application code as well as the Alpha kernel proper. The AP has 2 megabytes of local memory. The Scheduling Processor (SP) and Communications Processor (CP) are both (slightly-modified) Sun 1.5 CPU boards, each with 256 kilobytes of memory. The processors communicate over a Multibus I backplane. In addition to the processors, each node contains a

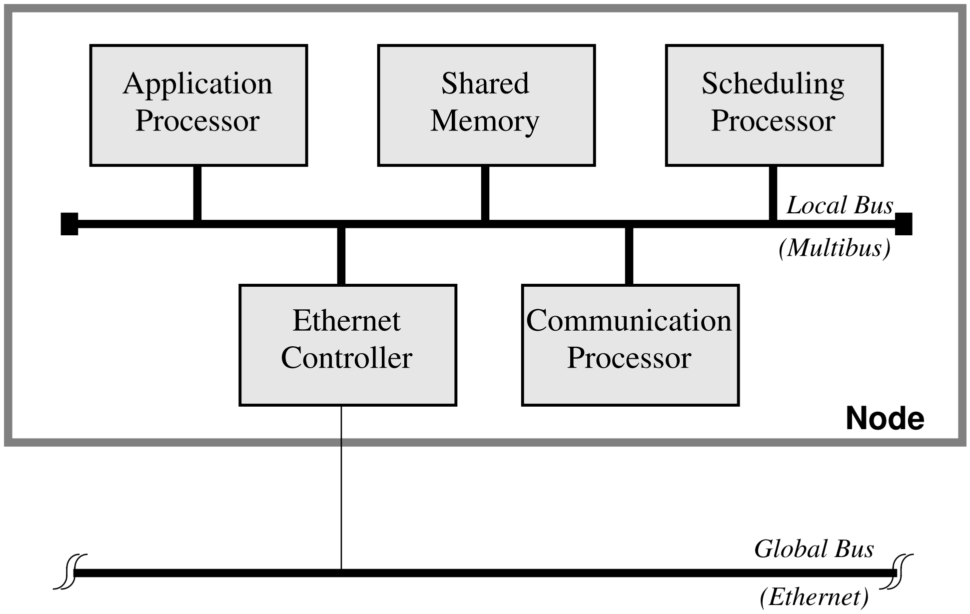


Figure 4: Alpha Testbed Node Architecture

Sun Ethernet controller and 256 kilobytes of shared Multibus memory. A technical report [Northcutt 88d] on the Alpha kernel describes how each of these components is used by Alpha.

2.5.2 Development and Control System

The Development and Control System consists of several Sun-3 and Sun-4 workstations running Sun's version of the UNIX operating system. The workstations are connected by an Ethernet and use NFS to share files between machines. The Alpha group at CMU has implemented several tools that allow software developers to write and compile Alpha and Alpha-based applications in the UNIX environment. For a complete description of the Alpha software development tools, refer to the *Programming Utilities* technical report [Northcutt 88e].

The Development and Control System is also used for experimental monitoring and control. One of the development workstations serves as a gateway to the Distributed System Testbed. The gateway machine has two Ethernet controllers—one connected to the development system network and the other connected to the testbed network. This shared network link serves two purposes. First, it allows program images to be downloaded to the testbed from the development machines. Second, it permits run-time monitoring of and communication with the application while Alpha is running.

The gateway machine also has a serial line multiplexer connected to the console lines of each of the nodes in the Distributed System Testbed. This connection allows developers to control and monitor the testbed nodes remotely from the development workstations. Software developed jointly by CMU and GD provides window-based remote access to the testbed console lines.

3 Application Requirements

In order to learn as much as possible from the application effort, we needed to start with a sufficiently realistic and demanding set of system requirements. The first meetings between the Alpha group and General Dynamics personnel were dedicated to choosing and tailoring the scenario. The goal was to develop an application that was representative of real-time command and control systems, and that was suitable for use as an operating system technology demonstration.

The partnership with GD was vital in designing a realistic application. The team from GD was experienced in the design and implementation of air defense systems, and was familiar with the requirements of contemporary and future C^2/BM applications. They had previously developed scenario simulations and user interfaces for similar applications. We relied mainly on their expertise to ensure that the chosen scenario was sufficiently realistic. When necessary, experts from other groups at GD were consulted to answer specific questions about system requirements.

The Alpha group wanted to highlight three key areas where the operating system provides unique support—decentralized computing, time-driven scheduling, and system survivability. Since proper handling of these areas often becomes more critical in larger systems, the "best" way to demonstrate the support would have been to construct a complete system including every conceivable activity. However, the complexity of large systems tends to obscure the relationship between a cause and its effects, making visual demonstrations difficult. Since the system had to be explained and demonstrated in a short, one-hour briefing, we needed easily-understood, highly-visible results. In addition, the scope of the effort was limited since only four months had been scheduled for the implementation of the entire system.

By the time the design meetings were completed (May 1987), we had chosen an air defense application and a set of requirements. We reduced the scope of the implementation effort by using simple mission planning algorithms and by partitioning functionality between the ADSP and the simulated environment.

The remainder of this chapter describes the scenario and the application requirements.

3.1 Scenario Description

The application scenario resembles a future coastal air defense environment. The primary purpose of the system is to coordinate the defense of the United States against airborne threats approaching over the Atlantic Ocean. Although inspired by the projected ADI environment, the details of the scenario are notional.

The application setting is the northeast United States air defense zone. Airborne, atmospheric threats approach the eastern seaboard at varying altitudes and velocities. The battle management system must use its assets to detect, track, and destroy the threats before they breach the integrity of the continental U.S.

The scenario assumes a wartime environment where approaching foreign craft are assumed hostile as soon as they are identified. The threats follow irregular flight paths and may change course at any time. The specific threats defined by the scenario are:

- manned bombers,
- manned cruise missile carriers,
- cruise missiles, and

- radar jammers.

Bombers and cruise missile carriers are large and relatively slow. They typically travel at altitudes where they can be detected at long range. In contrast, cruise missiles travel at high speeds and maneuver at low altitudes where they are difficult to detect with ground-based radar. The jamming devices constitute a secondary threat and have flight characteristics similar to the missiles.

The system identifies and tracks the hostile craft using sensor and reconnaissance facilities that include:

- over-the-horizon (OTH), ground-based radars,
- medium-range, coastal radar sites, and
- airborne early warning aircraft (*e.g.,* AWACS).

The battle management system has three types of weapons available to intercept approaching threats. These weapons are:

- manned interceptors,
- medium-range guided missiles, and
- point-defense, surface-to-air missiles (SAMs).

The manned interceptors are fighter aircraft armed with heat-seeking air-to-air missiles. Guided missiles are faster than manned interceptors and have automated terminal guidance. However, like manned fighters, they require external guidance until they are within range of their target. SAM's are available as a last-resort coastal defense. They have very limited range and must be aimed and activated by the ADSP.

The scenario does not define absolute performance characteristics for the threats, sensors, or weapon systems. In most cases, experiments were carried out using capabilities that were intended to represent a late-1990's environment. However, no attempt was made to guarantee the realism of the numbers used for altitudes, speeds, radar ranges, and so forth. Instead, emphasis was placed on the aspects of the scenario that define the types of activities the system must perform and the requirements it must fulfill. The following sections concentrate on these requirements.

3.2 Application Processing Requirements

The primary requirement of the air defense system is to keep hostile targets from entering the airspace of the continental U.S. The activities required to achieve this goal can be classified into three main areas: *tracking*, *mission planning*, and *weapon systems control*. At this high level, these functions are similar to those found in many C^2/BM applications. The following sections detail the application processing requirements in each of these areas. Figure 5 outlines their basic functionality.

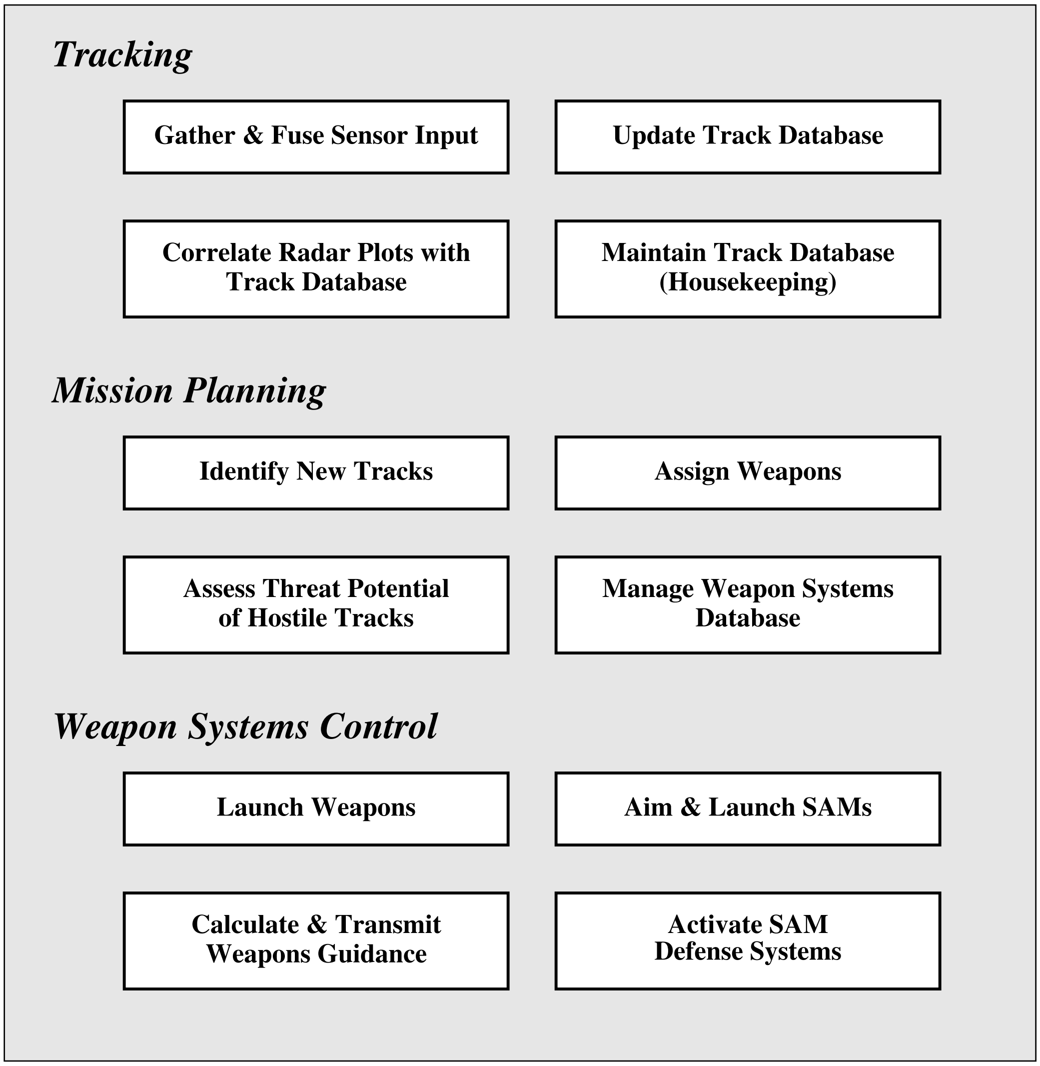


Figure 5: High-Level System Functions

3.2.1 Tracking

The first requirement of the air defense system is that it be able to provide information about the aircraft and other airborne platforms in the area of interest. The system needs to maintain a tracking database that contains position, velocity, and identity information for each target in the system. The database should also contain information about position histories to aid in track visualization and plot correlation.

To maintain the tracking database, the system must collect data from various sensor systems. This information must be fused into radar reports that contain the position, altitude, and identity (if known) of each object detected. These fused reports should then be transmitted to the other tracking functions for incorporation into the database.

When a sensor report is received, it must be correlated with previous information to determine whether or not the report is associated with a known flight path (or track). If so, the associated track information must be updated. If the report does not correlate with an existing track, the data should be entered in the database as an uncorrelated report. If subsequent radar plots match with the uncorrelated report, a new track entry should be created and the mission planning functions should be alerted of its existence.

There is also a certain amount of tracking database maintenance that must be performed to ensure that stale information does not cause errors in tracking or target identification. The system must periodically scan the tracking information, purging old tracks and uncorrelated reports that have not been updated within a certain time.

3.2.2 Mission Planning

When the tracking functions discover a new track, the mission planning components identify the track and determine whether or not it is a threat. Track identification may use such information as track origin (did it originate from a commercial U.S. airport?), velocity (is it too fast to be a bomber?), and identify-friend-or-foe (IFF) transponder information.

If the track represents a threat, its threat potential must be evaluated. This assessment is based on such considerations as threat type (*e.g.,* bomber or cruise missile), position (*e.g.,* distance from the coast), speed, and direction. Threats become more serious as they approach the coastline and interior of the U.S.

The system must determine the best choice of weapon system and locate the base nearest the target where the system is available. Defensive weapon assets include manned interceptors (*e.g.*, F-15, F-16), long-range guided missiles, and point-defense surface-to-air missiles. The weapons pairing decision is based on such considerations as the estimated intercept time for each type of weapon. Once chosen, the weapon must be allocated and passed to the weapon systems control functions for launching and in-flight control.

The mission planning component must also maintain situation assessment and resource availability information. In particular, the system must maintain a database containing, for each base, its location and its current inventory of weapons. It is also important to maintain a listing of all active target/weapon pairings.

In a real battle management system, the mission planning function would likely be carried out by a group of people using information provided by sensor data and intelligence sources. Since the ADSP is designed to be able to run without human intervention, each of the mission planning functions is carried out by the computer using simple heuristics. In a more advanced demonstration, these heuristics could be replaced by expert systems designed to operate in concert with a human advisor.

3.2.3 Weapon Systems Control

When a weapon and launch site have been chosen, the actual launch command must be issued along with initial guidance information. Once the interceptor is in the air, its progress must be monitored, and its course frequently updated to compensate for changes in the direction of the target and for navigational inaccuracies. If for some reason a weapon does not reach its designated target (*e.g.*, it is destroyed en route), the mission planning system must be notified so that replacement weapons can be allocated. In addition, for manned interceptors, the interceptors must be guided back to an airbase after their encounter with the target.

The weapons control functions are also responsible for the operation of the terminal defense system (coastal SAMs). The coastline must be periodically scanned for hostile craft that have penetrated the outer defense systems. When threatening tracks are discovered, a SAM site must be chosen and activated to intercept the target.

The application requirements make several simplifying assumptions with regard to weapon systems. In particular, weapons are presumed to have perfect kill ratios. That is, if a weapon is successfully launched and guided to within a set distance of its target, it is assumed to destroy the target. Although it would be trivial to model a kill ratio less than one, the behavior of the system would be more difficult to understand in a visual demonstration (since one could not tell if an intercept failed because of an Alpha policy decision or a random miss probability.)

3.3 Application Timeliness Requirements

The different activities that compose the application have a wide variety of timeliness requirements. The majority of the application tasks are aperiodic and have soft time constraints that vary with time and external inputs. Other activities are roughly periodic with soft time constraints that are fixed at compile time. Still others are aperiodic with hard deadlines. In addition to timeliness concerns, each of these tasks have different relative importances that may change dynamically. Three typical classes of timeliness requirements are represented by the plot correlation activity, the weapons guidance activities, and the SAM launch sequence. These requirements are outlined in the remainder of this section. The time-value functions corresponding to these requirements are discussed in Section 8.2.

3.3.1 Radar Plot Correlation

Radar plot reports arrive from the sensor system in *frames*. The data for each frame should be correlated with the tracking database before the next frame arrives. Since frames arrive at roughly periodic (six second) intervals, the plot correlation activity is approximately periodic with a six second deadline. However, the task is not truly periodic in the classical sense. In particular, the task is initiated by an external event (the arrival of a new frame) instead of being activated periodically. Also, plot correlation does not have a hard deadline. If the processing of a frame exceeds the six second time, it is still important to complete the processing of that frame. Only if the processing is not completed in two frame times should the attempt be aborted.

Because of the fundamental importance of tracking to the application, the plot correlation activity has one of the highest task importances in the system. The track database maintenance activity is also roughly periodic with a six second frame time. In contrast, however, the maintenance activity has a very low relative value. This example clearly demonstrates how a task's importance can be independent of its critical time.

3.3.2 Weapons Guidance

Weapons guidance is one of the most time-consuming activities in the system. Each guidance calculation requires a significant amount of processing time, and several interceptors may require guidance during a short interval. For the air defense system, weapon headings should be updated when approximately one fourth of the predicted time-to-intercept has elapsed. Since the time-to-intercept depends on the distance between the interceptor and its assigned threat, the guidance activity is fundamentally aperiodic and dynamic. It is impossible to know the critical times when the application is compiled.

The critical time for guidance updates indicates the time when course corrections are preferred. If excess resources are available, additional guidance updates may be issued (potentially decreasing the intercept time). Likewise, if the system is overloaded, guidance for interceptors that are still far from their targets can be delayed without a significant penalty. This flexibility is expressed by a variable "hardness" of the guidance timeliness requirements. When the time-to-intercept is large, the time constraint can be fairly soft since using slightly outdated tracking information will not significantly affect the intercept. However, as the intercept time nears, it becomes increasingly important to complete the course update prior to the critical time.

The importance of the guidance activities also varies. The value of completing an intercept is tied directly to the threat potential of the target. In the prototype system, the threat potential is based primarily on distance to the coastline and the type of threat. As with the plot correlation activity, the importance of a guidance task is independent of its critical time.

3.3.3 SAM Control

The weapon control program for the surface-to-air missiles is one of the few activities in the system with a strict hard deadline. The launch sequence models a system using an aim-aim-fire algorithm—where two successive sightings are used to aim the missile before firing. The aim-aim-fire sequence must be completed by its deadline or the launch is aborted and a new launch sequence initiated. In addition, a successful launch must be made before the threat leaves the range of the SAM battery.

Since the coastal SAMs are the final line of defense in the system, a failure in the launch sequence will mean that the battle management system has failed to meet is primary objective. Consequently, the SAM control tasks have the highest value to the system when they are active.

It is interesting to note that although the SAM control tasks are among the few activities with hard deadlines, they are strictly aperiodic. The SAM batteries are only activated when a hostile craft approaches within a certain distance of the coastline.

