Abstract

This paper describes MITRE’s Evolvable Real-Time Command, Control, and Communications (C3) systems initiative that attempts to develop an approach that would enable current real-time systems to evolve into the systems of the future. In particular, this article describes the infrastructure requirements that we have developed. We first provide an overview of the current real-time C3 systems and describe the systems of the future. Next, we describe some candidate architectures that we have examined for future systems. Then a detailed discussion of the requirements for the infrastructure are given. The main focus is on operating systems, data management systems, and communication systems requirements. The discussion is based on the candidate architectures that we have examined. The project has chosen Airborne Warning and Control System (AWACS) as an example to test out the concepts and architectures to be developed.

1. Introduction

Between now and the early part of the next century, significant portions of today’s real-time Command, Control, and Communication (C3) systems will become functionally inadequate and logistically unsupportable. Furthermore, due to the continuing budget reductions, new developments of next-generation real-time C3 systems may not be possible. Therefore, current real-time C3 systems need to become easier, faster, and less costly to upgrade in capability and easier to support. What is needed is an approach to evolve current real-time C3 systems into the extensible systems required for the future.

MITRE’s Evolvable Real-Time C3 project funded under the Air Force Mission Oriented Investigation and Experimentation (MOIE) program attempts to develop an approach that would enable current real-time systems to evolve into the systems of the future. The candidate evolution approach is to leverage off near-term system upgrade and/or P3I (Pre Planned Product Improvement) activity to put a new architecture framework in place. The emphasis is on transitioning to open architectures which are modular and free from proprietary or unnecessarily complex software designs. The open framework can also accommodate new upgrades more easily. Availability of an infrastructure to support a suitable software architecture is key for this approach to succeed. The investment plan would continue incremental transition of current systems into more flexible systems. The extensible system architecture would ultimately replace the current hardware and software architecture.

In order to provide an evolution path for real-time C3 systems, one needs to understand the requirements of current real-time C3 systems and the approach taken to design such systems. The project has chosen AWACS (Airborne Warning and Control System) as an example to test out the concepts and architectures to be developed. Currently, its centralized design is a closed architecture with monolithic custom software. It does not take advantage of state-of-the-art hardware. Processing upgrades to the system is time-consuming and expensive. The project has chosen Multi-Sensor Integration (MSI) function as a starting point for transitioning AWACS to an open architecture. This is because MSI supports important combat identification capabilities. Also, MSI function’s impact on data and display processing provides a thorough test of the concept. The technical challenge is to demonstrate the applicability of open software technology to AWACS and other real-time C3 systems. The successful execution of this project would facilitate the transition to open systems.

The ultimate objective is to provide a direction for designing and developing next generation real-time C3 systems. However, several steps need to be carried out before such systems are developed. This initiative will identify these steps and determine the issues. The major goals of this initiative are the following:

- Identify the software infrastructure requirements for extensible real-time C3 systems.
- Demonstrate the feasibility of the evolution.

That is, the steps involved in evolving a legacy system to a next-generation real-time C3 system need to be determined.

- Identify the migration path for legacy systems.
- Demonstrate the feasibility of the evolution.

That is, design the intermediate systems as well as the
final system and carry out proof-of-concept prototype implementations.

Additional goals of the initiative are to influence the industry and standards so that this effort will have an impact on the components necessary to build real-time C3 systems. This will in turn influence future government acquisitions. Our initiative has tracked the current state-of-the-art in real-time systems through participation in seminars, conferences, and standards groups–particularly in operating systems, database management systems, and communication systems.

The organization of this paper is as follows. In Section 2 we describe the approach we are taking to designing evolvable real-time systems. The candidate architectures we have investigated are discussed in Section 3. These are a centralized architecture, a message-passing distributed architecture, and a distributed object management client-server architecture. For each architecture, we provide an overview of the design, system evolution, fault tolerance issues, advantages, disadvantages, availability of Commercial-Off-The-Shelf (COTS) components and integration, cost, and risks. The paper is concluded in Section 4.

2. Project Description

2.1 Existing Systems

C3 Department of Defense (DOD) computer systems arose from the need to coordinate the gathering, analysis, and reporting of information on combat theater situations quickly and reliably. These systems have been around for many decades. The legacy DOD computer systems have characteristics that are very similar to commercial systems designed in the same time period.¹ The main difference is that the DOD systems tend to remain in use many years longer than their commercial counterparts. As a result, DOD systems face the "culture shock" of new design paradigms less often, but more overwhelming each time, than commercial systems.

These older systems faced higher costs for computing hardware than for software or other equipment. Note that software costs now typically exceed hardware costs, especially when the entire life cycle is considered. Peripheral costs can frequently exceed Central Processing Unit (CPU) and memory costs in a system. Also, the growth of the legacy DOD systems focused on supporting the changes then viewed as technically possible within 5-10 years after delivery. At the time, advancements in technology could be expected to handle growth in processor speed and memory density by about 2-5 times. That is, such growths in both of these areas could be accommodated with the initially-delivered hardware and software architecture. Support for architectural growth, such as major additions to software functionality, was generally not included in the system design since it was not expected to occur within the planned life cycle of the project.

Software architectural integrity, flexibility, and complexity have been traded frequently for gains in operational performance during enhancement efforts. This was done to save costs in the short term by minimizing software changes and hardware growth.

The legacy DOD systems do not take advantage of the state-of-the-art in hardware and software. Over the last few years software architecture design has advanced due to the available processing power, standards, and integrated COTS/GOTS² software products. These issues have to be taken into consideration when evolving legacy systems. For example, the operating systems, data management systems and network/communication systems technologies have advanced a great deal. Open architectures and object-oriented approaches are becoming common practice. Multiprocessor and parallel processing computers are now commercially available.

Real-Time systems technology has also advanced to a point where commercial products and standards are now available. As a result, instead of implementing a cyclic executive, one could choose among the variety of real-time operating systems which are now commercially available. Real-Time data management systems provide all of the advantages of data management systems and yet execute query and update requests so that the timing constraints are met. Real-Time communication systems meet the timing constraints during the communication between different processes or subsystems.