3.4 Application Survivability Requirements

One of the fundamental constraints of the air defense application is that the system must continue to fulfill its mission requirements even if one or more of the processing nodes fails. For the air defense application, fulfilling the mission means that coastal integrity is preserved even after nodes fail. In addition, the system must be capable of re-integrating nodes that have been repaired after a failure.

In order to respond when failures occur, the system must ensure that essential data remains available after a node failure. In particular, essential repositories such as the tracking database and the resource (weapons) database must survive failures so that their data will continue to be available. Not all data must be saved however. For example, it is acceptable for a single frame of radar reports to be lost when a node fails. Since radar reports will continue to arrive, the lost reports will quickly be replaced by fresh ones.

In addition to mission data, essential services must also continue to be available after failures. All vital tracking, mission planning, and weapon systems control functions must be obtainable after a node has failed. In many cases, it is not necessary to replicate the functionality so long as new service providers can be created dynamically to replace those that are lost.

Finally, it is important that the system not “forget” any important activities that were in progress when a node failed. In particular, weapons that were being guided by a lost node must have new guidance activities created on the nodes that remain available. Other system activities, such as database maintenance, also need to be restarted if they are prematurely terminated.

Ultimately, the real survivability requirements are that, as much as possible, the system continue to satisfy the processing and timeliness requirements outlined in the previous sections. In some cases, the combination of limited resources and additional exception processing after a failure might lead to system overloads. In such circumstances, the system should reconfigure itself or shed less vital activities until the situation is brought under control. Because the defense application operates in a highly dynamic environment, it would be impossible to construct a rigid system that could adapt to all of the failure scenarios. By designing a system that can respond flexibly, it is possible to continue satisfying mission requirements under a wide range of load and failure conditions.

4 Experimental Environment

The high-level architecture of the experimental system is shown in Figure 6. The system is divided into two major sections—the air defense system prototype (ADSP) and the experimental support environment. The ADSP runs under Alpha on the distributed computer system testbed. It implements most of the functions of a real air defense system including the tracking, mission planning, and weapon systems control functions outlined in the previous chapter. The remainder of the system supplies a simulated execution environment for the air defense system and provides monitoring and control functions for a human experimenter. These support functions run on Sun workstations under UNIX. The support environment is divided into several components—the Scenario Simulator, the Experimental Control Console, the Operator Console, and the Communication Interface.

As much as possible, the design team tried to preserve the division between the prototype system and its support environment. The goal was not to *simulate* an air defense system, but to build a (limited-function) air defense system prototype that operated in a simulated environment. Chapters 5, 6, 7, and 8 describe the design and operation of the ADSP. The remainder of this chapter describes the environment in which it operates.

4.1 Scenario Simulator

The Scenario Simulator is responsible for providing the environment in which the ADSP operates. It is based on a validated simulator developed by GD for an earlier project. The program simulates the movement of all hostile and friendly tracks and implements the radar coverage model that determines which tracks are visible to the defense system. It is also responsible for detecting “kills” that occur when an interceptor reaches its target. Every six seconds, the simulator computes new positions and velocities for the aircraft based on flight paths and guidance information. For those tracks that are visible, it transmits radar reports to the ADSP. Each of the reports contains the position, velocity, and any available identification information for the target. The actual position calculations performed by the simulator vary depending on the type of track.

Hostile flight paths are loaded into the simulator during system start-up. Flight paths are stored as a list of course changes. Each course change contains an activation time, an initial position, and a velocity vector. For each update, if the current simulation time equals the activation time for the next course change, the new position and velocity are set to the stored value. By storing both position and velocity for the course change, the simulator avoids cumulative round-off effects and ensures that hostile craft will follow their intended path. If a course change is not scheduled, the new position is computed based on the previous position and velocity.

For friendly interceptors, guided missiles, and radar aircraft, launch and guidance commands are received from the ADSP while the simulator is running. During each simulator update, new positions are computed for existing friendly tracks. The new positions are based on the previous position and velocity. After computing the new positions, any guidance updates are applied to adjust the speed and heading of the track. Finally, launch commands are processed by adding new interceptors, missiles, or AWACS to the simulation with the initial positions and velocities given in the launch commands.

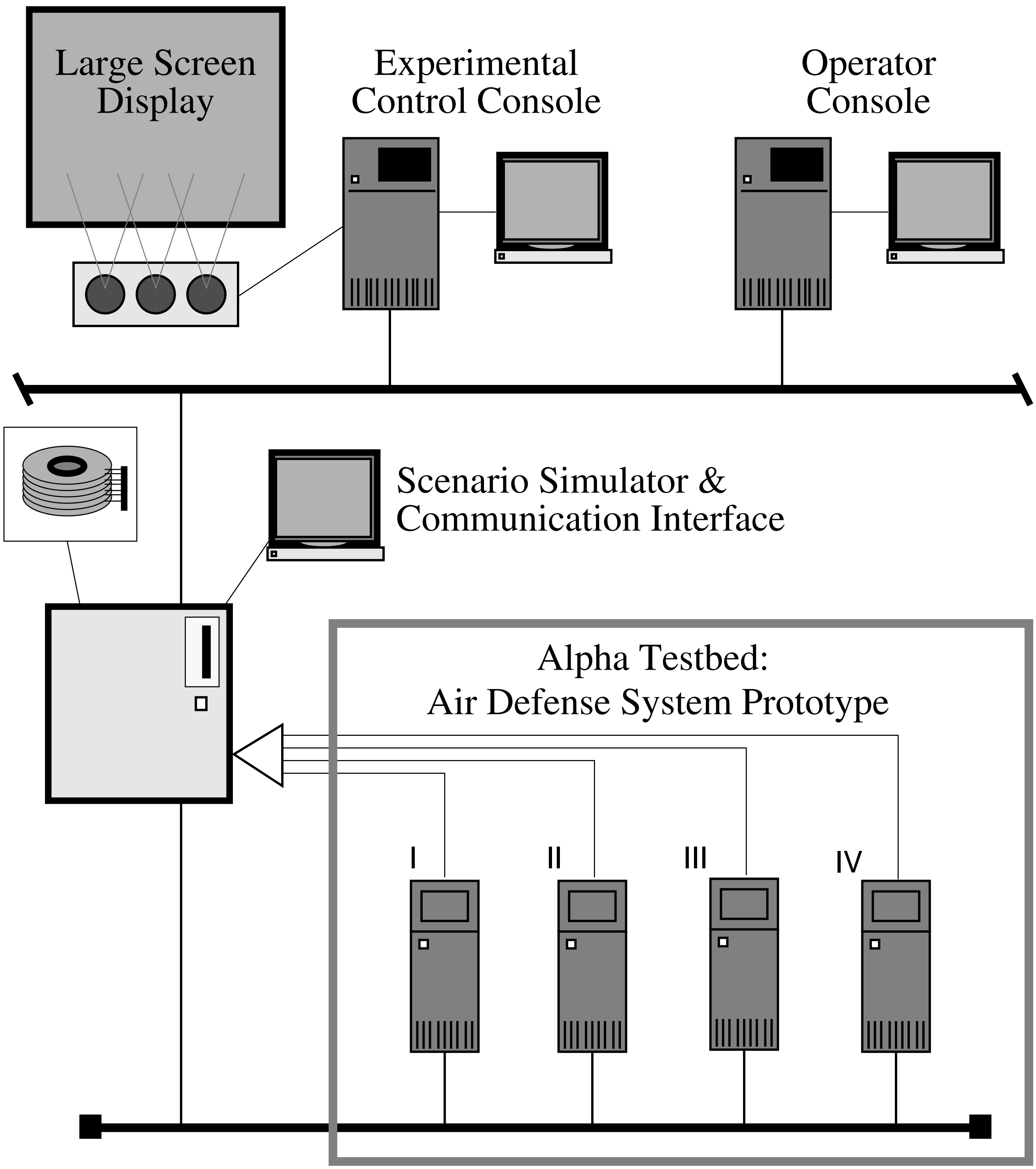


Figure 6: Overall System Architecture

After computing new positions and velocities for each track, the simulator applies a radar coverage model to determine which tracks are visible to the defense system. Each track position is checked to determine if it is visible to any of the radar sites (either ground-based or airborne). Each radar site has a cone-shaped coverage where the range of the radar increases with altitude. The simulator is implemented so that only one radar report is issued for any track in the system. This restriction means that no fusion processing is required in the ADSP. The design team chose not to implement sensor fusion for a variety of reasons. Chief among these reasons was the limited implementation time available.

In addition to course calculations, the simulator is responsible for detecting successful intercepts or "kills." When an interceptor or missile approaches within its kill range of a hostile craft, the enemy target is removed from the simulation. If a missile makes the kill it is also removed. The simulator does *not* issue any form of message when a kill happens. It is the responsibility of the ADSP to assess that a kill has occurred and initiate activities such as directing manned interceptors back to their airbases.

4.2 Operator Console

The Operator Console is analogous to the actual control console that a C^2/BM system operator would use. It provides an air situation display including a map, track information, resource locations, radar coverage, and other information useful in mission planning and control. Button panels allow the user to adjust the type and detail of information displayed. The console allows the operator to request detailed information about particular tracks or target/weapon pairings. In a real system, the operator's console would be highly interactive—requiring the user to input a variety of weapons assignment and control commands. For this application effort, the battle management system operates autonomously. The only commands (other than display control commands) issued by the human operator are those to launch airborne early warning aircraft.

The Operator Console receives display lists from the ADSP and converts the lists into an air situation display. This type of connection is similar to those used in many existing command and control systems. Ideally, however, we would have preferred to drive the Operator Console directly from one of the Alpha processing nodes. Unfortunately, the distributed system testbed does not support high-resolution color graphics, so it was necessary to implement the console on a Sun-3 workstation.

The display itself is based on the latest user interface technology and uses multiple display windows and virtual control panels that are activated by a "mouse" pointing device. The display is implemented under SunView(tm) using SunCORE(tm) graphics primitives. A sample screen snapshot is shown in Figure 7. The actual display uses color to enhance the contrast between different types of tracks and geographical features.

The Operator Console provides a variety of commands that control the type of information shown on the air situation display. Individual types of tracks can be turned "ON" or "OFF" to reduce display clutter. The following track types can be individually selected:

Uncorrelated Reports	*Hostile Missiles*	*Friendly Missiles*
Tentative Tracks	*Hostile Bombers*	*Friendly Interceptors*
Unknown Tracks	*Hostile Jammers*	*Friendly AWACS*

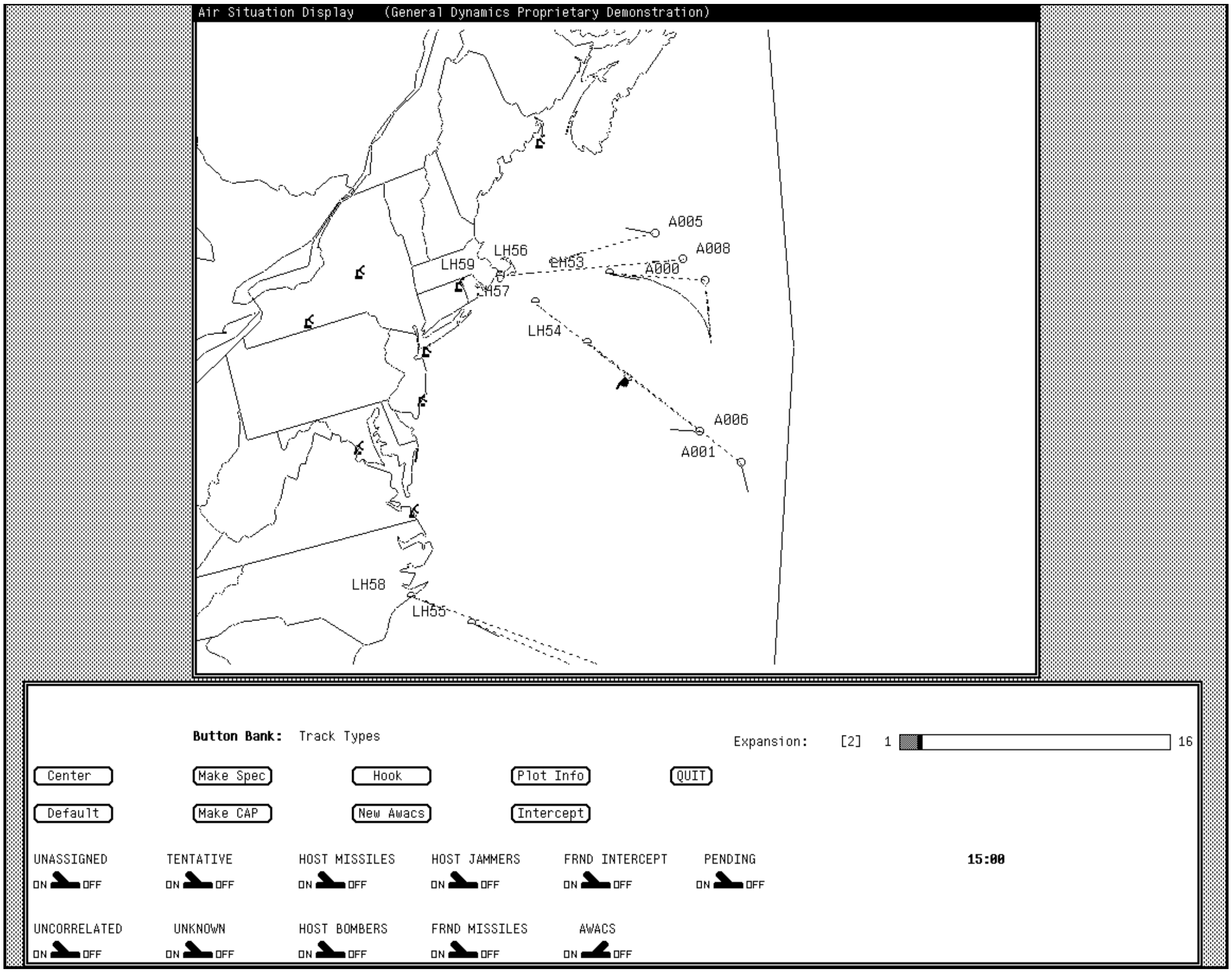


Figure 7: Operator Console Display

The console also allows the operator to control how much information is displayed about the different tracks. In particular, the user can choose whether or not to display the track ID, track block (position and velocity information), position histories, and pairing lines (lines that indicate which hostile target a friendly interceptor or missile is pursuing). The operator can also obtain additional information about particular tracks by selecting them with the mouse. The system will display detailed identification and tracking information as well as pairing information and predicted intercept paths for the selected track(s).

In addition to track information, the air situation display can present a variety of tactical data. Options allow the user to display military bases, major cities, SAM sites, and radar sites. For the radar sites, the system will display radar coverage rings for altitudes of 1000, 10000, and 25000 feet. The operator may place and selectively display reference points to aid visualization. Finally, the entire situation display can be expanded around arbitrary locations and displayed at magnification factors from 1X to 16X.

The only battle management function performed by the human operator is the launch of airborne early warning aircraft (AWACS). The operator uses the mouse to select the desired coverage area and issues a request to the battle management system. The ADSP then sends a launch command to the Scenario Simulator and guides the AWACS to the desired patrol area.

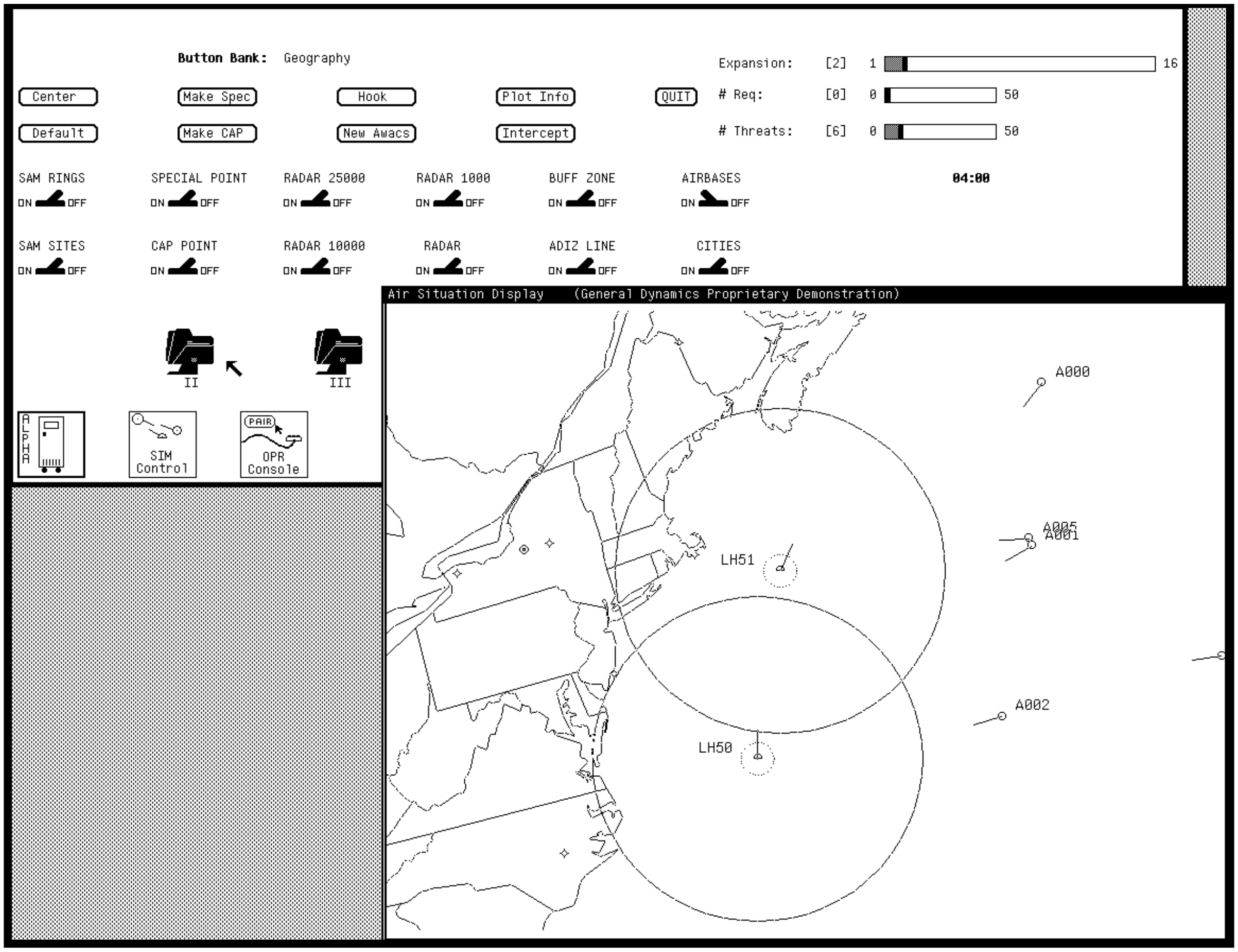


Figure 8: Experimental Control Console Display

Once in the desired region, the air defense system automatically guides the AWACS in an oval pattern. Since new AWACS planes are automatically added to the simulator model, the operator is able to control the scope of radar coverage available to the system.