Applications are typically distributed among computing nodes. The legacy systems forced a centralized architecture to manage the distributed applications. Over the past decade, distributed systems technology has advanced rapidly. Several distributed operating systems and distributed data management systems need to take the developments into consideration for distributed applications. More recently, the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) has specifications for heterogeneous applications to interoperate. While CORBA technology still has a long way to go before becoming common practice, due to its increasing popularity, it should be examined when designing new systems.

2.2 Approach to Building New Systems

The major goals of the initiative include determining the software infrastructure requirements and to identify the migration path for legacy systems. The infrastructure is a collection of all non-application specific software services. This infrastructure provides the software backplane for applications and insulates application software from hardware. Ideally, we want to use COTS products for the infrastructure. Figure 2-1 illustrates the infrastructure.

¹ By legacy systems we mean those developed in the 70's or earlier.

² GOTS stands for Government-Off-The-Shelf.
The services provided by the infrastructure include real-time operating systems services such as memory management and real-time scheduling, real-time communication services such as interprocess and intraprocess communications, and real-time data management services such as data sharing, querying, updating, transaction management, and enforcing integrity and timing constraints. The infrastructure also provides the mechanisms for interaction between the software components. All of the services must provide an integrated priority scheme and performance predictability.

The target C3 extensible architecture is illustrated in Figure 2-2. Ideally, all of the application components should be hosted on the infrastructure. The application components for a system such as AWACS will include MSI, display, weapons, surveillance and tracking, and communication. The infrastructure provides the means for the application subsystems to access and share the database as well as to communicate with each other. Implementing such a system will mean re-architecting the entire AWACS system. This is not feasible with the current budget. Therefore, our approach is to extract certain application subsystems and host them on the infrastructure while the other subsystems remain within the legacy environment. An example of the intermediate architecture is illustrated in Figure 2-3 where the MSI subsystem is hosted on the infrastructure.

Since the surveillance function is the one to be hosted onto the infrastructure, let us consider processing of the surveillance and identification subsystem after MSI algorithms replace current tracking algorithms, and functions that are not related to target state estimation are moved to other subsystems. The major inputs and outputs of the subsystem remain the same. MSI algorithms replace current association, correlation, and smoothing functions. MSI algorithms improve tracking because they use all the sensor data to estimate the target’s state, instead of choosing only the highest priority datum, which is currently done. The functionality of the new surveillance subsystem is being reorganized to maintain functionality of the original system. Some functional components are better suited for other subsystems, and some others are no longer needed. Some components have been moved to a different function in the surveillance subsystem.

Surveillance initialization updates the position data every minor cycle. The MSI tracking algorithms read-in sensor reports and track records, update the track state estimates, then write the updated track records and sensor reports back to the database. Sensor reports are flagged if they have correlated with a track. Sensor reports not correlated with tracks are used to automatically initiate tracks in operator specified regions. The track monitor and identification function determines if tracks are within flight corridors, exclusion areas, and safe passage areas, and that the target identification codes are consistent with those expected. This function updates tracks with identification information.

Our project will be carried out over a period of three years. During the first year of the initiative, we have analyzed C3 system requirements with special emphasis on AWACS, prototyped MSI applications component, developed functional models of the system for hypothesized architectures, deduced the infrastructure requirements, and assessed the research and industry trends.

Figure 2-3. Example AWACS MSI Within An Extensible Infrastructure

For a discussion of surveillance we refer to [WALT90].
in real-time systems. During the second year, we will acquire selected products and technologies and possibly extend and enhance them to meet C3 real-time requirements. We will also integrate the components to provide real-time embedded infrastructure. The selected functional model will be validated using the MSI application module and subsequently the infrastructure architecture will be evaluated. During the third year, we will have the MSI application hosted on the integrated infrastructure and conduct a series of experiments to validate the infrastructure support of the application requirements. We will also establish the feasibility of the transition approach.

3. Architecting Robust Real-Time System

3.1 Centralized Database Architecture

3.1.1 Design Overview

3.1.1.1 Essential Features

The central database computer system architecture for C3 systems consists of subsystem functional units operating on separate computer nodes each with their own local data manager and database. The local databases are partial replicas of a central database. The central database is maintained on a central node which contains all data for inter-subsystem communication. A central data manager periodically receives inputs from each subsystem node, and periodically broadcasts the central database out to each subsystem node. Since all the subsystems report their new or modified data each cycle, and the subsystems receive a full copy of the central database each cycle, all data is communicated between subsystems in a predictable, reliable way. This method of periodic updates and broadcasts eliminates the problem of update propagation typical of other replicated database designs (see for example [CERI84,DATE90]).

Each subsystem need not read the entire database being broadcasted. The local data managers incorporate only those parts their applications need. The format of the broadcasted central database is known to the subsystem nodes. That is, the explicit interface of data communications between nodes is known by all nodes. The inputs to the central database from the subsystem nodes are performed according to a schedule known by the central data manager and the local data managers, and supported by real-time communications.

3.1.1.2 Overall Philosophy of Design

The basic philosophy of the central database architecture is for the inter-subsystem communication and control to be periodic and predictable. At this high level of interaction, communication and the associated processing is very predictable, while data latency and consistency are clearly defined. The subsystems may interact with external resources in a manner most suitable to those external interfaces, but at the inter-subsystem level data need only be transferred on a periodic basis. This two-tier approach is appropriate for systems which can be divided into subsystems which interact on a periodic basis, which is true of existing C3 systems, and hence simplifies the transition from the existing to the future system. The length of the interval can be adjusted to fit the application timing requirements.

This design establishes the concept of predictable real-time system behavior at different design levels. By establishing predictable communication and processing at the highest level of the system, a solid foundation is created for the lower levels. Real-time requirements can be clearly defined at each level, which will promote software development, debug, integration, and testing. By comparison, a system whose real-time requirements are not modular may be impossible to fully guarantee.