4.3 Experimental Control Console

The main purpose of the Experimental Control Console is to allow the experimenter to monitor the behavior of the application and to control the simulated environment. For public demonstrations, it is connected to a large-screen display device so that a large audience can follow the scenario. The Experimental Control Console is similar in appearance to the Operator Console. Like the Operator Console, it provides an air situation display and resource information. However, the Experimental Control Console displays *reality* as defined by the Scenario Simulator, not as perceived by the defense system. Therefore, the tracks displayed are not subject to radar coverage limitations or time delays due to track processing.

The air situation display of the Experimental Control Console is almost identical to the Operator Console (see Figure 8). Unlike the Operator Console, the Experimental Control Console has an additional row of icons at the bottom of its button panel that monitor the health of the other components in the support system (e.g., Scenario Simulator, Operator Console, Commu-

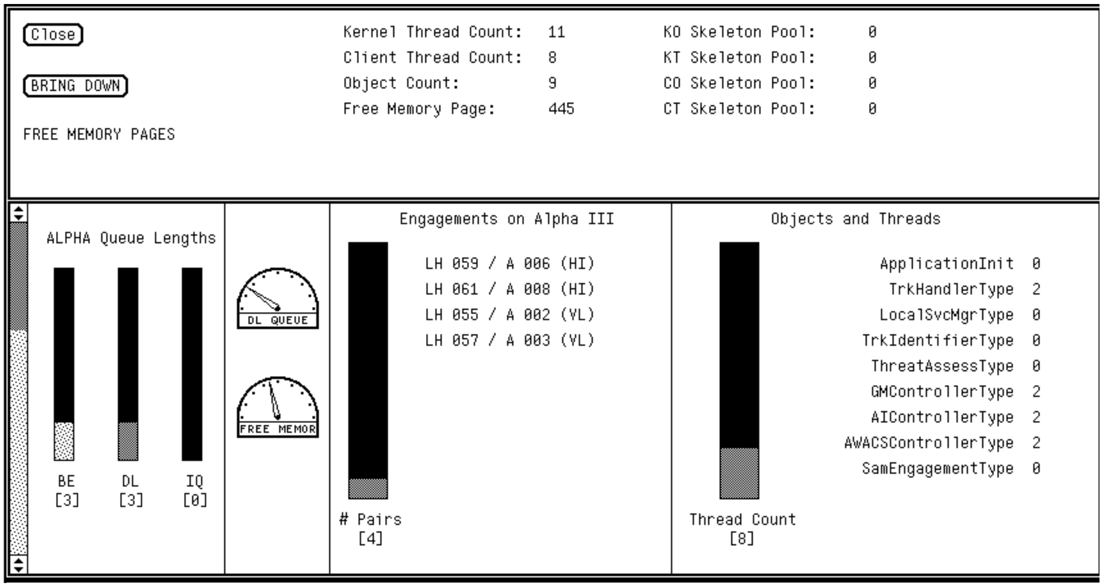


Figure 9: Node Information Display

nication Interface). There is also a row of icons indicating which of the Alpha nodes are currently running. When a node fails (is turned OFF during the demonstration) it is indicated on the Experimental Control Console.

The Experimental Control Console also allows the user to monitor the activity and performance of each node in the distributed system testbed. By selecting an Alpha node icon with the mouse, the experimenter activates a node information display. Node information displays (see Figure 9) provide resource information such as scheduler queue lengths and amount of available memory. They also list the weapon pairings being controlled by the node, the types of Alpha objects resident on the node, and the number of threads active in each object.

The Experimental Control Console is also used to control the Scenario Simulator. During system start-up it is used to script flight paths for hostile aircraft and missiles. A special display window allows the experimenter to enter flight paths graphically with the mouse. The scenario designer can specify hostile missile launch points and hostile warhead detonation points. Once the system is running, the console can be used to raise or lower the number of threats using control sliders. The sliders, located at the top-right of the button panel, display the current number of threats and allow the user to specify a desired maximum.

4.4 Communication Interface

The Communication Interface controls the flow of information between the ADSP and the different components of the support environment. Information such as radar reports from the simulator, display list information for the Operator Console, interceptor guidance information from the weapon controllers, and system monitoring information from the testbed is transmitted over this interface. The interface has two components—the message multiplexer and the Alpha/UNIX communications link. The message multiplexer distributes and routes messages between the Scenario Simulator, Operator Console, and Experimental Control Console. The communications link provides the actual data transfer path between the Alpha nodes and UNIX.

The message multiplexer provides a generalized data distribution service. The different components of the support environment communicate using tagged messages. Each message consists of a tag indicating its type (*e.g.,* radar report, guidance command) followed by the body of the message. The contents and length of a message are defined by a "C" language data structure for each message type. The multiplexer is connected to each of the components in the support environment via UNIX sockets. Messages arriving at the multiplexer are routed to one or more destinations based on a routing table that is loaded during system initialization. The message multiplexer also performs optional processing on each message when format conversions are needed between programs. The flexible routing scheme was especially valuable during the development and debugging stages since it allowed test stubs to be installed in place of components that were still under development.

The Alpha/UNIX communications link provides a reliable bidirectional connection between an Alpha object and a UNIX socket. On the Alpha side, a kernel object encapsulates the Alpha device drivers and provides operations to send and receive messages. On the UNIX side, a process handles low-level details and shuttles data to and from a socket that is connected to the message multiplexer.

The communications link operates either over an RS422 serial line or over the testbed Ethernet. Because Alpha supports an object interface, it is easy to change between the two links. The SerialIO and NetworkIO objects (that encapsulate the links) provide the same external interface, so it is only necessary to change the object reference that is distributed to other objects in the system. In the final version of the air defense system, the change between the RS422 and Ethernet links can be made by changing *one* line in the application code.

5 Air Defense System Design

The air defense system was the first significant application designed for and implemented using the Alpha programming model. Although there was no established methodology for developing Alpha-based applications, the design team found conventional software engineering principles to be well-suited for use with the object/thread paradigm. Many of the fundamental concepts of software engineering (*e.g.,* abstraction, information hiding, and modularity [Fairley 85]) are directly supported by the object model. In fact, GD found that Alpha's programming model simplifies the software design process by allowing designers to move directly from a functional decomposition of the problem to an object-based implementation. Since no conversion step was required to translate the natural system structure into "foreign" notions such as client/server processes or rate groups, the requirements specification led directly to an implementation that was easy to understand and simple to modify as requirements changed.

Starting from the system requirements outlined in Chapter 3, team members from GD and CMU used a top-down approach to develop a design for the air defense system. This design was subsequently enhanced and implemented by the group working at GD.

5.1 Design Approach

The application effort provided the participants with significant experience in the design and development of distributed, real-time systems. By applying some fundamental software engineering principles to the design of the ADSP, the designers developed an informal approach to creating applications for the Alpha programming model. Although the technique is not sufficiently refined to classify it as a methodology, it does offer guidelines and insights that can be used in developing future systems.

Using the system requirements as a basis, team members from GD and CMU used a top-down approach to design the air defense system. The design process can be broken down into the following components:

- Identify major system functions and activities
- Specify objects that provide the necessary functions
- Define threads that carry out the required activities
- Assign time constraints to threads
- Distribute the application across nodes

The first step in designing the air defense application was examining the requirements to identify the major functions and activities. The designers found it useful to represent the major components as (potentially ill-defined) objects and threads. Attempting to specify functionality and operations for these "objects" often revealed natural boundaries where components should be further specified or divided.

Once the major system functions had been defined, standard software engineering principles were used to modularize the system into well-defined objects and operations. The specification of the objects was a major part of the design effort. It was also the part most amenable to conventional design methods. Designers used techniques such as data abstraction and functional decomposition to identify object boundaries and specify appropriate operations. Well-defined metrics such as *coupling* and *cohesion* were used to aid with the division and to identify parts of the system where more design work was needed.

While the system functions were being decomposed into object specifications, work continued on defining the application threads. Threads correspond to activities performed by the system. Describing activities as threads provides a way of specifying how compound functions are accomplished using many objects. Threads travel transparently between objects and nodes and may execute concurrently. The designers considered several alternatives when deciding which activities should have separate threads. The general guideline adopted was that threads should be defined for activities that were logically independent and/or that had independent timeliness requirements. Chapter 7 describes the application threads that were chosen.

Once the threads had been defined, rules were developed for determining their time constraints. In some instances, the timeliness requirements were static and easy to define. In others, the time constraints were both dynamic and data-dependent. In the more complex cases, the general shape of the time-value function was chosen based on the timeliness characteristics of the task. Specific rules were then developed for calculating the parameters that define specific time-value functions (such as critical time and maximum value) from run-time information. Section 8.2 describes the time-value functions that were eventually derived.

To a large extent, the object type specifications were developed without considering how the system would be distributed across multiple nodes. Since Alpha provides transparent distribution, the objects could be defined as if the application were going to execute on a centralized system. Distribution issues became important when considering the survivability, availability, and concurrency requirements of the application. Survivability requirements determined which objects would be replicated, while availability considerations influenced the number and placement of object instances. The desire to improve performance through application concurrency influenced how threads were distributed among the nodes. Section 8.1 explains how these requirements were used to develop a plan for distributing the objects and threads that comprise the air defense system.

5.2 Functional Decomposition

The application requirements outlined in Chapter 3 divide the ADSP into three major components—tracking, mission planning, and weapon systems control. The first step in designing the air defense system was analyzing these areas to identify the major functions and activities. The purpose of identifying these components was not to develop an exhaustive or precise list. Rather, the goal was to provide a basis from which further design work could proceed. The following paragraphs briefly outline the results of the analysis.

Tracking revolves around the maintenance of a database containing positions, velocities, and other information about airborne objects. The primary tracking activity involves:

- collecting radar reports from the Communication Interface,
- correlating the radar reports with the tracking database,
- updating the tracking database to reflect the new information, and
- transmitting display information to the Operator Console.

The tracking component is also responsible for identifying new tracks that enter the coverage zone and for purging old ones that have either left the zone or been destroyed. Additional tracking functions allow other parts of the system to retrieve and update information about particular tracks.

The mission planning component determines and implements the appropriate response to new tracks. The major functions include:

- determining the identity of new tracks,
- assessing the threat potential of hostile tracks,
- assigning appropriate weapons to intercept hostile craft, and
- initiating weapons launch and guidance sequences.

Every new track that enters the system must be identified. The remaining mission planning tasks are performed only for hostile tracks. Identification and threat assessment rely on information from the tracking database. To support the weapon assignment function, the mission planning system maintains a separate database containing information on available weapon systems and active target/weapon pairings.

Weapon systems control involves:

- launching weapons and
- guiding weapons to their designated targets.

Specific versions of the control functions are required for each type of weapon. The long-range weapons (*i.e.,* airborne interceptors and guided missiles) require frequent course corrections to ensure that they successfully intercept their targets. AWACS aircraft also require course updates to maintain their oval flight pattern. The short-range SAMs only require initial guidance. However, a separate weapon control activity is required for detecting when targets approach within range of the SAM batteries.

The tracking, mission planning, and weapon systems control functions exist primarily to satisfy the application processing requirements. Additional functions are needed to fulfill the system survivability requirements. Specific functions dedicated to distribution and survivability are responsible for tasks such as:

- distributing the processing load among multiple nodes, and
- helping the system recover from node failures.

Finally, because the ADSP is an experimental prototype, other functions are required to integrate the system with its operating environment. These support functions include:

- handling communications with the experimental environment, and
- supplying system monitoring information to the Experimental Control Console.

Although this list of activities and functions identified by the requirements analysis was not exhaustive, it provided the basis for defining the objects and threads that would compose the air defense system prototype. The following two chapters describe the actual objects and threads that were chosen.

6 Air Defense System Objects

The functional decomposition given in Section 5.2 provided the basis for choosing the object types. After the major system functions had been identified, related functions were grouped

Tracking Objects

- Plot Correlator
- Track Database

Mission Planning Objects

- Track Handler
- Track Identifier
- Threat Assessment
- Weapons Manager

Weapon Systems Control Objects

- Interceptor Controller
- Guided Missile Controller
- AWACS Controller
- SAM Controller
- SAM Engagement

System Support Objects

- Communication Interface
- Serial I/O
- Network I/O
- Node Status
- Distributed Services Manager
- Local Services Manager
- Recovery Manager

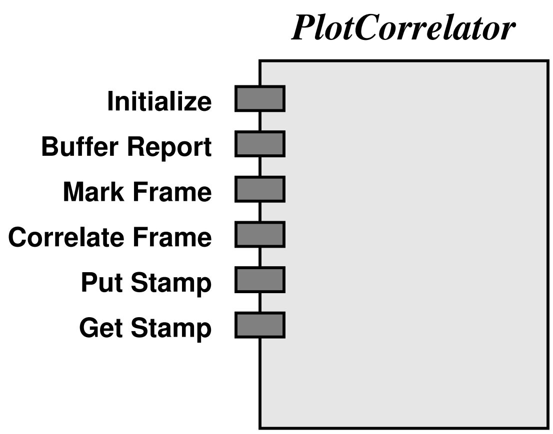
Figure 10: Air Defense System Object Types

into objects using standard software engineering techniques. Figure 10 lists the object types that were chosen. The remainder of this chapter describes each of the objects in greater detail.

In the remainder of the text, object and operation names are set in italic type (*e.g., PlotCorrelator*). At times, spaces have been inserted in object or operation names to improve legibility. Operations within a specific object are referenced using a "dot" notation (*e.g., Object.Operation, PlotCorrelator.Initialize*).

6.1 Tracking Objects

6.1.1 Plot Correlator

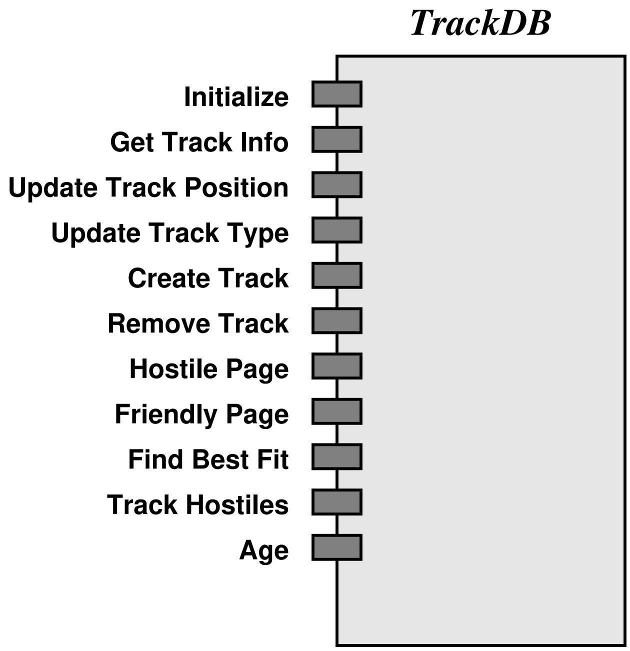


The *PlotCorrelator* object correlates incoming radar reports with existing tracks in the tracking database (*TrackDB*). Incoming radar reports are buffered in the *PlotCorrelator* until an end-of-frame marker is received. The entire frame of data is then correlated with the database contents. Reports that do not correspond to existing tracks are entered into the database as uncorrelated reports. When three consecutive reports are received for a new track, its type is changed to "unknown" and a new thread is created to handle its identification and processing.

The operations defined on the *PlotCorrelator* are:

- *Initialize* — initializes internal data structures and creates a thread in the *CorrelateFrame* operation.
- *Buffer Report* — buffers a radar report in the local queue.
- *Mark Frame* — marks the occurrence of the end of a radar frame.
- *Correlate Frame* — correlates an entire frame of radar reports. The operation blocks until a full frame has been received. It then correlates the reports (see *TrackDB.FindBestFit*), creating new tracks for plots that do not match, and creating new threads to handle unknown tracks as they enter the system. When processing for the frame has finished, the operation loops back and waits until the next frame of data has been received.
- *Put Stamp* — stores a timestamp in the timestamp buffer.
- *Get Stamp* — retrieves the next timestamp from the timestamp buffer.

6.1.2 Track Database



The track database *(TrackDB)* object encapsulates the tracking database and all of the routines that manipulate the database. It maintains position, velocity, and identity information for all known tracks. Tracks can be created, removed, and updated (with new position or type information). The database also defines special operations used to correlate radar reports with existing database entries and to scan the coastline for hostile aircraft.

The operations defined on the *TrackDB* are:

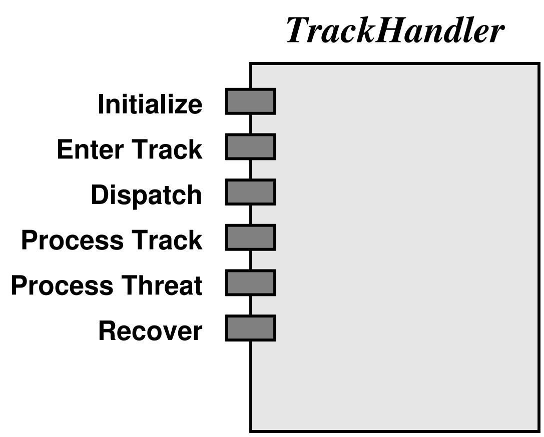
- *Initialize* — initializes the database and starts a “housekeeping” thread in the *Age* operation.
- *Get Track Info* — retrieves information about a specific track (based on its *track_id*).
- *Update Track Position* — sets a track’s position and velocity to the given values. This operation is used by the plot correlation function to update the database when new radar data is received.
- *Update Track Type* — sets the “track type” entry for the specified track. This operation is used by the new track processing functions to indicate whether a track is a hostile bomber, friendly interceptor, friendly missile, etc.
- *Create Track* — creates a new track entry in the database.
- *Remove Track* — deletes a track entry from the database.
- *Hostile Page* — returns a block (page) of several track entries. Multiple invocations can be made to retrieve all of the hostile entries in the database. This operation is used to update the Operator Console display.
- Friendly Page — is equivalent to HostilePage except it returns entries for friendly tracks.
- Find Best Fit — returns the track_id of the track that correlates most closely with a

given position. A special indication is returned if there are no matching tracks. Correlation is based on available IFF information and predicted aircraft positions. This operation is used by the plot correlation thread to match radar reports with existing tracks.

- Track Hostiles — is a specialized operation used by the SAMController object to locate hostile tracks that will soon come within range of a SAM defense site.
- Age — scans the database for tracks that have not been updated recently. Track entries that are too old (because the aircraft was destroyed or moved out of range) are deleted. A thread is started in this object by Initialize.

6.2 Mission Planning Objects

6.2.1 Track Handler



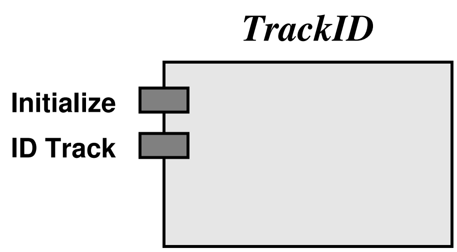
The *TrackHandler* object coordinates the response to new tracks that enter the system. The *ProcessTrack* operation is invoked for every new track. It is primarily responsible for establishing the identity of unknown tracks. When hostile tracks are discovered, *ProcessThreat* uses the other mission planning objects to assess threat potential, choose weapons, allocate weapons, and initiate weapons launch. *ProcessThreat* encapsulates the tactical decision functions that determine how the system responds to threats.

The operations defined on the *TrackHandler* are:

- *Initialize* — initializes internal data structures and creates a thread in the *Dispatch* operation.
- *Enter Track* — registers the existence of a new track in an internal queue. This operation is invoked by *PlotCorrelator.CorrelateFrame* when a new track is discovered.
- *Dispatch* — creates threads to handle the processing of new tracks. A *Dispatch* thread loops in this operation, removing entries from the new track queue (see *Enter Track*) and creating threads in the *ProcessTrack* operation.
- *Process Track* — handles the processing of a new track that has entered the system. The track is identified using the *TrackID* object and, if hostile, is handed off to the *ProcessThreat* operation. This operation registers each new track with the *WeaponsManager* object so that the activity can be recovered if a node fails.

- *Process Threat* — determines and implements the proper response to a new threat. The *ThreatAssess* and *WeaponsManager* objects are used to determine the threat potential and to choose and allocate an appropriate weapon. Once a weapon has been chosen, a new thread is created to launch and guide the weapon to its target.
- *Recover* — restarts activities that were lost when a node failed. These activities include new track processing and weapons guidance. Section 8.1 describes node failure recovery in greater detail.

6.2.2 Track Identifier

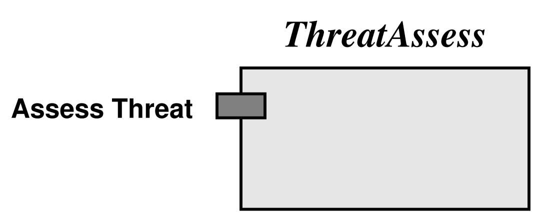


The track identifier (*TrackID*) object determines the identity of a track based on origin, velocity, and IFF information. The initial version of this object relies on very simple heuristics to determine the identity—friendly aircraft are assumed to transmit reliable IFF information, while other aircraft are assumed to be hostile. Velocity and altitude are used to discriminate between bombers, missiles, and interceptors.

The operations defined on *TrackID* are:

- *Initialize* — initializes internal state.
- *ID Track* — identifies a track based on altitude, velocity, and IFF information.

6.2.3 Threat Assessment

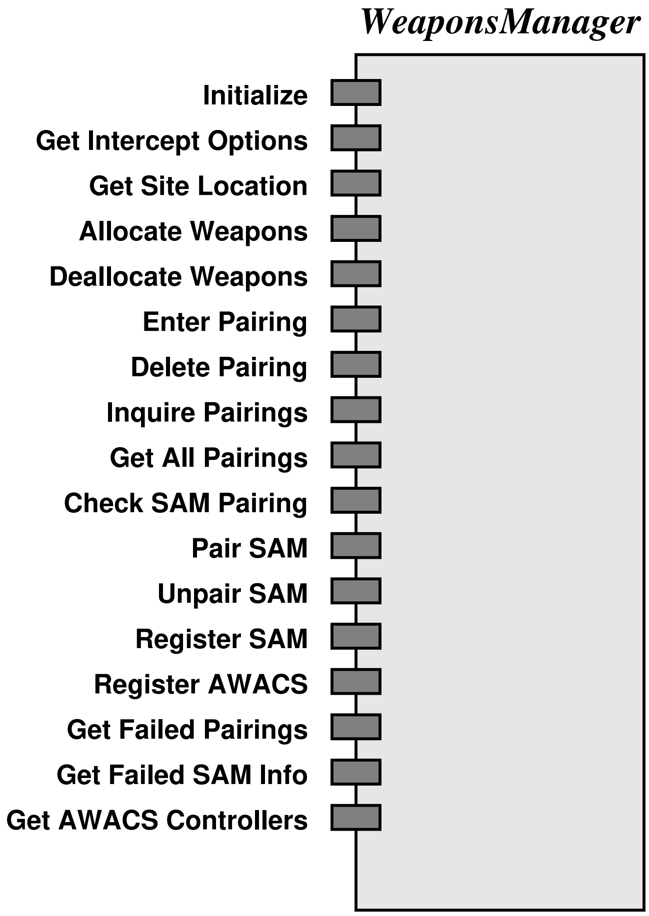


The threat assessment (*ThreatAssess*) object calculates the threat potential of a track based on its position, velocity, and type. Tracks become more threatening as their estimated travel time to the coast decreases. This object is used by the *TrackHandler* and weapon control objects to assign importance values to time constraints.

The only operation define on *ThreatAssess* is:

- *Assess Threat* — assesses the threat potential of a hostile track.

6.2.4 Weapons Manager



The *WeaponsManager* object maintains a database of interceptors and missiles that are available at each airbase. The *TrackHandler* consults this object to identify the type and location of weapons that can intercept a particular threat. Once the *TrackHandler* has chosen a weapon, it invokes the *WeaponsManager* to allocate it. Weapons that have been allocated are removed from the pool of available resources. Guided missiles are removed permanently, while manned interceptors are returned to the pool after they have completed their mission and returned to base.

The *WeaponsManager* also maintains a list of active target/weapon pairings. This pairing information is transmitted to the Operator Console where it is (optionally) used to display pairing lines and predicted intercept information. The pairing information is also used after a node failure to restart weapon guidance threads that were executing on the failed node.

The operations defined on the *WeaponsManager* are:

- *Initialize* — initializes internal data structures, including the weapons database.
- *Get Intercept Options* — computes the predicted time required to intercept a given threat for each available weapon type. The options are returned in order of increasing intercept times.
- *Get Site Location* — returns the coordinates of an airbase based on its site ID.

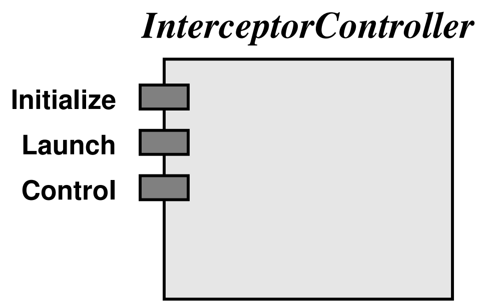
- *Allocate Weapons* — allocates a particular weapon (airborne interceptor or guided missile).
- *Deallocate Weapons* — deallocates a particular weapon (*e.g.,* after an interceptor has returned to base after a mission).
- *Enter Pairing* — registers a pairing that associates a hostile track with a particular weapon (either a guided missile or manned interceptor). The pairing information specifies the node where the weapon guidance thread is running.
- *Delete Pairing* — removes a target/weapon pairing from the database.
- *Inquire Pairings* — checks to see if one or more weapons are already paired with a particular hostile track.
- *Get All Pairings* — returns a list of all target/weapon pairings
- *Check SAM Pairing* — checks to see if a SAM site has been paired against a particular hostile track.
- *Pair SAM* — registers the assignment of a particular SAM site to a hostile track.
- *Unpair SAM* — removes a target/SAM pairing.
- *Register SAM* — saves the name of the node where the *SAMController* is currently active. This information is used to create a new *SAMController* if the previously active one is lost because of a node failure.
- *Register AWACS* — registers an AWACS guidance activity (including node placement). This information is used in the event of a node failure to restart AWACS control threads that were active on the lost node.
- *Get Failed Pairings* — returns a list of pairings registered on a particular node (which is presumed to have failed). The pairings are also removed from the pairing database.
- *Get Failed SAM Info* — checks to see if the *SAMController* object was active on a lost node.
- *Get AWACS Controllers* — returns a list of AWACS guidance threads that were active on a failed node.

6.3 Weapon Systems Control Objects

The weapon systems control objects are responsible for launching individual weapons and for providing necessary in-flight guidance. The primary weapon systems are manned interceptors and guided missiles. The secondary systems are terminal defense SAMs and airborne early warning aircraft (AWACS).

Interceptors and missiles require frequent course corrections to intercept their targets successfully. These guidance updates are required in part because threats may change course and in part because the weapon controllers use *pursuit guidance* algorithms. Pursuit guidance uses the current position of a target to compute a weapon's course. An alternate guidance algorithm, *lead guidance*, predicts the future position of a target by extrapolating its current flight path. Lead guidance would require fewer course corrections and would likely result in quicker intercepts. However, pursuit guidance has a particular advantage for the ADSP. One of the goals of the application effort was to demonstrate the effects of proper time-driven scheduling. If lead guidance were used, weapons could hit their target with only a single guidance command (assuming the target did not change course or speed). Using pursuit guidance ensures that there will be a visible effect when guidance updates are delayed. This allows observers to determine when load shedding occurs and to see which threats have been bypassed.

6.3.1 Interceptor Controller

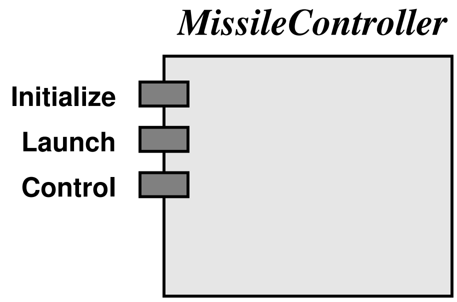


The *InterceptorController* object is responsible for launching manned interceptors and guiding them to their targets. The *Launch* operation transmits a launch request to the Scenario Simulator and registers the target/weapon pairing with the *WeaponsManager*. Once an interceptor has been launched, control is passed to the *Control* operation which calculates further course updates. The *Control* operation also determines when the next course update should be made (based on the estimated intercept time) and establishes the proper time constraints. When the threat has been intercepted, the *Control* operation guides the interceptor back to base where it lands and is returned to the pool of available weapons.

The operations defined on the *InterceptorController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of a particular interceptor. Launching consists of computing an initial trajectory, creating a track for the interceptor in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched weapon is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers a weapon to its target by issuing intermittent guidance updates. The frequency of the guidance updates is determined by the distance between the interceptor and its target. Guidance updates are computed using a pursuit guidance algorithm.

6.3.2 Missile Controller

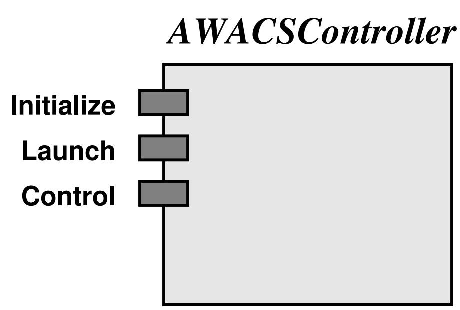


The *MissileController* object is responsible for launching missiles and for guiding them until they intercept their targets. The *MissileController* is very similar to the *InterceptorController*. The major difference between the two object types is the handling of "kills." It is possible for a missile to "hit" the wrong target (and be destroyed) by passing too close to it. If the missile is destroyed and its original target still exists, the guidance activity notifies the *TrackHandler* to allocate and launch a new weapon against the hostile track. Similar situations can occur when a missile's target is killed by another weapon. If the guidance thread is unable to locate the threat, a self-destruct command is issued to the Scenario Simulator to remove the missile.

The operations defined on the *MissileController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of a particular missile. Launching consists of computing an initial trajectory, creating a track for the missile in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched weapon is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers a weapon to its target by issuing intermittent guidance updates. The frequency of the guidance updates is determined by the distance between the missile and its target. Guidance updates are computed using a pursuit guidance algorithm.

6.3.3 AWACS Controller

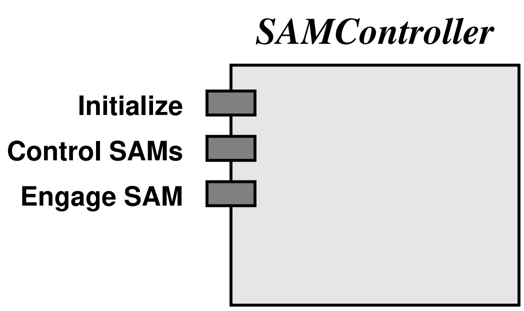


The *AWACSController* object guides an airborne early warning and control aircraft in an oval pattern over its coverage area. The coverage area for each AWACS is defined by the human operator. This placement information is passed to the *Launch* operation which sends a launch command to the Scenario Simulator and calculates the initial guidance information. Each AWACS launch is registered with the *WeaponsManager* so that the guidance activity can be restarted if it is lost because of a node failure. Control of the aircraft is then passed to the *Control* operation which calculates the course corrections necessary to maintain the oval. AWACS planes do not require course updates while flying the straight legs of their flight pattern. However, since guidance commands are issued in terms of absolute headings, they require frequent updates while navigating the circular portions of their course.

The operations defined on the *AWACSController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of an early warning aircraft. Launching consists of computing an initial trajectory, creating a track for the AWACS in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched AWACS is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers an AWACS in an oval pattern by issuing intermittent guidance updates. No course updates are required while the aircraft is flying the straight legs of the pattern. Frequent updates are required to navigate the turns at the end of the pattern.

6.3.4 SAM Controller

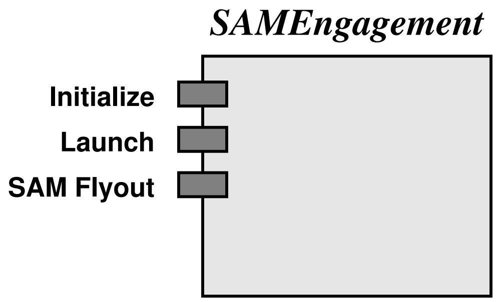


The terminal defense SAM batteries are managed by two objects—the *SAMController* object and the *SAMEngagement* object. The *SAMController* is responsible for detecting hostile tracks that are about to enter the range of the SAM batteries. When hostile tracks are detected within range, the *SAMController* engages the appropriate SAM by starting a thread in the *SAMEngagement.Launch* operation.

The operations defined on *SAMController* are:

- *Initialize* — initializes internal data structures and creates a thread in the *ControlSAMs* operation.
- *Control SAMs* — monitors the locations of hostile tracks to identify threats that will soon enter the range of a SAM battery.
- *Engage SAM* — initiates the *SAMEngagement.Launch* operation.

6.3.5 SAM Engagement



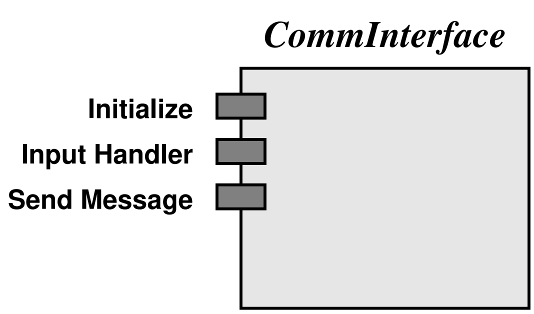
The *SAMEngagement* object simulates the aim-aim-fire sequence of a SAM battery using a sequence of delays and deadlines. The *Launch* operation assigns a particular SAM to a hostile craft and waits for the target to come within range. When the target is within reach, the *SAMFlyout* operation is invoked to simulate the aiming and firing sequence. If the *Launch* and *SAMFlyout* operations successfully meet their time constraints, the SAM is defined to be successful and a kill signal is sent to the Scenario Simulator. If any of the time constraints are not satisfied, the launch operation is retried. This sequence continues until the target is either killed or travels out of range of the SAM battery.

The operations defined on *SAMEngagement* are:

- *Initialize* — initializes internal data structures.
- *Launch* — assigns a particular SAM to a threat and monitors the target distance to the kill ring. When the target is in range, the *SAMFlyout* operation is invoked to engage the weapon.
- *SAM Flyout* — simulates the aim-aim-fire sequence of a SAM launch. Each stage of the sequence is simulated by a time-constrained delay.

6.4 System Support Objects

6.4.1 Communication Interface

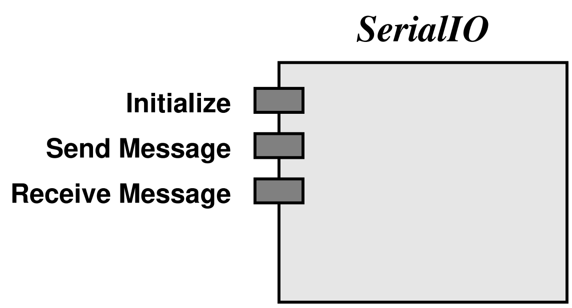


The *CommInterface* object manages communication between the ADSP (running under Alpha) and the simulated environment (running under UNIX). The actual message transmission is handled by either the *NetworkIO* or *SerialIO* objects. A thread in the *InputHandler* operation accepts messages as they arrive from UNIX and invokes operations on the appropriate destination objects to deliver the incoming data. When an Alpha thread needs to send a message to UNIX, it invokes the *SendMessage* operation with the message to be transmitted.