C3 surveillance systems have traditionally operated on a cyclic processing basis. The processing requirements of these systems are in terms of processing intervals. Interfacing established legacy systems, such as the AWACS, which are themselves cyclic, is easier if the future architecture itself can operate on a cyclic basis.

3.1.1.3 System Level Description

The central data manager periodically broadcasts the entire database once every processing interval. Also at every interval, the central data manager receives data from each individual subsystem, if there are data changes, according to a schedule known by all data managers. There are no specific queries to the central data manager, which together with the database act as a central clearinghouse only.

The local data manager has functionally the same interface in all subsystems. The local data manager must coordinate the transfer of data between itself and the central data manager. The local data manager will respond to queries from the local application. A subsystem interacts with external systems via a communication network that is separate from the inter-subsystem communication. The inter-subsystem communication network is dedicated so it can remain periodic and predictable, while clearly defining its real-time requirements. This approach maintains a modularity of predictable real-time behavior. The predictability of the inter-subsystem communication forms a foundation on which a real-time system can be built. The central database architecture only addresses inter-subsystem control and communication and does not restrict the design of a subsystem besides its having to interact with the central data manager on a regular basis.

Figure 3-1 below illustrates the basic processing nodes of a typical C3 system implemented in a central database architecture. The sensory input, communication, display interface, surveillance and weapons control subsystems all
receive the broadcast of the central database, and update their data to the central database every processing period. Each subsystem has a data manager and a local network manager. The local network managers are dedicated to the inter-subsystem communication and control. The sensory input, communication, and display interface all have an external network manager to interface systems on the external network.

![Diagram of C3 System Implemented in a Central Database Architecture](image)

**Figure 3-I. C3 System Implemented in a Central Database Architecture**

### 3.1.1.4 Fault Tolerance

Although there may be other alternatives, one solution to fault tolerance is to use hot and cold backup nodes that are ready to take on the responsibilities of any other node. If one of the subsystem processing nodes goes down, it can be restarted on another node within a small number of cycles. When the central data manager does not receive the inputs from a faulty node it detects a problem. A specified protocol would determine positively if the node is down. A spare node can be brought up to replace the faulty node at the request of the central data manager. Since the entire central database is broadcast regularly, the data local to the original faulty node will be recovered. The spare nodes must have the ability to assume the role of any other node, including the central node.

If the central data manager goes down, it also can be restarted on another node in a few cycles, or a hot spare for the central data manager can be part of the design. When the local subsystem nodes get no acknowledgment of receipt of their inputs from the central data manager, they conspire to bring up the central data manager on a spare node once they agree that the central node is down. A spare node is designated as the new central node. Since the central database consists of contributions from the local nodes, the local nodes can send their inputs to the newly designated central node via their periodic data upload and have the central database recovered. A spare processing node is logically identical to either a local or central node and can operate as either.

### 3.1.2 Support for System Evolution

System evolution from the legacy system to a central database architecture will be greatly facilitated by the fact that the central database architecture can be cyclic. Functional components will be migrated from the existing processor to processing nodes of the central database architecture while working within their existing timing constraints. All the communications between the nodes are going through the central database node, which can be done cyclically, priority based, or application specific way. A node dedicated to interfacing the existing system will receive the broadcast central database every processing interval and write the appropriate data to the appropriate location in the existing system memory.

As new functionality is required by the system, it is accommodated by adding new subsystem nodes to the system. The parts of the system that must be changed when adding a new component are: (1) the schedule that coordinates the inputs to the central data manager from the local nodes, (2) a map of local node data to reconstruct the central or local databases in case of a fault, and (3) the application must be changed to use the new functionality. Items one and two above could be automatically updated when a new component is added, providing "plug and play" capability. Modifications to a subsystem impacts the overall system in the same way as adding a new component. No changes to other nodes software will be needed, only the communication scheduling have to be modified to accommodate application modifications.

### 3.1.3 Infrastructure Requirements to Support the Central Database Architecture

It is a requirement that the operating system of each node have the real-time features which ensure that the reading of the central database broadcast, and the scheduled upload of data to the central database occur at the scheduled times. For time driven real-time communication local data manager on each machine must gain control of the processor at the predetermined times within some latency limits. For that he operating systems on various nodes must stay time synchronized with each other. Alternatively, the event based real-time communication (for example, control message based) can be easily implemented at some expense on the performance. The network managers on various nodes must support the broadcast of the central database and the scheduled upload of data to the central database without deadlock, or live lock problems.

**Advantages:** Real-time requirements can be clearly defined at each design level. This predictable real-time system behavior specified at different design levels, facilitates software development, debugging, integration, and requirements testing. Predictable communication and processing at the highest level of the system forms a solid foundation for the lower levels to build upon. There is
also nothing precluding the hierarchical application of this architecture to the subsystems, providing a step-wise approach to realizing the real-time requirements. Because of the simple modular design of the central database architecture, subsystem software development can take place with only two interfaces, the central data manager and the systems on the external network. This may make for easier software debug, diagnostics, and testing (which is always underestimated). Functional additions to the overall system have minimal impact to the existing system because new processing nodes are integrated in a “plug and play” manner (changes to local database map, and upload schedule are automatic).

The broadcast and scheduled upload of data provides a very predictable communication and processing load on the system. This predictability is necessary for a reliable real-time system. Real-time requirements at the inter-subsystem level are clearly defined by the timetables for broadcast and upload. The communication channels between subsystems can be designed with the appropriate bandwidth and can be expected to deliver the required inter-subsystem data. These inter-subsystem data communication channels are dedicated. Subsystems which interface the external world utilize separate networks and network managers for their external communications, thus greatly simplifying the real-time communication requirements for both networks.

The central database architecture deals with the problem of data latency between subsystems in a very simple way. Since all new local data is uploaded to the central database every cycle, and all central data is broadcast to the subsystems every cycle, the data latency problem is reduced.