The operations defined on *CommInterface* are:

- *Initialize* — selects and initializes the communications link (either *NetworkIO* or *SerialIO*), creates a thread in the *InputHandler* operation to dispatch incoming messages, and initializes internal data structures.
- *Input Handler* — receives messages from the communication link and invokes operations on the appropriate objects based on the message type.
- *Send Message* — transmits an outbound message (from Alpha to UNIX) over the active communication link.

6.4.2 Serial I/O

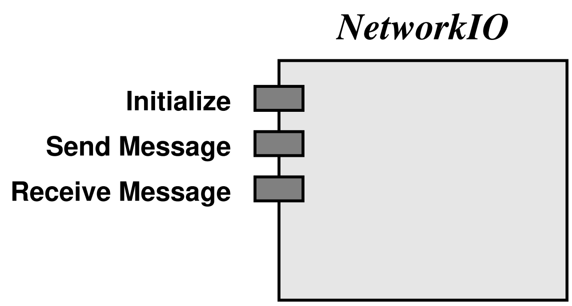


The *SerialIO* object provides a reliable communications link between an Alpha node and a UNIX host over an RS422 serial line. *SerialIO* uses a custom protocol to ensure the accurate reception of data packets. The protocol attaches a checksum and message number to each message sent across the link. When a message is received, the checksum is recomputed and compared with the transmitted value. If the checksums disagree, a negative acknowledgment (NACK) for that message is returned. If the checksums agree, no response is generated (implicit acknowledgment). When the sender receives a NACK, it re-transmits the corrupted message and all messages sent after the corrupted one.

The operations defined on *SerialIO* are:

- *Initialize* — initializes the serial line driver and protocol buffers and establishes communication with the UNIX side of the communication link.
- *Send Message* — transmits an outbound message (from Alpha to UNIX) over the serial line using the reliable serial protocol.
- *Receive Message* — reads the next message from the serial line and returns it in the caller's message buffer. The invoking thread is blocked until a message is available.

6.4.3 Network I/O

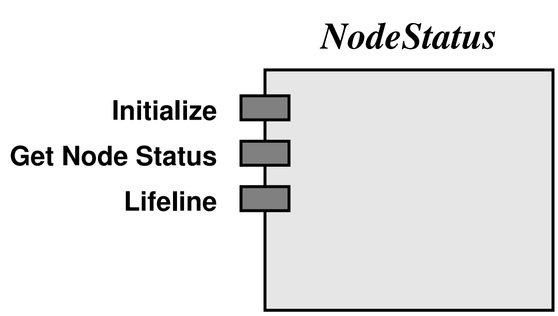


The *NetworkIO* object provides a reliable communications link between an Alpha node and a UNIX host using an Ethernet-based communication protocol. The *NetworkIO* object acts as an interface to the Communications Processor (CP) on the Alpha node. The CP is responsible for transmitting and receiving messages using the Alpha external communications (EXT) protocol. On the UNIX side of the connection, a special daemon handles the reception and transmission of EXT messages.

The operations defined on *NetworkIO* are:

- *Initialize* — initializes the message queue and establishes communication with the UNIX side of the communication link.
- *Send Message* — transmits an outbound message using the EXT protocol.
- *Receive Message* — reads the next message from the Communication Processor and returns it in the caller's message buffer. The invoking thread is blocked until a message is available.

6.4.4 Node Status



The *NodeStatus* object provides load and performance statistics for the node on which it resides. The *GetNodeStatus* operation returns information about such things as memory utilization, number of object instances, number of active threads, and scheduling queue lengths. The *Lifeline* operation is used by the *RecoveryManager* object to detect when a node has failed.

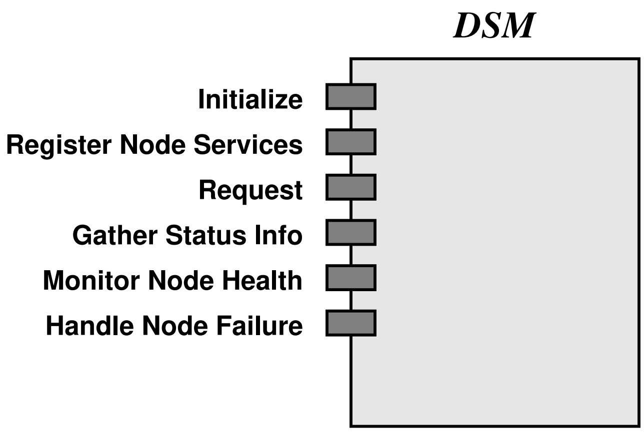
The operations defined on *NodeStatus* are:

- *Initialize* — initializes the statistics gathering buffers.
- *Get Node Status* — returns information about such things as memory utilization,

number of object instances, number of active threads, and scheduling queue lengths for the local node.

- *Lifeline* — blocks indefinitely on a semaphore. (A thread that invokes this operation should never return to the invoking object. If the thread does return, it means the thread has been broken due to a node failure.)

6.4.5 Distributed Services Manager



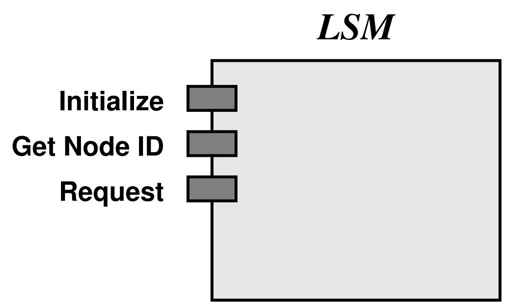
The distributed services manager *(DSM)* object manages the distribution of objects and threads among different nodes. When the application needs to create a new object or thread, it *Request*s the new service from the *DSM*. The *DSM* then places the new object or thread on the most lightly-loaded node. The actual creation of objects and threads is handled by a local services manager *(LSM)* on the target node. (This scheme of using the *LSM* as an agent was a convenient shortcut to ensure that threads would be rooted on the desired node. A more general solution is expected to be included as a mechanism in a later Alpha release.)

Load information is maintained by a thread that periodically requests status updates from the application nodes. The *DSM* is also responsible for initiating recovery operations in the event of a node failure. The *MonitorNodeHealth* operation sends "feeler" threads out to the different application nodes and initiates the *HandleNodeFailure* operation when any of the application nodes fail.

The operations defined on *DSM* are:

- *Initialize* — initializes local data structures and starts the status gathering and monitoring functions.
- *Register Node Services* — registers the *LSM* and *NodeStatus* objects for a node. The newly registered node is assigned a logical ID that is returned as a parameter.
- *Request* — invokes *LSM.Request* on the most lightly-loaded node in the system.
- *Gather Status Info* — periodically updates the status and loading information for each of the application nodes.
- *Monitor Node Health* — sends a "feeler" thread to the *NodeStatus.Lifeline* operation on a particular node.
- *Handle Node Failure* — marks a node as down and initiates a recovery operation on the most lightly-loaded node.

6.4.6 Local Services Manager

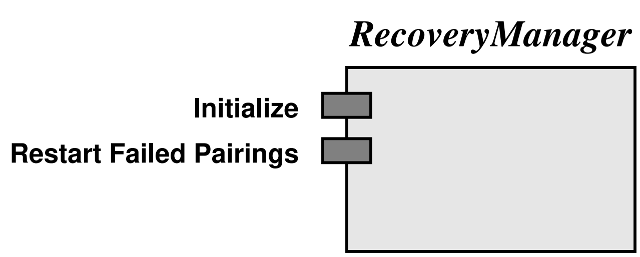


The local services manager *(LSM)* object acts as an agent of the *DSM*—creating local objects and threads on its behalf. The *LSM* is also responsible for registering the local node with the *DSM* when first powered up. When the node registers, it is given a logical node ID that is used by the application to track the location of objects and thread roots in case the node fails.

The operations defined on *LSM* are:

- *Initialize* — registers the local node with the *DSM* and initializes the logical ID of the local node.
- *Get Node ID* — returns the logical ID of the local node.
- *Request* — creates a local object or thread on behalf of the *DSM*.

6.4.7 Recovery Manager



The *RecoveryManager* object is invoked by the *DSM* when a node failure has been detected. The *RecoveryManager* is responsible for starting new instances of weapon (AWACS, Interceptor, Guided Missile, and SAM) control threads that were active on the failed node.

The operations defined on *RecoveryManager* are:

- *Initialize* — initializes internal state.
- *Restart Failed Pairings* — restarts controller threads that were active on a node that has failed.

7 Air Defense System Threads

The air defense system object types define the functions that can be performed by the application. The system threads define the sequence of activities that are actually executed.

The ADSP threads were defined by examining the functional decomposition given in Section 5.2. Threads were initially chosen by locating logically-related sequences of activities. Once the sequences had been refined, decisions were made about when a single thread should perform several jobs in succession and when separate thread instances should be created to execute concurrently. The resulting design defines several different types of threads. In some cases there is a single instance of a particular thread type in the entire system. For other types, separate threads are created to handle each track or each weapon that is active.

Tracking Threads

- Plot Correlation *(1 thread)*
- Track Database Maintenance *(1 thread)*

Mission Planning Threads

- New Track Dispatching *(1 thread)*
- Track Processing *(1 thread/new track)*

Weapon Systems Control Threads

- Interceptor Launch & Control *(1 thread/launch)*
- Missile Launch & Control *(1 thread/launch)*
- AWACS Launch & Control *(1 thread/launch)*
- SAM Launch & Control *(1 thread/launch)*
- SAM Monitoring *(1 thread)*

System Support Threads

- Input Processing *(1 thread)*
- Node Status Monitoring *(1 thread)*
- Node Lifeline *(1 thread/node)*
- Node Recovery Processing *(1 thread/failure)*

Figure 11: Air Defense System Threads

Figure 11 lists the different types of threads in the ADSP and shows how many instances of each are created. The remainder of this chapter describes each of the thread types in greater detail.

7.1 Tracking Threads

7.1.1 Plot Correlation

The plot correlation thread is responsible for updating the tracking database based on incoming radar reports. A single instance of the thread is rooted in the *PlotCorrelator.CorrelateFrame* operation. When a new frame of data has been received, the thread retrieves the next timestamp from the input buffer (see *PlotCorrelator.GetStamp)* and begins processing the radar reports. For each report, the thread correlates the location of the report with the contents of the tracking database (see *TrackDB.FindBestFit)* and updates the database as necessary. When new tracks are discovered, the thread registers their existence using the *TrackHandler.EnterTrack* operation so that a new track processing thread can be created. When an entire frame has been correlated, a new display list is transmitted to the Operator Console using the *CommInterface.SendMessage* operation. Finally, the plot correlation thread loops back and waits until the next frame of data is available.

7.1.2 Track Database Maintenance

The track database maintenance thread removes (*"ages"*) old tracks that are no longer active. A single maintenance thread resides in the *TrackDB.Age* operation. Every few seconds (once per radar frame), the thread scans the database for threads that have not been updated recently. Track entries that are too old (because the track moved out of range or was destroyed) are deleted.

7.2 Mission Planning Threads

7.2.1 New Track Dispatching

The new track dispatching thread creates a track processing thread for every new track that enters the system. A single thread rooted in *TrackHandler.Dispatch* waits for new tracks to be identified by the plot correlation thread, and creates a new thread in the *ProcessTrack* operation. Creating a thread for each track allows mission planning operations such as track identification and weapon assignment to be carried out concurrently for several tracks, and allows the time constraints for each of the threads to be set independently according to the threat potential of the associated track.

7.2.2 Track Processing

Track processing threads perform mission planning activities associated with new tracks. First, the *TrackID* object is invoked to determine whether the new track is friendly or hostile. If the track appears to be hostile, the track processing thread passes into the *TrackHandler.ProcessThreat* operation where functions such as threat assessment and weapon assignment are performed. The final act of the track processing thread is to launch any missiles or interceptors that are assigned to intercept the track. Once the weapons have been launched, the thread dies.

7.3 Weapon Systems Control Threads

7.3.1 Interceptor Launch and Control

The interceptor launch and control (L&C) thread launches an individual interceptor and provides any necessary in-flight guidance to ensure an accurate intercept with the target. A separate L&C thread is created for every interceptor launched. Each thread is started in the *InterceptorController.Launch* operation where it issues the launch request and registers the pairing with the *WeaponsManager*. The thread then invokes the *Control* operation to guide the interceptor to its target. The L&C thread is responsible for detecting when the target has been successfully intercepted and for guiding the interceptor back to its base after the completion of the mission. When the interceptor is safely home the thread dies.

Since a separate L&C thread is created for every interceptor, the system can independently set the time constraints of each weapon control activity based on the potential threat of its target. The exact nature of these time constraints is described in Sections 3.3.2 and 8.2.1.

7.3.2 Missile Launch and Control

The missile L&C threads are identical to the interceptor L&C threads except that missiles that survive their mission (because another weapon intercepted their target) are issued self-destruct commands rather than being guided back to their base.

7.3.3 AWACS Launch and Control

An AWACS L&C thread is responsible for launching a single AWACS aircraft and guiding it along an oval-shaped flight path. A separate thread controls each of the early warning aircraft. Each L&C thread adjusts its time constraints dynamically depending on whether the controlled aircraft is flying along a straight leg of the flight path (where course adjustments are not required) or is flying around an end curve (where adjustments must be made frequently).

7.3.4 SAM Launch and Control

SAM L&C threads manage the targeting and firing of individual SAM batteries. When the SAM monitoring thread detects a hostile target about to enter the range of a SAM battery, it creates a SAM L&C thread in the *SAMEngagement.Launch* operation. The L&C thread waits for the target to come within range, then invokes the *SAMFlyout* operation. The thread then enters a series of time constraints designed to simulate an aim-aim-fire operation (see Sections 3.3.3 and 8.2.1). If the time constraints are satisfied, the launch is defined to be successful and a kill message is issued to the Scenario Simulator. (This is the only instance where the ADSP system running under Alpha issues a kill command. In all other cases, kills are detected by the Scenario Simulator and must be discovered by the ADSP. The short time scale of SAM engagements made this small aberration necessary.)

7.3.5 SAM Monitoring

The SAM monitoring thread is responsible for detecting hostile tracks that are nearing the range of the terminal defense SAMs. A single monitoring thread is created at system initialization time. When a target is discovered (using *TrackDB.TrackHostiles*), the SAM monitoring thread assigns a SAM battery and initiates the engagement by creating a SAM L&C thread in the *SAMEngagement.Launch* operation.

7.4 System Support Threads

7.4.1 Input Processing

The input processing thread collects messages from the Communication Interface and distributes them to the appropriate objects based on the message type. The majority of messages received are radar plot reports which the input thread stores using the *PlotCorrelator.BufferReport* operation. If the *PlotCorrelator* object does not exist (*e.g.,* because of a node failure), the input processing thread requests the creation of a new correlator (see Section 8.1).

7.4.2 Node Status Monitoring

A single node status monitoring thread travels around the system gathering node status information (*e.g.,* number of active threads, memory utilization, etc.) for each node. This information is used by the *DSM* to direct load balancing and to display monitoring information on the Experimental Control Console. The status monitoring thread is rooted in *DSM.GatherStatusInfo* and travels between the *NodeStatus.GetNodeStatus* operations on each node.

7.4.3 Node Lifeline

A lifeline thread is created for each node when it registers itself at system initialization or after a node recovery. The thread is created in *DSM.MonitorNodeHealth* where it immediately invokes *NodeStatus.Lifeline* on the destination node. If a monitored node fails, Alpha reports a thread break exception to the *MonitorNodeHealth* operation and the lifeline thread invokes *DSM.HandleNodeFailure* to initiate recovery processing. A more detailed description of the failure recovery mechanism is given in Section 8.1.

7.4.4 Node Recovery Processing

When a node fails, any weapons L&C threads that were active on that node must be restarted on the remaining nodes. The *DSM.HandleNodeFailure* operation creates a node recovery processing thread in the *RecoveryMgr.RestartFailedPairings* operation. The recovery thread retrieves a list of the lost threads using the *GetFailedPairings* and *GetFailedSAMInfo* operations of the *WeaponsManager*. It then creates new L&C threads for each of the pairings using the *DSM.Request* operation. When all of the threads have been regenerated on other nodes, the recovery processing thread dies.

8 Technology Evaluation

The primary goals of the Alpha application effort were to apply the decentralized, real-time technology incorporated in the Alpha operating system, and to evaluate that technology in a realistic context. This chapter discusses and evaluates how the advanced systems technology provided by Alpha was used to meet the distribution, survivability, and timeliness requirements of the air defense application.

8.1 Distribution and Survivability

Distributed computers are rapidly becoming a major component in the design of large application systems. Alpha is designed to integrate these physically dispersed nodes into a single decentralized computer system—providing transparent access to, and unified management of, distributed resources. This integration allows applications to obtain the many performance, availability and survivability benefits of distributed systems without incurring many of the difficulties often associated with conventional distributed programming.

The air defense system prototype is designed to operate, without modification, either on a single centralized computer or on a network of distributed computing nodes. The actual system has been tested in configurations containing between one and four Alpha nodes. When executing in a distributed environment, the ADSP automatically distributes work among the available processors. If one or more of the nodes fail, the ADSP automatically recovers and continues to operate using the remaining nodes. If new or repaired nodes become available, they are seamlessly integrated into the already running system.