The periodic processing nature of the central database architecture is consistent with existing C3 systems that have traditionally operated on cyclic processing intervals. The subsystems of typical C3 systems expect to receive and send their data on a cyclic basis. The display expects to receive the tracks and sensor reports periodically. The surveillance function expects sensor reports and user commands periodically, then periodically updates and predicts the tracks for the next interval. Communications periodically reads the local tracks and sends them over the tactical data links, and also periodically reports external tracks and data it has received. Weapons control periodically receives user commands, reads the track information it needs and sends radio control track messages to communications. Requirements for current C3 systems call for the periodic transfer of data between functional units and the central database architecture does that in a reliable way.

The central database architecture can be constructed with heterogeneous hardware components. This is an advantage in the event that one or more of the COTS products are no longer supported at some point in time.

It is easy to build in fault tolerance to the central database architecture. Spare generic nodes which contain the central and local data manager software are already connected to the inter-subsystem communication network. When a fault occurs in any node, a spare is ready to assume its role in a few processing cycles.

On the whole, the central database architecture is the most supportive of migrations and improvements to individual subsystems among the three architectures presented here.

Disadvantages: The major disadvantage is potential overload. If there is not enough slack time in the cycle to broadcast and receive all the data, the system will become overloaded, as the current AWACS does. We need to explore further what methodology, if any, might be effective within the cyclic nature of this design to handle potential overload.

3.1.5 Availability and Integration

The hardware and software components needed to put together a central database architecture are almost all currently available. The real-time operating systems are fast becoming commercially available. Network managers are currently available, but it is not clear which is most suitable since the requirements for the real-time communication have not been defined. Two commercial networks under consideration are the Asynchronous Transfer Mode (ATM) and the Fiber Distributed Data Interface (FDDI) token ring. The central and local data managers will probably not be available as COTS products. In summary, the central architecture could be put together with much of it made up of COTS components within the next few years.

The central architecture may be implemented on either loosely coupled heterogeneous processing nodes, on a tightly coupled symmetric multi-processor, or a massively parallel computer system. The most general hardware model is one which is a combination of loosely coupled separate machines, some of which are symmetric multi-processors and massively parallel processors.

COTS software components include the real-time operating system, real-time network manager, real-time data manager, and external network manager operating on each node. All these components could use the same software on each node to provide interchangeability of nodes necessary for fault recovery methods.

The central architecture consists of interchangeable processing nodes. The nodes may be configured to different hardware types, as well as software system components most suitable for the application subsystem. The interprocess communication and data managers ensure system integration. The individual processing nodes are coordinated by the data managers.

Along with the simplicity of design, the cost of implementing a central database architecture is low because of the availability of COTS components, and the ability of much of the system to be reliably debugged and tested as subsystems independent of the completed system; thereby reducing software development costs. Also, because the existing C3 systems are periodic in their processing, the evolution to new central database
architecture (which is also periodic) will reduce complexity that in turn will reduce costs. The fact that the central and local data managers may need to be written and maintained as part of the application software will increase costs initially.

The central database architecture is relatively low risk. The periodic broadcast and upload of data is consistent with existing C3 systems, easy to make an evolvable transition to, and simple to create, debug, and maintain. The overload issue needs to be evaluated further and the mechanisms explored to reduce this risk.

3.2 Message-Based Distributed Architecture

3.2.1 Design Overview

3.2.1.1 Essential Features
Following the continuing trend in the computer industry to connect several computing nodes either in the distributed environment consisting of several computers connected via communication network, or in Massive Parallel Processing (MPP) environment, we directed our effort toward developing an evolvable parallel/distributed environment for C3 systems. As part of our assumption to support evolvability and allow gradual transition of existing C3 systems like AWACS, to proposed architectures that can be built using COTS components, we map individual subsystems (functional components) onto individual computers (or a group of computers). We then focus on the kinds of data flows, the characteristics of the data flows between the individual subsystems, and the connection media between the computers on which the individual subsystems are mapped onto. The organization of subsystems on individual nodes, including data organization and interfaces for communication between various parts of the C3 systems, play an important role in the overall design in order to support evolvability and gradual transition. The most critical aspect of the design is its ability to support timing characteristics of both existing and future system requirements.

The architecture presented here is the least common denominator for the distributed environment for C3 systems. This message passing architecture separates the AOCP software along functional lines and maps individual functional modules (that we call subsystems) onto nodes of the distributed system. For simplicity, we can consider that each individual subsystem will be mapped onto a separate node (note that this is not required). Each subsystem has its own individual database of all required data, and different subsystems share/exchange data via messages (see Figure 3-2).

3.2.1.2 Overall Philosophy of Design

This architecture is the first one that comes to mind when one thinks of transitioning an existing C3 system to a distributed environment. We do not require any additional constraints on the existing software design of C3 systems and rely on the improved hardware and software (middleware) of the COTS products to ensure evolvability, cheaper maintenance, extensibility and fault-tolerance. This architecture effectively shifts the existing problem of real-time scheduling of subsystems on a single processor to a real-time communication scheduling of the messages on the underlying network. This assumes that the real-time communication protocol with its underlying scheduler will solve the problem. Since we need end-to-end requirements to be satisfied, both real-time communication services and real-time Operating System (OS) services are needed, and the interface between them becomes a major issue. As state of the art in communication scheduling improves, we can upgrade real-time communication to improve timing performance and service support.

Figure 3-2. Distributed Database Architecture

3.2.1.3 System Level Description

The main idea behind this architecture is to map the existing C3 systems onto the most widely used computer system class of today – distributed network of COTS computers. No constraints are added to the existing design. This allows the maximum flexibility for the architecture but adds more requirements for the infrastructure to support the existing C3 systems. From a preliminary analysis, it seems that this architecture would be the most complex one to transition an existing C3 system among the three proposed architectures. One of the interesting aspects of this architecture is the transition from the existing time-driven systems to the event-driven systems where the events triggering individual subsystems are message arrivals. This will ease the burden of handling deviations of message sizes, timing arrival and sending. Hence, this architecture is the most robust among the three proposed. On the other hand, it is the hardest to ensure graceful degradation in overload situations and the most difficult to recover from faults.
3.2.1.4 Fault Tolerance

A large redesign effort is needed to build fault-tolerance for this architecture. While some faults, like data not arriving, or arriving only partially, are easier to handle using event-driven methodology, others like late arrival, data consistency, as well as fault recovery are harder to do. To build these features, extensive redesign of the software architecture of the existing system, as well as internal redesign of individual subsystems may be required. Since data is distributed among individual subsystems and the only way to get to it is to send messages, the data consistency and synchronization in general becomes the most crucial problem. The large amount of network traffic has to be dedicated to solve this problem which may require more services from the real-time communication than are currently available. The distributed real-time database that tackles this problem or some of its functional parts must be developed with the support from the underlying communication mechanism. There are no non-proprietary products (note that proprietary ones may require proprietary architecture and software, and may not be compatible with COTS) on the market that can do this, and the research on this topic is in rudimentary stages which began less than five years ago.

3.2.1.5 Software Anomalies

The timing faults like missing deadlines, as well as incorrectness of the results, can propagate throughout the system. In order to prevent this, extra data may need to be sent around and additional software may be written. These are difficult problems to solve and they are currently of much research interest to several communities like parallel and distributed simulation, distributed databases, and real-time systems. No complete solution exists at present, even though through extensive testing and system tuning some of these problems could be solved in practice.

3.2.1.6 Anomalies

With respect to hardware failures, since the number of processors have increased and the internal network has been added, the number of places where the faults can occur would also increase. For every hardware fault, the synchronization and data consistency issue have to be addressed. Hence, the recovery time increases, which may require partial system redesign as we may not be able to recover from some faults in the required time.

The data anomalies as well as almost every fault and anomaly in the system require data synchronization and consistency problems to be solved. The solutions to these problems are very costly and runtime consuming. The lack of central data repository and centralized recovery mechanism would require more time than the current designs and implementations.

3.2.1.7 Overload

The graceful degradation issue is again difficult to solve because of lack of centralized control. We cannot simply drop a message or stop the task from finishing, but we need to notify other processors to prevent error propagation. This in turn, increases communication load even further and increases the time needed to gracefully degrade. The simplest way to handle overload would be to drop a cycle of data and synchronize the system to the previous cycle, but unfortunately this may be the time when this data is most urgently needed.

3.2.2 Support for System Evolution

Even though this architecture is scalable, every change requires a complete check of the communication methods, and verification of the underlying assumptions of real-time scheduling. Issues like synchronization, data consistency, exact timing of message delivery and sending, need to be ensured for every change. The extensibility for functional changes may require hardware changes for networking, and network interfaces to support the increased load.

Transitioning the current system to this architecture requires the complete knowledge of data requirements and data flows with timing characteristics for each subsystem. To identify all hidden dependencies between all subsystems and data anomalies is a hard and time-consuming process. Each partial transition requires the complete re-analysis of real-time communication due to the changing demands on intersystem communication.

The redesign will still support existing interoperability with other DOD systems, but all additions or modifications to the current and future interoperability interfaces require complete verification of timing constraints, especially for communication.

3.2.3 Infrastructure Requirements to Support the Message Passing Architecture

The main requirements that fall under the real-time communication area include synchronization, and data consistency. Since we need end-to-end responses, the interface issue between network scheduler and Operating System (OS) scheduler which guarantees latency, becomes more critical than for other architectures.

Operating System: The operating system should provide predictable behavior to ensure that the message generation is achieved at predetermined times, even though the time itself may be data (scenario) dependent, and be ready to receive messages to ensure that communication networking interface works correctly. This may require support for both event-driven and time-driven paradigms, as well as lower-level predictable interaction by interrupts, signal handlers, memory access protocols and other OS services.

Communication: One can expect most of the timing requirements to be imposed on the communication mechanisms. Note that at present, communication is handled via a proprietary (cyclic) executive in C3 systems such as AWACS. Next generation systems must support a great spectrum of communication loads and still guarantee performance. The main burden of graceful degradation also falls on the communication mechanisms.
Thus, most of the performance requirements directly translate into real-time communication requirements.

**Database:** The database services are unclear for this architecture at this time. The current systems do not have a separate database service. Hence, the database requirements need to be designed for individual subsystems and are not available now. Some of the features like data consistency and synchronization may be handled via a real-time database manager or an active database manager. These features could produce strong requirements which at present cannot be handled by existing real-time database managers.

### 3.2.4 Advantages and Disadvantages

The main advantage of this architecture is its flexibility. With the exception of the communication, it has weak requirements for hardware and software. The new products will not require special modification or adaptation in order to be incorporated, but timing requirements for communication need to be verified for each change.

There are some disadvantages. The real-time communication issue becomes an Achilles heel of the system which will hamper its evolvability. All the difficulties that were associated with cyclic executive and data tables are now pushed into real-time communication. To adopt this design for one C3 system may not guarantee its suitability for others. For each individual system, a substantial effort has to be invested which can only partially be leveraged between different systems.

### 3.2.5 Availability, COTS Components, and Integration

There are no distributed (nonproprietary) OSs which will provide the required service even though there are several which are currently being developed, most notable among them is OSF Mach-RT. There are several possible candidates for real-time communication protocols, like FDDI-II, but it is not clear if their throughput/performance would be sufficient, nor how well they can be integrated with real-time OS on the individual nodes.

The integration issues are most severe for this architecture among the three proposed. This is mainly due to the fact that all the burden of timing support was loaded onto the components instead of the overall design methodology. The fact that this architecture is the most flexible puts extra burden on the individual components and their integration.