To achieve this level of availability and survivability, the design team at GD (working closely with the Alpha group at CMU) developed a plan for distributing the ADSP objects and threads among the available nodes. Figure 12 illustrates a sample distribution of objects and threads between two of the application nodes. The bar graph at the left of each display monitors the total number of threads active on the node. The listing on the right shows the object instances present on the node and the number of threads active in each object. (The object names listed in Figure 12 differ slightly from those used in this report. The expanded names used in the report correspond directly to the object types used in the ADSP, but are more legible.) The re-

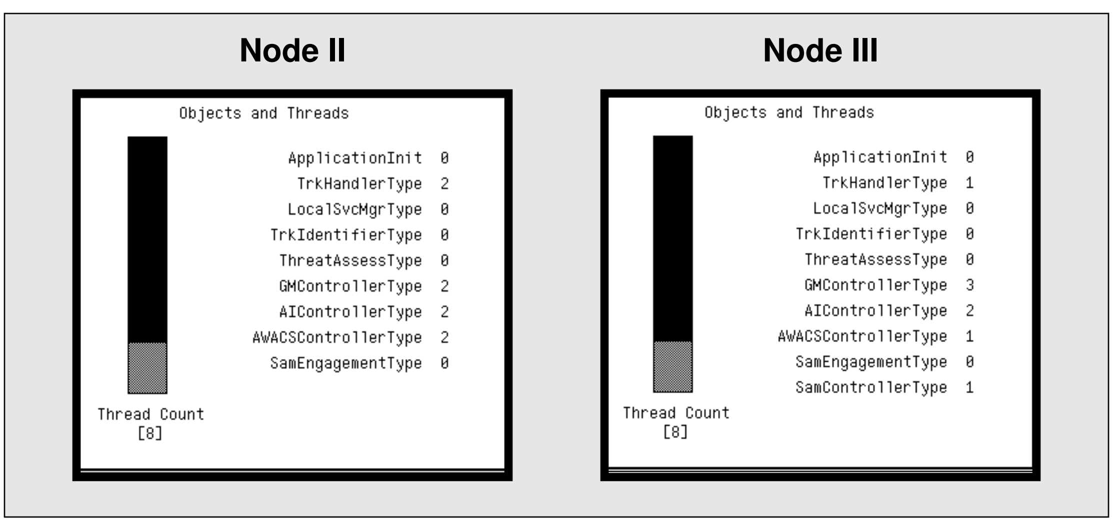


Figure 12: Typical Distribution of Objects and Threads

Stateless Computation Servers

Track Handler
Track Identifier
Threat Assessment
Guided Missile Controller
Interceptor Controller
AWACS Controller
SAM Engagement
Node Status
Local Services Manager
Recovery Manager

Recoverable Stateful Servers

Plot Correlator
SAM Controller

Essential Data Repositories

Track Database
Weapons Manager

Figure 13: Object Distribution Classifications

mainder of this section describes in greater detail how the objects and threads are distributed and how that distribution allows the ADSP to recover and continue operating after node failures.

8.1.1 Object Distribution

Since Alpha provides for the transparent distribution of objects among processing nodes, applications can largely be designed as if they were going to execute on a single processor. Distribution issues become important when considering the survivability and availability requirements of the application. In particular, the survivability requirements determine which objects need to be inclusively replicated, while availability considerations influence the number and placement of object instances.

The survivability requirements of the air defense application were outlined in Section 3.4. In designing the ADSP, those requirements were used to classify the application objects into three categories—stateless computation servers, recoverable stateful servers, and essential data repositories. The categories are primarily distinguished by the amount and type of application data maintained in the objects. Figure 13 lists the object types in each class. The remainder of this section outlines their characteristics.

The stateless computation server (SCS) objects do not maintain any important state information between incoming operation invocations (*i.e.,* they contain executable code, but no important local data). As a result, if one of the SCS objects disappears because of a node failure, no vital information is lost. Independent instances of the SCS objects can be maintained concurrently on several nodes since no application data must be kept consistent between the copies. For the ADSP, the designers decided to create local instances of every SCS object on each of the application nodes. Maintaining local copies on each node reduces the number of remote invocations required and simplifies failure recovery by eliminating the need to create replacement objects after a node failure.

Unlike the SCS objects, the recoverable stateful server (RSS) objects do maintain application state information between invocations. The information they store, however, is often transient and can either be recreated or replaced if it is lost because of a node failure. The best example of an RSS object is the *PlotCorrelator*. The *PlotCorrelator* buffers radar reports until they are processed. If the *PlotCorrelator* is lost, the radar reports that have been buffered will be lost also. However, radar reports are constantly being generated by the sensors. Under the stressful conditions of a node failure, the ADSP designers decided that it is better to process current reports as they arrive than to use potentially limited processor cycles to restore and process outdated information. At most one instance of any RSS object exists at any particular time. If that instance disappears because of a node failure, a new instance is created on demand on one of the remaining nodes.

The essential data repository (EDR) objects maintain information that is vital to the continued operation of the ADSP. The tracking database and weapons resource database are the primary examples. The data stored in EDR objects should remain available and consistent as long as any of the application nodes are functioning. To achieve this level of reliability, the EDR objects should be inclusively replicated on multiple application nodes. Since multiple copies of replicated objects are maintained in a consistent state, the loss of a single node would not result in the loss of the data. The number of replicas maintained can be set depending on the level of reliability desired.

Since the Alpha replication facilities were not in place in time for the ADSP demonstration, single instances of the EDR objects were placed on a system interface node along with the communications functions. The need for an interface node originated because of the serial communication link between the ADSP and the experimental environment running under UNIX. We did not have the time or the desire to implement redundant communication links, and so decided to dedicate one node to act as the interface and to hold (temporarily) the EDR objects. The remainder of the nodes are interchangeable "application" nodes that handle the bulk of the application processing.

8.1.2 Thread Distribution

In addition to survivability, one advantage of distributed systems is the ability to execute several activities concurrently. In the ADSP, multiple instances of several thread types can be active at any time. While one node is processing a new frame of radar data, other nodes may be computing guidance information for weapons that have already be launched. In general, it is desirable to distribute the processing load evenly among the nodes (to avoid local overloads). Load balancing methods for distributed system are the topic of continuing research. For the ADSP, load balancing is managed by the *DSM*. The *DSM* maintains status information on each of the nodes in the system. It uses this status information (including number of active threads,

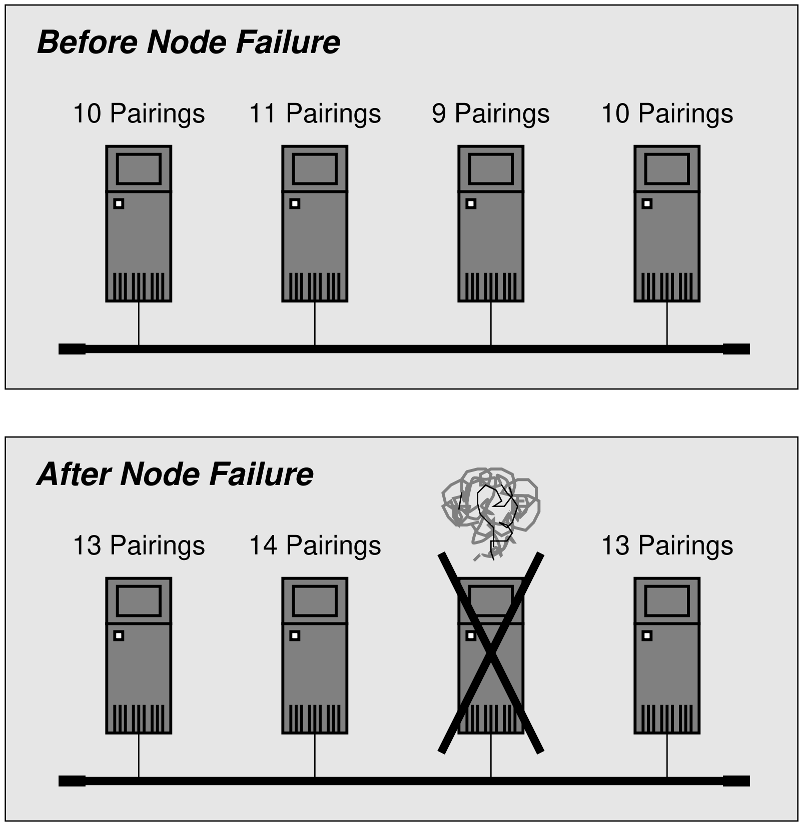


Figure 14: Reconfiguration After a Node Failure

memory utilization, *etc.*) to estimate the load on each node. Each time a new mission planning or weapon systems control thread is required, an invocation is made on the *DSM* to create the new activity. The *DSM* then forwards the request to the *LSM* on a lightly-loaded node where the thread is then created.

This load balancing scheme works relatively well for the ADSP. Eventually, however, many of the services provided by the *DSM* will be supplanted by more general system services offered by Alpha. Our experience with the *DSM* has given us a better understanding of the types of services that should be offered and the ways in which applications are likely to use them.

8.1.3 Node Failure Response

One of the fundamental survivability requirements specified by the application design team was that the ADSP should continue to operate correctly after one or more node failures. This capability requires both that essential application data remain available and that partially-completed activities active on the failed node be cleaned up and, if necessary, restarted. The designers ensured that important data would remain available after a failure by keeping vital application information in the essential data repository objects. The remaining task of cleaning up and restarting failed activities is handled by Alpha and by the *RecoveryManager*.

The distribution and exception-handling facilities provided by Alpha already handle a significant amount of the processing required when a node fails. Alpha automatically detects a node failure and initiates system cleanup operations. Any threads that were active on the failed node are trimmed and an exception is returned to the invoking object. Exception handling code then initiates application-level recovery operations.

Once the system-level cleanup has been completed, a node failure is signalled to the *DSM* (via the node lifeline thread for the failed node). The *DSM* removes the failed node from the resource pool and starts a new thread in the *RecoveryManager* object. The recovery operation queries the *WeaponsManager* for a list of guidance threads that were active on the failed node. Replacement activities are then created and distributed across the remaining nodes in the system. Any recoverable stateful server objects that were active on the failed node are recreated on demand on one of the remaining nodes. Figure 14 illustrates how weapons guidance threads are redistributed in response to a node failure. In this example, the nine target/weapon pairings being controlled by the third node are reassigned to the other available nodes.

During public ADSP demonstrations, node failures are simulated by turning off the power to one of the application nodes. By monitoring the node displays on the Experimental Control Console, it is possible to watch the defense system recover by restarting the lost guidance activities on the remaining nodes. The “failed” node is then turned back on and re-integrated into the running defense system.

8.2 Time-Driven Resource Management

The primary requirement of the air defense system is that it prevent hostile targets from violating U.S. airspace. To meet this requirement, the tracking, mission planning, weapon systems control, and system support activities must be carried out on time. If a SAM launch sequence is completed ten seconds after its target has dropped a bomb on New York City, it has failed just as much as if the launch had never been completed. The situation is further complicated by the diverse characteristics and dynamic nature of the time constraints presented by the ADSP. Applications such as air defense have complex timeliness requirements that are not easily expressed in terms of fixed, periodic, or hard deadlines. For example, the urgency and importance of issuing guidance updates when a target is hundreds of miles away is significantly lower than when the target is about to reach the coastline. Finally, the ADSP must continue operating to preserve the coastal integrity even when insufficient processing resources are available (as may be the case in the event of one or more node failures).

Alpha supports this type of complex and changing real-time application through its use of a unified time-driven resource management mechanism. In Alpha, timeliness requirements are specified dynamically by application threads using time-value functions. Time-value functions allow a variety of time constraints to be specified both concisely and accurately. This information is then used by the best-effort scheduler to allocate processor cycles in a way that maximizes the benefit to the application. Under normal conditions this means that the timeliness requirements of all activities will be satisfied. If the system is overloaded (*e.g.,* because of a node failure), those activities that are the most valuable to the application as a whole will be given preference. For a more complete discussion of Alpha’s time-driven resource management mechanisms and policies, refer to Section 2.2.

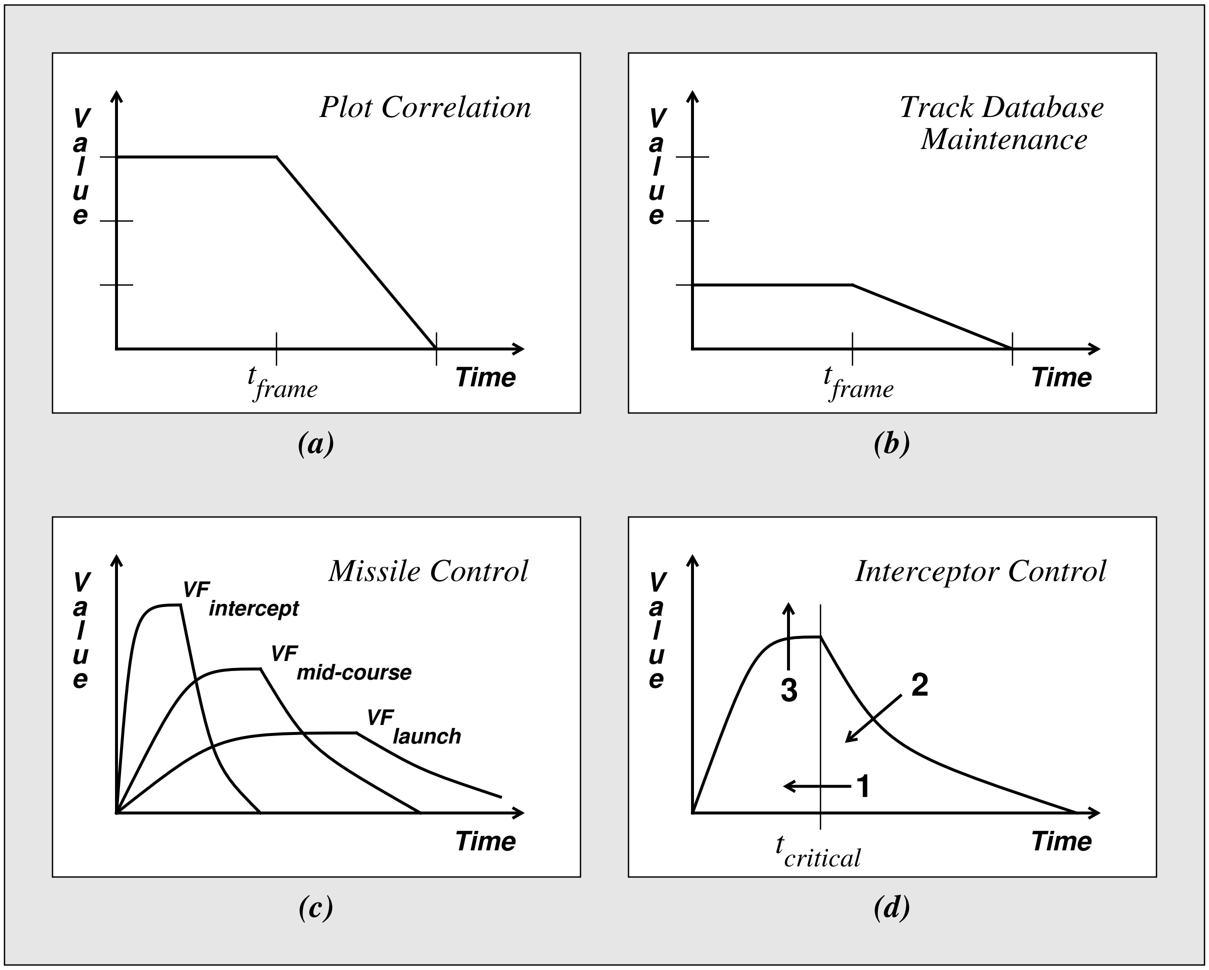


Figure 15: ADSP Time-Value Function Examples

8.2.1 Application Time-Value Functions

The timeliness requirements of the air defense application were discussed in Section 3.3. Once the application threads had been defined, the designers found that converting these textual requirements into actual time-value functions was fairly straightforward. A significant amount of system "tuning" was eliminated since the team did not have to massage the requirements into a foreign set of real-time specifications such as rate groups or priority levels.

To derive time-value specifications from the written requirements, each of the time-constrained activities was analyzed to determine its critical time and its completion value over time (to determine the shape of the function). The different activities were then compared to determine the amplitude of the function by estimating their relative value to the system during overload situations. These considerations combined to define the final time-value function specifications.

The plot correlation and track database maintenance threads have critical times corresponding to the radar frame arrival rate. In both cases, it is best if the processing is completed before the next frame of data arrives. However, it is acceptable for the processing to slip as much as one additional time frame under extreme overload situations. The time-value functions for these threads are shown in Figure 15(a) and Figure 15(b). As illustrated, both functions share the same shape and critical time. The plot correlation activity, however, has a much greater value to the system under overload conditions.

The timeliness requirements for the missile and interceptor launch and control threads vary over the course of an engagement. After launch, the guidance control threads must issue timely course updates to ensure a successful intercept. However, the required frequency of these updates and the importance of completing the course corrections at the desired time change both as the distance between the weapon and the target and from the target to the coastline decrease. Figure 15(c) illustrates how the time-value functions for weapon control threads vary over the course of an intercept. The characteristics can be summarized as follows:

- *loose time-constraint*s — The critical time indicates the time at which course correction is preferred, not absolutely required.
- *variable critical times* — Course corrections are needed more often as the distance between the threat and the weapon decreases. The desired update time is one fourth of the predicted time to intercept. Arrow 1 in Figure 15(d) illustrates how the critical time changes as the intercept nears.
- *variable "hardness"* — It becomes more important to use the most recent position information as the distance between the threat and weapon decreases. Therefore as the distance narrows, it may be preferable to abort a late update and restart the guidance calculations with fresh information. Arrow 2 in Figure 15(d) shows how this requirement is reflected by a decrease in the completion value of the constraint after the critical time.
- *dynamic maximum* — The value of successfully completing an intercept corresponds to the threat potential of the target being intercepted. The perceived threat depends on changing parameters such as distance from coastline. Arrow 3 in Figure 15(d) illustrates how the time-value function is scaled upward as the threat increases.

Although very different from more conventional real-time specification methods, the ADSP designers were pleased with the expressive ability and ease of deriving time-value functions. Other specification methods would likely not have allowed time constraints for individual activities to be specified so easily. If only one constraint could be specified, all instances of an activity would have to execute under the most stringent constraint of any instance—potentially causing time constraints to be missed unnecessarily. The ability to specify importances, or values, for separate instances of a thread allow the system to make much better resource decisions when the system is overloaded.

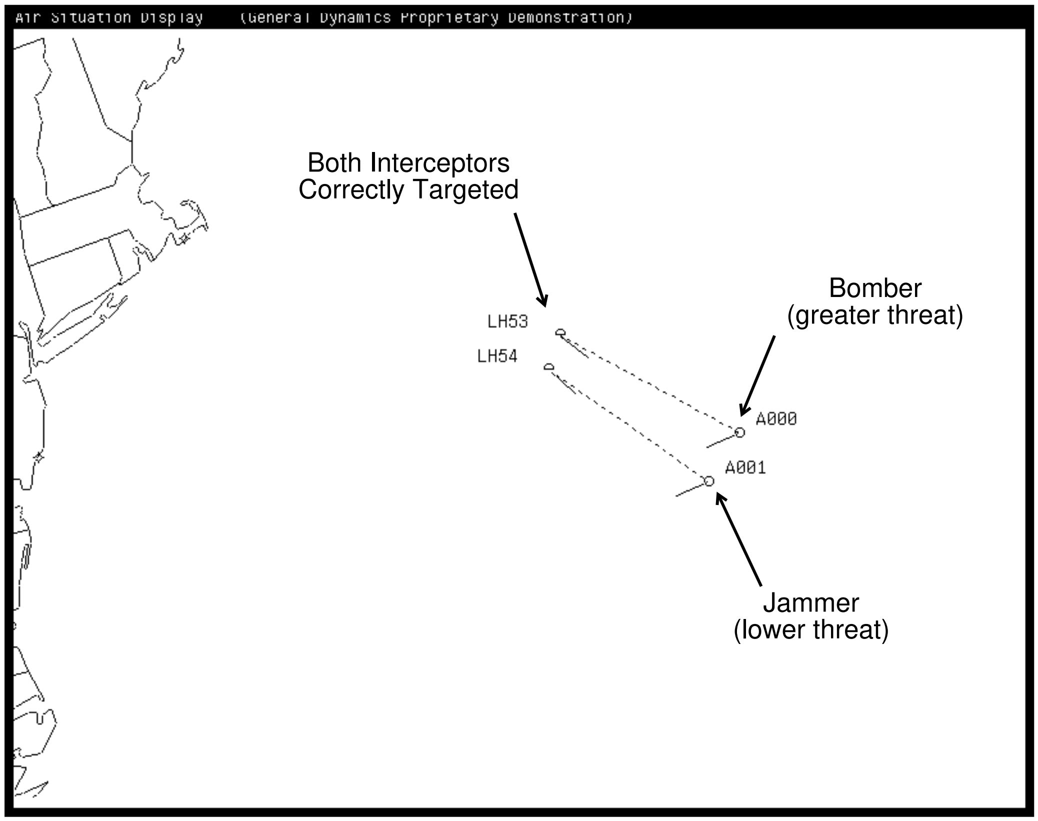


Figure 16: Normal Behavior

8.2.2 System Overload Behavior

Many real-time systems react to temporary overload conditions by breaking completely and permanently. To reduce the probability of such failures occurring, designers may require such massive redundancy that "overloads will never happen." Besides being wasteful, such techniques may not work if a system suffers multiple simultaneous failures (as might be expected during an intentional attack).

The best-effort scheduler used by Alpha responds to overload conditions by concentrating its efforts on completing activities that will be the most valuable to the application (as specified by the time-value functions). For the ADSP, this means that the Operator Console display is accurately maintained and that targets within the range of the SAM defenses receive primary attention. Other targets receive attention based on their perceived threat.

The visible effect of this overload management policy is that guidance updates for weapons paired with the greatest threats are issued on-time, so the most significant threats are intercepted before they can inflict damage. Guidance updates for other weapons may be delayed. The delay may result in longer flight paths and later intercept times, but will not allow hostile targets to leak through unless the system is so overloaded that even the most essential activities cannot be completed.

Figures 16 and 17 illustrate the overload handling policy. In this case, two friendly interceptors (LH53 and LH54) have been paired with a hostile bomber (A000) and a hostile jammer (A001) respectively. These pairings are indicated by dashed pairing lines between the friendly

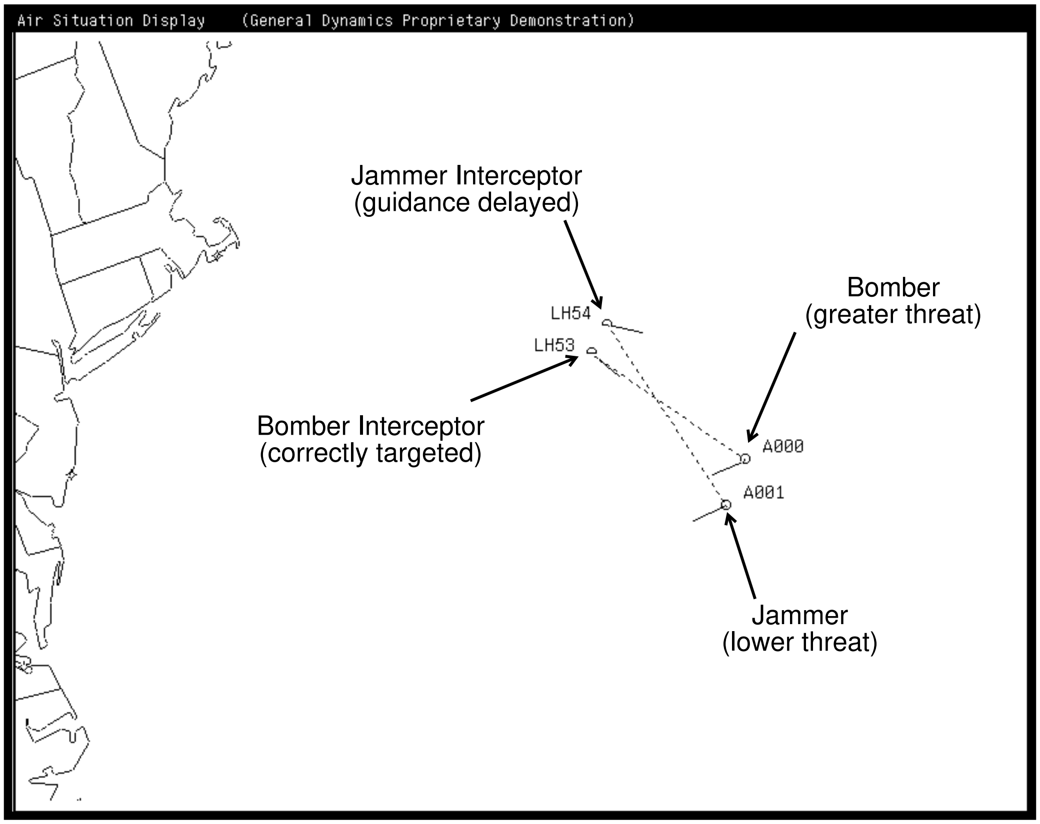


Figure 17: Overload Behavior

interceptors and the threats. In the normal case (Figure 16), both interceptors receive guidance updates when needed and are correctly directed toward their targets. (Track headings and speeds are indicated by the direction and length of the line emanating from the track marker.) In Figure 17, an artificial overload has been imposed on the ADSP. In the overload case, the interceptor paired with the bomber (a great threat) is given preference over the interceptor paired with the jammer (a lesser threat). As a result, the bomber is correctly targeted by LH53, while guidance updates for LH54 are delayed (LH54 is not flying toward its target and is farther away from the threat). The aggregate result of this behavior is that hostile targets tend to penetrate deeper when the system is overloaded than under normal conditions. Even under overload conditions, however, the ADSP continues to satisfy its mission.

9 Conclusions

The results of the Alpha application effort have been extremely worthwhile and very encouraging. Using Alpha, a small group of programmers was able to design and implement a realistic air defense system prototype in a period of a few months. The prototype has successfully demonstrated many of Alpha's distribution, time-driven scheduling, and survivability mechanisms, and has validated our belief that Alpha is well-suited for the design and construction of large, distributed, real-time applications. This chapter relates some of the many lessons we have learned from the effort, and describes some of the continuing work that is in progress.

9.1 Lessons Learned

9.1.1 Decentralized Computing

Alpha is designed to integrate a set of physically dispersed computer nodes into a single decentralized system. Our experience with the ADSP indicates that we have made great strides toward that goal. Objects can be dynamically created at any node, and can be invoked without knowing the node on which they actually reside. Application threads move freely across node boundaries—carrying attributes such as atomicity and time constraints with them. A major benefit of this decentralization is that it allows applications to obtain the availability and survivability benefits of physical distribution without incurring many of the difficulties often associated with conventional distributed programming.

9.1.2 Programming Model

The application designers at GD found the Alpha programming model easier to use than conventional process-based models. The object/thread paradigm allowed the design team to think in application terms when designing the structure of the software. Since there was no conversion step to translate the natural system structure into foreign constructs such as client/server processes, the overall design time was shortened. The requirements specification led quickly to an implementation that was both easy to understand and simple to modify as requirements changed.

9.1.3 Time-Driven Resource Management

Our experience with the ADSP has confirmed that C^2/BM applications have timeliness requirements that are not easily expressed in terms of fixed, periodic, or hard deadlines. The coastal air defense application presented a variety of timeliness requirements. Activities such as radar plot correlation have roughly periodic arrival rates, but soft time constraints. Weapons guidance activities have dynamically changing timeliness requirements whose duration, "hardness," and importance vary as the encounter progresses. Finally, activities such as SAM launch control have strict hard deadlines, but occur aperiodically. None of these examples match the popular notion that real-time tasks are either periodic with hard deadlines or aperiodic with loose timing requirements.

In comparison, the ADSP has demonstrated that the time-driven scheduling facilities provided by Alpha are appropriate for use in large distributed systems. Time-value functions allow a variety of system timeliness requirements to be specified both simply and accurately. The best-effort scheduling algorithm used for the ADSP successfully allocates processor cycles to ensure that, when possible, activities are completed on time. When not enough resources are available (*e.g.,* because of a node failure), the overload handling capabilities of the best-effort algorithm keep the ADSP operating and allow it to continue fulfilling its primary mission requirements.

9.1.4 System Survivability

Mission-oriented systems such as the ADSP need to continue functioning even if parts of the computer system fail. Designing a system that could weather node failures was made much easier by the survivability mechanisms provided by Alpha. Alpha automatically detects node failures, returns the system to a consistent state, and reports the failure to the affected application threads via a uniform exception handling mechanism. As a result, the application designers could concentrate on developing a recovery policy that ensures that no vital information or activities are lost. Because the operating system handles many of the details, the actual implementation and testing of the recovery operations for the ADSP was relatively simple. The resulting application successfully recovers from multiple node failures and even allows repaired nodes to be re-integrated into the running system.

Our experience designing the application-specific distributed services and recovery managers has allowed us to identify additional system-level services that would aid the development of survivable applications. Future releases of Alpha will incorporate this experience in the form of system service objects that provide much of the functionality of the *DSM* and *RecoveryManager*, but in a more general and consistent manner.

9.1.5 Program Management

One unique aspect of the application effort was its timing with respect to the development of the operating system. Since Alpha was being developed in tandem with the application, good communications between the operating system group at CMU and the application team at GD were vital. The use of a dedicated liaison, who spent a significant amount of time at both sites, worked extremely well under these circumstances. The people at GD knew who to ask when questions arose (which, considering the dearth of Alpha documentation at the time, they often did). Meanwhile, the group at CMU could concentrate their efforts on developing Alpha. The liaison, being familiar with both the ADSP code and the operating system internals, could efficiently investigate bugs as they appeared, and could work to isolate and fix problems with minimal disruption of the application or operating system development.

To support the application effort, the Alpha group had to reorganize the project schedule and priorities somewhat. Implementation of some features was deferred (notably secondary storage and atomic transactions) in order to expedite the development of other features required by the ADSP. Features that were accelerated or enhanced beyond their planned level of functionality include the time-driven scheduling policies, the application programming environment, Alpha/UNIX interoperability mechanisms, and asynchronous exception handling. Any negative impact of the schedule changes, however, was far outweighed by the benefits of and experience gained from the application effort.

9.2 Future Directions

The ADSP was the first significant application developed for Alpha. One of the goals in undertaking such an early application effort was to identify areas where further research and development efforts would be most effective. As expected, the experience has highlighted several areas where research is continuing. The success of the effort has also prompted us to implement other applications in an attempt to expand our understanding of different application requirements and determine how well they are met by the mechanisms and policies provided by Alpha.

9.2.1 Related Research

Our experience with the ADSP has highlighted several areas where additional research is warranted. In particular, it seems vital that we continue our exploration of real-time scheduling techniques, object replication, and real-time, atomic transactions.

The best-effort scheduling algorithm employed in the prototype has performed well. However, Alpha researchers have identified methods of improving its ability to handle certain types of activities. In particular, two Ph.D. theses are addressing the issues of threads with dependency relationships and time-constrained threads that span multiple nodes (see [Clark 88], [Archons 88]). We have also recognized the need to develop a better methodology for deriving time-value functions and better tools for computing the execution-time estimates required by many real-time scheduling algorithms.

9.2.2 System Enhancements

The ADSP continues to be a valuable tool for experimenting with Alpha. By more thoroughly monitoring the current system and by implementing extensions and enhancements, we can learn significantly more from the air defense application. To date, only limited performance monitoring of the ADSP has been carried out. By further instrumenting the system, it should be possible to learn a great deal about the resource request patterns of a real application. This information would provide valuable insight into the nature of C^2/BM applications, and would allow the Alpha designers to concentrate performance tuning efforts in areas that would make the greatest difference.

The ADSP should also provide an excellent environment for testing new Alpha functionality. The essential data repository objects would be ideal candidates for exercising such facilities as object replication and atomic transactions. Similarly, much of the functionality of the *DSM* object could gradually be migrated into more general system-level services that support system initialization, load distribution, and failure reconfiguration. Already having a large application that can benefit from these facilities will prove extremely valuable when testing their functionality.

Because the ADSP was designed as a prototype, it would be relatively simple to enhance the system by increasing the fidelity of many of its functions. Many of the mission planning objects could be changed to use more sophisticated algorithms or could be converted into decision-support systems for a human operator. The system could also be enhanced to support additional functions such as multi-sensor fusion. These types of enhancements should especially interest application developers who could gain experience designing command and control systems in the Alpha context.

9.2.3 Future Applications

Experience with the ADSP indicates that the best-effort scheduler works well in large real-time systems. However, the complexity of the ADSP limits the direct visibility of scheduling decisions, and makes it difficult to quantify the performance of the scheduler. For these reasons, the Alpha designers are developing an application designed explicitly to test the behavior and performance of the Alpha scheduling facility. The new application will allow us to monitor the decisions made by the scheduler in a more controlled environment, and will allow us to compare the results obtained using alternate scheduling algorithms. The initial application uses

Alpha to move a set of simulated paddles that reflect balls travelling through the air. Different scheduling algorithms move the paddles in different patterns—allowing a direct, graphical display of the processor allocation policies. A separate technical report will detail these experiments when they have been completed.

In addition to specialized application systems such as the scheduling application, we anticipate that other real-time, C^2/BM prototypes will be built using later generations of Alpha. Planning for such application systems, involving several industrial partners, is already in progress at Concurrent Computer Corporation.

References

[Archons 88] Archons Project
Alpha Preview: A Briefing and Technology Demonstration for DoD.
Presentation Notes, Archons Project Technical Report #88031, Department of Computer Science, Carnegie Mellon University, March, 1988.

[Clark 88] Clark, R. K.
Scheduling Dependent Real-Time Activities.
Ph.D. Thesis Proposal, Department of Computer Science, Carnegie Mellon University, October, 1988.

[Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.

[Fairley 85] Fairley, R. E.
Software Engineering Concepts.
McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill, New York, 1985.

[Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May, 1986.

[Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.

[Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie Mellon University, March, 1988.

[Northcutt 88d] Northcutt, J. D. and Clark, R. K.

The Alpha Operating System: Kernel Internals.
Archons Project Technical Report #88051, Department of Computer Science, Carnegie Mellon University, May, 1988.

[Northcutt 88e] Northcutt, J. D.
The Alpha Operating System: Programming Utilities.
Archons Project Technical Report #88041, Department of Computer Science, Carnegie Mellon University, April, 1988.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie Mellon University, April, 1988.

Appendix I: Application Program Example

The following pages contain the source code for the *TrackHandler* object. The *TrackHandler* object coordinates the response to new tracks that enter the system. The *ProcessTrack* operation is invoked for every new track. It is primarily responsible for establishing the identity of unknown tracks. When hostile tracks are discovered, *ProcessThreat* uses the other mission planning objects to assess threat potential, choose weapons, allocate weapons, and initiate weapons launch. *ProcessThreat* encapsulates the tactical decision functions that determine how the system responds to threats.