### 3.2.6 Cost and Risk Assessment

In addition to most of the costs associated with the centralized architecture, the distributed architecture has additional costs for maintaining data consistency and real-time communication. The latter might even increase the maintenance cost even when COTS products are used for individual components. Other costs include those for development and transitioning.

This architecture is most risky among the three proposed since it has the hardest requirements on real-time communication and integration of real-time services, and timing performance on the nodes.

### 3.3 Distributed Object Management (Corba-like) Architecture

#### 3.3.1 Design Overview

#### 3.3.1.1 Essential Features of CORBA

In our investigation of the client-server architecture, we have considered the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA) which provides both client-server and peer-to-peer services. The proposed architecture in this section describes a Real-Time Distributed Object Management (RTDOM) system based as closely as possible on the OMG CORBA [OMG91, OMG94] model to act as the high-level transfer agent between the components of the C3 system. This will ensure that interfaces between components are defined using a standard, declarative syntax and semantics, and that these interfaces are isolated from component changes. Figure 3-3 shows the distributed processing model.

CORBA specifies a standard for a Distributed Object Management (DOM) architecture. Three major components of CORBA are the object model, the Object Request Broker (ORB) and object adapters, and the Interface Definition Language (CORBA-IDL). We discuss the essential points of each component.

The object model describes object semantics and object implementation. Object semantics describe the semantics of an object, type, requests, object creation and destruction, interfaces, operations, and attributes. Object implementation describes the execution model and the construction model. The ORB essentially enables communication between a client and a server object. As stated in [CORB92], a client is an entity that invokes an operation on the object, and the object implementation is the code and data that actually implements the object. The ORB, together with object adapters, provides all the mechanisms to find the object implementation for the request, to prepare the object implementation to receive the request, to communicate the data making up the request, to activate and deactivate objects and their implementations, and to generate and interpret object references. IDL is the language used to describe the interfaces that are called by client objects and provided by object implementations. IDL is a declarative language. Note that clients and object implementations are not written in IDL. The IDL grammar is a subset of ANSI C++ with additional constructs to support the operation invocation mechanism. An IDL binding to the C language

---

4 Note that one may not regard this architecture to be a separate architecture. One could implement the first two architectures using the Distributed Object Management approach. That is, the components of the first two architectures may be encapsulated as objects.
has been specified; other language bindings are being processed.

![Figure 3-3. Overview of Distributed Object Management via CORBA](image)

### 3.3.1.2 Overall Philosophy of Design

This design is based on the philosophy of encapsulation, the treating of each relevant component of the system as a black box which has responsibility for executing a set of actions (operations.) The inputs and outputs and syntax of each component’s operations are published; the internal implementation of each component is hidden and is only available to itself and to the DOM. Encapsulation, which came into prominence with the theory of Abstract Data Types (ADTs) is a key element of this design to promote evolvability.

A second key feature of the DOM design is that the DOM system itself contains the information about how to transmit data and requests between application components. This is a major localization of information about the system and environment within which each component is functioning. As new components are added to the system or migrated from the legacy system, no change to any other component is required to accommodate them. Using a middleware design for system knowledge forces the application layer to be concerned only with application domain knowledge and requirements, and decouples the components from system environmental dependencies. The system design issues relating to communication media and protocols, processors, operating systems, database management policies, memory management, scheduling policies, and so forth, are isolated from the application objects.

The use of published and vendor-backed standards is another important feature of this design. The Object Management Group (OMG) is a consortium of hundreds of companies representing significant and varied vendor support. Established standards with multi-vendor support enhances the evolvability of the C3 system design. It becomes much more probable that components can be moved to new platforms or to upgrades of existing platforms without major impact on their internals. Note that if the developers use non-standard features, this assurance is negated.

Nothing in the design of a DOM requires either a centralized database, a distributed one, or in fact, any database. Thus, the central database architecture in Section 3.2.1 can just as well be implemented within a DOM architecture, where the central database manager is one component object, managed by its own ORB. The local database managers are encapsulated within the new component objects, or are separate objects which have a shared-memory area with the local component that they directly support.

### 3.3.1.3 System Level Description

There will be a set of objects in the DOM, each one of which represents a component in the C3 system. One of these components will be an object encapsulating the current legacy C3 system along with any additional hardware and software which is needed to pull data from, and insert data into, the legacy C3 system. This legacy object, along with, perhaps, a Legacy object adapter, will be responsible for interfacing transparently with an existing C3 system such as AWACS. The other objects or components will initially be new functional subsystems which replace or add to the subsystems into which the current C3 system is partitioned. As further development under this architecture is pursued, finer-grain objects may be added, such as a functionally-coherent subset of a subsystem or a set of common data such as target tracks.

As the existing C3 systems have been built with cyclic executive managers which ensure processing on a periodic cycle, this places a requirement to detect and signal events (including time) on all of these hypothesized architectures. With a DOM architecture, the responsibility for coordinating with the cyclic nature of the legacy system can reside in the legacy object itself. This would signal the beginning or end of each subcycle in the legacy system. The cyclic nature could also instead be enforced by a central database as described in 3.2.1. Alternatively, there could exist a cyclic coordination object as one of the components in the new architecture.

Figure 3-4 shows the broadcast from the central database with distributed database managers at each component. Figure 3-5 shows data owned by each component with broadcast from each component and a central database for recovery. In both cases, a DOM manages the objects and the communications of requests are sent to the objects.

### 3.3.1.4 Fault Tolerance

Most of the issues of fault or anomaly detection, prevention, and repair are common to all architectures and are discussed in Section 3. Those particular to the DOM are discussed below.

ORBs and object implementations are processes and are detected to be missing when a message is sent to one
and the underlying system reports an error back to the sender (the ORB). If it is desired to detect problems earlier than on an as-needed basis, a daemon could be sent around, periodically checking on all activated objects.

The CORBA DOM has provisions to change an implementation after an object has been activated. This means that if a hardware node is lost and is re-established by using a spare processor, object implementations can be activated on that new node and all requests will now be directed to that new node. There are three CORBA object activation policies which are relevant to the minimization of the time interval that a component is down. The first policy for an object is that once its implementation is activated, it is fixed to an ORB and if it is not available (the ORB is not responding), a fault is signaled. This gives the fastest location of that object and leaves the repair to the designer. The second policy is, whenever there is an ORB already established for that object, that is the one that will be automatically used; however if an ORB does not exist, a new implementation will be activated and used. This policy is as fast as the previous one when there are no faults; it does take immediate corrective action when the object implementation or ORB goes down for any reason. The last policy looks to see what is the best ORB to use each time a request is made. This policy performed in software, does not seem well suited for hard real-time systems.

3.3.1.5 Some Issues on RT-DOM

3.3.1.5.1 Real-time DOM Design

The OMG is not yet looking at the Real-Time DOM requirements. However, we have begun an evaluation of integrating real-time systems technology with CORBA. Our current thinking on the requirements for a real-time DOM (RTDOM) are described below[KRUP94].
time remaining, objects with lower criticality values would then be accessed.

IDL must be extended to include syntax and semantics for these new concepts of timestamps, max/min execution times, deadlines, priority and criticality values, periods, exception handling for time constraint violations, and other required constructs. Real-Time extensions to programming languages such as C and C++ have already been proposed [WOLF93, ISH92] and can serve as a basis for real-time extensions to IDL.

### 3.3.1.5.2 Real-Time Implications

The location information is in the object reference, which in fact is handed from ORB (to ORB) to object adapter. Once an object reference is established, it can be stored in a string form as a logical reference to the object. Only when the object is activated is the logical reference turned into information which tells the DOM how to locate the object. References for permanent objects such as for the subsystem objects and the DBMS objects could be established at initial DOM activation time.

Predictability needs to be ensured. This implies that a real-time ORB, real-time object adapter and real-time IDL extensions are needed. The RT-ORB and RT-object adapter code will be designed to be as efficient as possible and fault-insensitive. The RT versions must embody the concepts of timing constraints and deadlines, execution priorities and critical data. This system must be layered onto a POSIX-compliant RT operating system and RT communication protocols. The interaction between the scheduling performed in the lower layers and that performed by the ORB must be designed so the necessary information is passed so each can perform their tasks. This will, most probably, need to be a two-way interaction. An analysis of the RT ORB code and the RT object adapter code should be done to predict execution time within these components and to determine that there will not be any unbounded behavior.

Language for data and requests must be provided in IDL to deal with time stamps, the time-decay value of data and criticality where appropriate. Also, language must be provided to describe execution time of requests as well as deadline and other RT information.

### 3.3.2 Support for System Evolution

As described above, the DOM architecture is highly modular, localizes information sources and knowledge, decouples subsystems, thus reducing the impact of future change, is flexible, provides defined interfaces between components, and separates the environment and system level from the application level. System evolution is supported in all these ways.

Inherently, the DOM can be scaled up. There are no single foci for blockage as long as the static (compile-time) interface is used, as it would be, for real-time processing. The design of the application partitioning, the usage patterns, and the underlying implementation of the DOM may be limiting factors. There might come a point when one or more objects became bottlenecks, due to their heavy use. Redesign by maximizing multi-threading of the objects, by spreading them over multiple processors, by having multiple copies of a given object, may greatly increase the scale-up factor. In any event, the question of scalability has to be quantified. Is the question whether we can increase from five components to 50, from five to 5,000, or is it volume of data we are scaling up? Part of the answer lies in the underlying implementation of the DOM. The rest in the application design and implementation, and the hardware, operating and communication systems. If the underlying systems have been designed and built with one of their prime objectives, of providing scalability, unpleasant surprises are less likely to occur.

Since the current system would be just another component object in the DOM, the barriers to extensibility are those inherent in the design of the current system, not in the new DOM architecture. A DOM is also flexible in providing support for future Software/Hardware (SW/HW) standards. All the capabilities that the legacy system provides will still be available. Adding new interactions with other DOD systems should be looked at on a case by case basis. To determine whether to add this new capability by modifying the legacy system or by simply adding a new object to the DOM system which would communicate with other DOD systems through the external communications links. This is feasible whenever the data in the legacy system which would be needed, can be made available to the new object, or when no such legacy data is needed. As more and more subsystems are migrated to the new architecture, more and more of the underlying data will be immediately available.

### 3.3.3 Infrastructure Requirements to Support the CORBA-Like Architecture

#### 3.3.3.1 Overview

**Operating System:** Real-Time POSIX-RT-compliant operating system which supports multiple threads, priority-based scheduling, event/time-based scheduling and/or signaling, real-time clock and timing devices will be required.\(^5\)

**Communication:** The CORBA design requires that all object requests are processed through the ORBs. The particular communication mechanisms that are used in present day commercial CORBA products vary. One uses RPCs, another TCP/IP. A real-time communication underlayer will be important to ensure that the ORB traffic is bounded. However, the real determinant in the limitation of inter-ORB traffic is the design of the object network and how the objects interact. It should also be noted that, while object requests go through the ORBS, this does not preclude using other real-time

---

\(^5\) POSIX stands for Portable Operating System Interface; RT stands for real-time.
communication mechanisms for other purposes, for example, to broadcast data or to signal events.

**Database:** A real-time database management system which provides predictable, fast, and deadline-cognizant interactions with the DOM and RT operating systems is required. A real-time persistent object-adapter is needed to ensure the predictable operation of the database under the DOM system. Database design within the DOM may be centralized, partially distributed, or fully distributed. The same issues of updates to data which are not in sync with the broadcast data, which other systems need to deal with, have to be addressed here.