For more details about the syntax used in coding the ADSP, refer to [Shipman 88].

```
/*********************************************************************
 *                                                                   *
 *                    Distributed C2 Demonstrator                    *
 *                                                                   *
 *               General Dynamics Fort Worth Division                *
 *            Carnegie-Mellon University Archons Project             *
 *                                                                   *
 *********************************************************************/

OBJECT TrkHandlerType {

/*************************************************************************\
*
*  Track Handler:
*       directs the processing of tracks throughout the system.
*
* OPERATIONS:
* -----------
*    Initialize - Initializes the TrackHandler.  Currently just accepts the
*        CAPAs that are needed for processing.
*
*    ProcessTrack - Handles the processing of a new track that has entered
*        the system.  New tracks are identified, and if , handed off
*        to the threat handler.  Since we don't want to lose this activity
*        if a node fails, this activity is registered with the TrackDBMgr.
*
*    ProcessThreat - Determines the correct reaction to a threat that for
*        some reason needs to be reconsidered.  This may happen when the
*        threat is first identified, or when a previous intercept attempt
*        has failed.  This operation directs the process of 'ThreatAssesment',
*        'WeaponsAssignment', and 'WeaponsLaunching'.  Once a weapon has been
*        launched, its control is passed to a dedicated thread.
*
*    Recover - Recovers from a node failure by restarting 'TrackHandler' and
*        'WeaponsControl' threads.
*        NOTE: If the node handling the recovery dies before it has finished,
*        the rest of the recovery will NOT be completed and work will be lost.
*
* HISTORY
* -------
* 12-Dec-87  David Maynard (dpm) at Carnegie-Mellon University
*       Modified deadline constructs to meet new execution times.
* 20-Oct-87  Duane Northcutt (jdn) at Carnegie-Mellon University
*       Modified to use the new communications interface.
* 17-Aug-87  Russell Kegley (rbk) at General Dynamics
*       Added initial deadlines.
* 23-Jun-87  Betsy Zimmerman (betsy) at General Dynamics
*       Created.
*
\*************************************************************************/
```

```
/***************\
*
* INCLUDE FILES
*
\***************/

#include    "../h/c3demo.h"
#include    "../h/defs.h"
#include    "../h/trackdb.h"
#include    "../h/distrsvcs.h"
#include    "../h/wpns.h"


/*******************\
*
* PERSISTENT STORAGE
*
\*******************/

#undef DEBUG
#undef RBKBUG

WELLKNOWN CAPA SemaphoreManager SemaphoreManager;
WELLKNOWN CAPA ThreadManager    ThreadManager;
WELLKNOWN CAPA TempServices     TempServices;
WELLKNOWN CAPA CommIntfType     Comm;
WELLKNOWN CAPA DistrSvcMgrType  DSM;

CAPA TrackDBMgrType     TrackDB;
CAPA WeaponsMgrType     WeaponsMgr;
CAPA TrkHandlerType     TrkHandler;
CAPA ThreatAssessType   ThrtAssess;
CAPA TrkIdentifierType  TrkID;
CAPA LocalSvcMgrType    LSM;

/* Thread CAPA's */
CAPA Launch_AI_Thread;
CAPA Launch_GM_Thread;
CAPA Launch_SAM_Thread;

node_id_t       LogicalNodeID;

/* TrackQ Definitions */
#define TRACKQ_SIZE     100

CAPA Semaphore  TrackQSem;
CAPA Semaphore  TrackLockSem;

id_type trackq[TRACKQ_SIZE];

int     trackq_head,
        trackq_tail;
```

```
/**********************\
*
* OPERATION DEFINITIONS
*
\**********************/

OPERATION Init(IN CAPA TrkHandlerType     Handler,
               IN CAPA TrackDBMgrType     TDB,
               IN CAPA WeaponsMgrType     Weapons,
               IN CAPA ThreatAssessType   ThreatA,
               IN CAPA TrkIdentifierType  TrackID,
               IN CAPA LocalSvcMgrType    LocSM)
/*
* Initializes the TrackHandler.  Currently just accepts the CAPAs that
*  are needed for processing.
*/
{
    status_msg_t        stat;
    u_int               size;
    CAPA                DispatchThread;
    extern u_long       OCC_OO_TrkHandlerType_Dispatch;

    /* save CAPAs */
    TrkHandler = Handler;
    TrackDB = TDB;
    WeaponsMgr = Weapons;
    ThrtAssess = ThreatA;
    TrkID = TrackID;
    LSM = LocSM;

    /* initialize the track queue data structures */
    trackq_tail = 0;
    trackq_head = (TRACKQ_SIZE - 1);

    /* create the producer/consumer semaphore for the track queue */
    /*** NOTE:  Do not turn tracking on for this semaphore ***/
    INVOKE SemaphoreManager.AllocSem(IN (u_long)0,
                                     IN (boolean)FALSE, IN (boolean)FALSE,
                                     OUT (CAPA Semaphore)TrackQSem)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: sem create err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }
```

```
    /* create semaphore for locking new track queue */
    INVOKE SemaphoreManager.AllocSem(IN (u_long)1,
                                     IN (boolean)FALSE, IN (boolean)FALSE,
                                     OUT (CAPA Semaphore)TrackLockSem)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: locksem create err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }

    /* start the dispatch thread */
    INVOKE ThreadManager.Create(IN (CAPA)TrkHandler,
                                IN (u_long)OCC_OO_TrkHandlerType_Dispatch,
                                OUT (CAPA)DispatchThread)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: create thread err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }

    INVOKE TempServices.MakeReady(IN (CAPA)DispatchThread)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: make ready err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }
};
```

```
OPERATION ProcessTrack(IN id_type trackid)
/*
* Handles the processing of a new track that has entered the system.  New
*  tracks are identified, and if , handed off to the threat handler.
* Since we don't want to lose this activity if a node fails, this activity
*  is registerd with the 'TrackDBMgr'.
*
* Invoked by: PlotCorrelator.CorrelateFrame
*/
{
    int          abort_count;
    u_int        size;
    boolean      abort_flag;
    status_msg_t        stat;
    track_type          track_report;
    trackentry_type     tracktype;

    abort_count = 0;

begindl:
    abort_flag = FALSE;
    DEADLINE(6000, 300, 20, 0, 0, 8000, 0) {

        INVOKE LSM.GetNodeID(OUT LogicalNodeID)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrackDB.GetTrackInfo(IN (id_type)trackid,
                                    OUT (track_type)track_report)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrkID.IDTrack(IN (id_type)trackid,
                             IN (track_type)track_report,
                             OUT (trackentry_type)tracktype)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrackDB.UpdateTrackType(IN (id_type)trackid,
                                       IN (trackentry_type)tracktype)
        ON FAILURE {
            ERROR_HALT();
        }
    }
    ON ABORT {
        abort_flag = TRUE;
        abort_count = (abort_count + 1);
    }
```

```
    if (abort_flag) {
        sprintf(stat.buffer, "Handler: ProcessTrack ABORT - FIXME %d",
                abort_count);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        /* handle this somehow! */
        if (abort_count < 4) goto begindl;
    }

    if (Hostile(tracktype)) {
        /* if the track is , process the threat */
        INVOKE TrkHandler.ProcessThreat(IN (id_type)trackid)
        ON FAILURE {
        }
    }
};
```

```
OPERATION ProcessThreat(IN id_type trackid)
/*
* Determines the correct reaction to a threat that for some reason needs to
*  be reconsidered.  This may happen when the threat is first identified, or
*  when a previous intercept attempt has failed.  This operation directs the
*  process of 'ThreatAssessment', 'WeaponsAssignment', and 'WeaponsLaunching'.
* Once a weapon has been launched, its control is passed to a dedicated
*  thread.
*/
{
    int          i,
                 size,
                 abort_count,    /* indicates how many times DL aborted */
                 threat_value;   /* perceived threat of track */
    short        site_index;     /* source of resource */
    boolean      unpaired,       /* TRUE while threat is unpaired */
                 pairable,       /* TRUE while there are options available for
                                  *  pairing the threat */
                 abort_flag;     /* indicates if DL aborted on this try */
    int_options_t favored,       /* weapons actually allocated */
                   assigned,     /* weapons actually allocated */
                   int_options;  /* intercept options to choose from */
    pairing_info_t pairings;     /* structure containing actual pairings */
    id_type      inter_track;    /* trackid of interceptor */
    track_type   inter_report,   /* dummy report entry for creating
                                  *  interceptor track */
                 track_report;   /* dummy report entry for getting
                                  *   track position */
    pos_type     track_pos,      /* coordinates of  track */
                 site_coord;     /* coordinates of site */
    status_msg_t stat;           /* buffer for status messages from Alpha */

    /* for now the decision of what type of resource to allocate *
     *  will be alternating between missiles and AI's */
    statictoggle = 0;
    abort_count = 0;

begindl:
    abort_flag = FALSE;
    DEADLINE(6000, 900, 40, 0, 0, 8000, 0) {
        INVOKE TrackDB.GetTrackInfo(IN (id_type)trackid,
                                    OUT (track_type)track_report);

        track_pos.x = track_report.pos[LASTH(track_report.currpos)].x;
        track_pos.y = track_report.pos[LASTH(track_report.currpos)].y;
        track_pos.z = track_report.pos[LASTH(track_report.currpos)].z;

        INVOKE WeaponsMgr.InquirePairings(IN (id_type)trackid,
                                          OUT (pairing_info_t)pairings);

        pairable = TRUE;
```

```
        /* later on, when we want to get real sophisticated, we may
         * want to check the pairings that are left to see if they
         * are adequite. */

        unpaired = ((pairings.num_paired == 0) ? TRUE : FALSE);

#ifdef DEBUG
        sprintf(stat.buffer, "pairable=%d, unpaired=%d", pairable, unpaired);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
#endif DEBUG

        while (pairable && unpaired) {
            /* assess the threat potential of the track */
            INVOKE ThrtAssess.AssessThreat(IN (id_type)trackid,
                                           IN (track_type)track_report,
                                           OUT (int)threat_value);

            /* get a list of intercept options available to us */
#ifdef DEBUG
            sprintf(stat.buffer, "PROC Threat: Ready to GetInterceptOpt");
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif DEBUG

            INVOKE WeaponsMgr.GetInterceptOptions(IN (id_type)trackid,
                                        IN (pos_type)track_pos,
                                        OUT (int_options_t)int_options);

#ifdef DEBUG
            sprintf(stat.buffer, "TrkHnd: Type %hd, #%hd, Type %hd, #%hd",
                    int_options.ai.intercepts[0].resource_type,
                    int_options.ai.num_intercepts,
                    int_options.gm.intercepts[0].resource_type,
                    int_options.gm.num_intercepts);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif DEBUG

            /* decide which interceptors are the best choices, i.e.,
             *  simulate c3 operations here. For right now just set
             *  'n_intercept' to 1, indicating we'll settle for the
             *  first choice returned from 'WeaponManager' */
            if ((int_options.ai.num_intercepts > 0) ||
                (int_options.gm.num_intercepts > 0)) {
```

```
#ifdef DEBUG
                sprintf(stat.buffer, "PROC Threat: Allocating Weapons");
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
#endif DEBUG

                /* allocate the weapon type */
                /* assess the response.  if we got what we asked for,
                 *  launch the requested weapons at the target track;
                 *  otherwise just release the allocated weapons and
                 *  reassess threat, options for interception, etc. */
                if ((toggle++) % 2) {
                    /* chose an interceptor (manned) */
                    int_options.ai.num_intercepts = 1;
                    int_options.ai.intercepts[0].resource_count = 1;
                    int_options.gm.num_intercepts = 0;
                    int_options.gm.intercepts[0].resource_count = 0;
                    INVOKE WeaponsMgr.AllocateWeapons(IN (int_options_t)
                                                      int_options,
                                                      OUT (int_options_t)
                                                      assigned);

                    favored.ai.num_intercepts = 1;
                    favored.ai.intercepts[0] = int_options.ai.intercepts[0];
                    favored.gm.num_intercepts = 0;
                }
                else {
                    int_options.ai.num_intercepts = 0;
                    int_options.ai.intercepts[0].resource_count = 0;
                    int_options.gm.num_intercepts = 1;
                    int_options.gm.intercepts[0].resource_count = 1;
                    INVOKE WeaponsMgr.AllocateWeapons(IN (int_options_t)
                                                      int_options,
                                                      OUT (int_options_t)
                                                      assigned);
                    favored.gm.num_intercepts = 1;
                    favored.gm.intercepts[0] = int_options.gm.intercepts[0];
                    favored.ai.num_intercepts = 0;
                }

                if (SameWeapons(&favored, &assigned)) {
                    for (i = 0; (i < assigned.ai.num_intercepts); i++) {
                        /* launch the assigned weapon at the target *
                         *  first get a specific controller object */
                        site_index = favored.ai.intercepts[i].site_index;
```

```
                    /* NOTE: the ID of the friendly is yet to be
                     *  determined, so NIL is passed in */
                    UNCHECKED INVOKE DSM.Request(
                                               IN (activity_t)LAUNCH_AI,
                                               IN (short)site_index,
                                               IN (id_type)NIL,
                                               IN (id_type)trackid,
                                               IN (short)INITIAL_LAUNCH,
                                               OUT (CAPA)Launch_AI_Thread)
                    ON FAILURE {
                        sprintf(stat.buffer, "LAUNCH bad %d", STATUS);
                        size = (STAT_MSG_HDR + strlen(stat.buffer));
                        UNCHECKED INVOKE Comm.SendMsg(
                                               IN (msg_t)MSG_ALPHA_STAT,
                                               IN (u_int)size,
                                               IN (status_msg_t)stat);
                    }
                }

                for (i = 0; (i < assigned.gm.num_intercepts); i++) {
                    site_index = favored.gm.intercepts[i].site_index;

                    /* NOTE: the ID of the friendly is yet to be
                     *  determined, so NIL is passed in */
                    UNCHECKED INVOKE DSM.Request(
                                               IN (activity_t)LAUNCH_GM,
                                               IN (short)site_index,
                                               IN (id_type)NIL,
                                               IN (id_type)trackid,
                                               IN (short)INITIAL_LAUNCH,
                                               OUT (CAPA)Launch_GM_Thread)
                    ON FAILURE {
#ifdef DEBUG
                        sprintf(stat.buffer, "LAUNCH GM bad %d", STATUS);
                        size = (STAT_MSG_HDR + strlen(stat.buffer));
                        UNCHECKED INVOKE Comm.SendMsg(
                                               IN (msg_t)MSG_ALPHA_STAT,
                                               IN (u_int)size,
                                               IN (status_msg_t)stat);
#endif DEBUG
                    }
                }
                unpaired = FALSE;
            }
            else {
                /* didn't get the weapon we wanted, so release the weapons
                 *  just allocated and try again */
                INVOKE WeaponsMgr.DeallocateWeapons(
                                              IN (int_options_t)assigned);
#ifdef DEBUG
                sprintf(stat.buffer,
                        "Nothing launched, not same weapons");
                size = (STAT_MSG_HDR + strlen(stat.buffer));
```

```
                    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                                  IN (u_int)size,
                                                  IN (status_msg_t)stat);
#endif DEBUG
                    unpaired = TRUE;
                }
            }
            else {
                /* no intercept option available */
                pairable = FALSE;
            }
        }

        if (unpaired == FALSE) {
            /* COMMIT SUICIDE - DO NOT RETURN */
            /* NOTE: check on what clean up needs to be done here */
        }
        /* NOTE: we now know there is no interceptor paired with the track */

        if (pairable == FALSE) {
            /* assuming infinite resources we should not see this case now.
             * later on probably alert operator who may be able to take
             *  action to reassign 'int_options' or make other options
             *  available. */
             /*** TBD ***/
        }
    }
    ON ABORT {
        abort_flag = TRUE;
        abort_count = (abort_count + 1);
    }

    if (abort_flag) {
        sprintf(stat.buffer, "Handler: ProcessThreat ABORT - FIXME %d.",
                abort_count);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        /* handle this somehow! */
        if (abort_count < 4) {
            goto begindl;
        }
    }
};
```

```
OPERATION Dispatch()
/*
* This is the entry point for a captive thread which actually dispatches
*  the DSM requests for new tracks.
*/
{
    status_msg_t        stat;
    u_int               size;
    boolean             abort_flag = FALSE;
    id_type             next_track;
    CAPA                track_thread;

#ifdef DEBUG
    sprintf(stat.buffer, "Handler.Dispatch Ready");
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg_t)stat);
#endif DEBUG

    while (TRUE) {
        /* wait for something to be put into queue */
        INVOKE TrackQSem.PSem()
        ON FAILURE {
            sprintf(stat.buffer, "Handler.Dispatch Queue Fail R=%d S=%d",
                    RESULT, STATUS);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }

        DEADLINE(6000, 800, 100, 0, 0, 9000, 10) {
            /* lock the new track queue */
            INVOKE TrackLockSem.PSem()
            ON FAILURE {
                sprintf(stat.buffer,
                        "Handler.EnterTrack: Lock fail R=%d S=%d",
                        RESULT, STATUS);
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

            /* remove track from head of queue */
            trackq_head = ((trackq_head + 1) % TRACKQ_SIZE);
            next_track = trackq[trackq_head];
```

```
            /* unlock the new track queue */
            INVOKE TrackLockSem.VSem()
            ON FAILURE {
                sprintf(stat.buffer,  "Handler.EnterTrack:  Unlock  fail  R=%d
S=%d",
                        RESULT, STATUS);



                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

            /* ask DSM for a process track activity */
            UNCHECKED INVOKE DSM.Request(IN (activity_t)PROCESS_TRACK,
                                         IN (id_type)next_track,
                                         OUT (CAPA)track_thread)
            ON FAILURE {
                sprintf(stat.buffer, "Handler.Dispatch: DSM fail %d %d",
                        RESULT, STATUS);
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

#ifdef RBKBUG
            sprintf(stat.buffer, "TrkHandler.Dispatch: dispatched %d",
                    next_track);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif RBKBUG
        }
        ON ABORT {
            abort_flag = TRUE;
        }

        if (abort_flag) {
            abort_flag = FALSE;
            sprintf(stat.buffer, "Handler.Dispatch: DL ABORT - FIXME");
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }
    }
};
```

```
OPERATION EnterTrack(IN id_type trackid)
/*
* Enters a new track in the queue of new tracks.  The dispatch thread will
*  actually ask the DSM for a 'ProcessTrack' activity on the least-loaded
*  node.
*/
{
    status_msg_t        stat;
    u_int               size;

#ifdef RBKBUG
    sprintf(stat.buffer, "TrkHandler.EnterTrack: entering track %d",
            trackid);
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg_t)stat);
#endif RBKBUG

    /* lock the new track queue */
    INVOKE TrackLockSem.PSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Lock fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }

    /* check for queue full */
    if (trackq_head == trackq_tail) {
        /* queue is now full (or EXTREMELY close since we do synchronize),
         *  signal error to let people know there is a problem.  This
         *  is actually an early warning, but we don't want this queue
         *  getting anywhere near full. */
        sprintf(stat.buffer, "Handler.EnterTrack: Queue FULL - FIXME");
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        INVOKE TrackLockSem.VSem()
        ON FAILURE {
            sprintf(stat.buffer, "Handler.EnterTrack: Unlock fail R=%d S=%d",
                    RESULT, STATUS);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }
        RETURN FAILURE;
    }
```

```
    /* enter new track at tail of queue */
    trackq[trackq_tail] = trackid;

    /* advance tail pointer */
    trackq_tail = ((trackq_tail + 1) % TRACKQ_SIZE);

    /* unlock the new track queue */
    INVOKE TrackLockSem.VSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Unlock fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }

    /* indicate that there is another item in the queue */
    INVOKE TrackQSem.VSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Queue Fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }
};
```

```
/******************\
*
* LOCAL PROCEDURES
*
\******************/

int     SameWeapons(requested, assigned)
    int_options_t       *requested;
    int_options_t       *assigned;
/*
* Return TRUE iff requested weapons match assigned weapons.
* For now a crude check for the counts being equal is used.
* Later on, each element of the requested list will have to be
*  checked against the assigned list.
*/
{
    if ((requested->ai.num_intercepts == assigned->ai.num_intercepts) &&
        (requested->gm.num_intercepts == assigned->gm.num_intercepts)) {
        return (TRUE);
    }
    else {
        return (FALSE);
    }
};


int     Hostile(trktype)
    trackentry_type     trktype;
/*
* Return a TRUE if the given track is a  type
*/
{
    if ((trktype == HOST_BOMBER) || (trktype == HOST_MISSILE) ||
        (trktype == HOST_JAMMER)) {
        return(TRUE);
    }
    else {
        return(FALSE);
    }
};

}       /* end of object */
```