### 3.3.3.2 Example

Let the set of tracks and the set of weapons be two different objects in the DOM. Then the tracks object could be assigned to reside in the same node as surveillance and be broadcast to other nodes on either a periodic or event-driven basis, as in the previous example of a centralized database. The weapons object could reside in the node with the weapons application object and could also be broadcast. The central database would serve as a repository for backup and recovery, rather than as the repository of the truth. Each component would be responsible for maintaining, updating, and distributing the data it is responsible for. Distribution could be on a cyclic broadcast as well as upon request. The latency of up-to-date data would be one cycle less for the system this way, because the accurate data would not need to be placed in the central database before the other components would have access to it.

Moreover, the DBMS could allow a given "owner" application object to update its set of data objects while maintaining a time-consistent previous version of that data set for the rest of the system. This will permit an overlap of operations while maintaining temporal consistency. For example, surveillance can be performing association and smoothing on new data while the previous data is available for display, and other operations.

Note that the use of a DOM is consistent both with the central database manager/broadcast concept and with the distributed owner, broadcast concept.

### 3.3.4 Advantages and Disadvantages

The DOM architecture provides well-designed capabilities to encapsulate legacy applications. The CORBA standards are vendor-independent and supported by multiple vendors. They are designed explicitly to provide extensibility and evolvability, and multiple platform support. By using a DOM, the system is providing checked-out functionality for common distributed features which do not have to be provided in the applications using the DOM. An example of how a CORBA-like DOM can be used to one's advantage in a real-time problem is an operation which may have multiple implementations differentiated by the accuracy of their computed result and their time to execution (worst-case, average, etc.). When one implementation is too time-consuming to meet the required deadline, the best implementation feasible can be selected by the ORB/object adapter.

The major disadvantage with DOM is that it is a new technology which is not yet production quality. There are not many trained practitioners. On the other hand, those practitioners that do exist are very capable. Real-time performance is also an issue to be shown.

### 3.3.5 Availability, COTS Components, and Integration

The current status of the CORBA/COSS standard development process is that standards for new capabilities are being actively worked on by the OMG. Five vendors have implemented virtually all of the features of CORBA 1.1 and are committed to the implementation of future standard releases (many more vendors claim CORBA-compliance!) Usage in production systems is just over the horizon and not yet a reality.

There are competing de-facto standards for DOMs, such as Common Object Mode (COM) from Microsoft. The marketplace will eventually show whether one, several, or no DOM standard will prevail. However, distributed systems need a distributed management layer to localize and to carefully and completely handle the common distribution capabilities. While this is an area that is still in great flux, there are enough actual products on the market, that future distributed systems may well benefit from inserting DOM into the technology underpinnings of their architecture.

### 3.3.6 Cost and Risk Assessment

Obviously, the development of the RT DOM is necessary. It is possible that this will be performed by a vendor who has a large customer base requiring such a capability. If not, a Government Off The Shelf (GOTS) effort may be required.

The maintenance of the RT DOM will be a combination of the vendor and a system engineer belonging to the maintenance team of the C3 system, just as the OS system maintenance is currently handled.

Overhead is another issue. For example it can be controlled by:

- Optimizing the DOM for RT applications.
- Coarse-grain encapsulation of applications and data.

That is, surveillance, weapons, and display subsystems can be one object each; DBMS could be an object by itself or the set of track data and the set of weapons could be two objects in the DOM. It is important that the number of objects and interactions between these objects be manageable and predictable. At the present state-of-the-art, this means keeping the number of objects in the low to middle two-digits. However, advances in hardware and firmware will undoubtedly change this over the next three years.

This approach is clearly too risky to commit to for today's C3 systems development. This is because it is a new technology which is not yet production quality (but close). There is also the problem that even some of the
real-time features which are needed in the combined middleware of operating system and communications, have not been fully implemented, heavily used, and integrated together, let alone those of real-time distributed object management systems and real-time database management systems.

However, Distributed Object Management should be an important cornerstone in the design of a future, evolvable, reconfigurable C3 system. We must, therefore, participate in and influence directions for real-time in which this technology is moving, and maintain a level of hands-on experience to clarify and maintain our own knowledge.6

3.4 Conclusions

All three architectures presented above have their strong and weak points. But the central database architecture appears to be closer to current designs as well as easier for current systems to be transitioned. It is also more feasible to implement a system based on the centralized architecture using COTS products. However, both the distributed message-based architecture and the client-server CORBA-like architecture seem to be more appropriate for next generation systems. In any case, our next step is to conduct a detailed evaluation of the three architectures and select one for further consideration.

4. Summary and Future Considerations

This paper has described some of the work carried out during the first year on the Evolvable Real-Time C3 Systems project. The goals of this project are to:

1. Identify software infrastructure requirements for supporting AWACS and other real-time C3 systems.
2. Determine how existing C3 systems could be transitioned into such a software infrastructure.
3. Demonstrate feasibility of the approach through modeling, prototyping, and evaluation of commercial products.

This paper has discussed the approach to designing the evolvable system as well as the architecture considered for development. The next step is to examine the proposed architectures and select one for design and implementation. We will carry out the detailed design of the evolvable system, select commercial products from those we have surveyed and evaluated, and implement a proof-of-concept prototype. In particular, the following tasks are expected to be carried out during the next two years. During the second year, we will acquire selected products and technologies and possibly extend and enhance them to meet C3 real-time requirements. We will also integrate the components to provide real-time embedded infrastructure. The selected functional model will be validated using the MSI application module and subsequently the infrastructure architecture will be evaluated. During the third year, we will have a MSI application hosted on the integrated infrastructure and compare architecture performance to system performance requirements. We will also establish the feasibility of the transition approach. Preliminary results on the related project demonstrate the feasibility of the infrastructure approach [KRUP95].

Acknowledgements

This work was carried out at the MITRE Corporation under contract F19628-94C-0001. We gratefully acknowledge the Air Force Mission Oriented Investigation and Experimentation Program for supporting the work described in this report. We thank Dr. Tom Lawrence of Rome Laboratory for monitoring the project, the AWACS program office and MITRE members of the AWACS project for their support, various groups from Industry and Academia for the interactions and information provided to us, and MITRE management for their support.

List of References