EVOLVABLE REAL-TIME C3 SYSTEMS - II: REAL-TIME INFRASTRUCTURE REQUIREMENTS

Bhavani Thuraisingham, Peter Krupp, Arkady Kanevsky, Edward Bensley, Ruth Ann Sigel, Michael Squadrito, Alice Schafer, Mike Gates, Thomas Wheeler
The MITRE Corporation,
202 Burlington Road, Bedford, MA 01730

Abstract

MITRE's Evolvable Real-Time Command, Control, and Communications (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 project has chosen Airborne Warning and Control System (AWACS) as an example to test the concepts and architectures to be developed. In this paper we discuss the requirements for the infrastructure for next generation complex real-time command and control systems. This discussion also includes an overview of the infrastructure requirements for each of the three architectures that we have considered [BENS95].

1. Introduction

Between now and the early part of the next century, significant portions of today's real-time Command, Control, and Communication systems will become either functionally inadequate or logistically insupportable. 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.

The project has chosen AWACS as an example to test the concepts and architectures to be developed. Currently, its centralized design is a closed architecture with monolithic custom software, that does not take advantages of state-of-the-art hardware and makes processing upgrades time-consuming and expensive. The project has chosen Multi-Sensor Integration (MSI) function [WALT90], because of its support of important combat identification capabilities, as a starting point for transitioning AWACS to an open architecture. 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 specific goals of our project are the following:

- To identify real-time infrastructure requirements (RTIS) to support AWACS and other real-time C3 systems.
- Determine how existing C3 systems could be transitioned into such a software infrastructure.
- Demonstrate feasibility of the approach through modeling, prototyping, and evaluation of commercial products.

In this article we discuss the requirements for the infrastructure for next generation real-time command and control systems. This discussion also includes an overview of the real-time infrastructure requirements for each of the three architectures that we have considered [BENS95]. We first provide a brief overview of our approach to designing evolvable systems as well as summarize the three architectures in section 2. A detailed discussion of the requirements for the real-time infrastructure are given in section 3. In section 4 we provide separate requirements for data management. We conclude the paper by discussing current directions in section 5.

2. Background

2.1 Approach to Building New Systems

In order to provide an evolution path for real-time C3 systems, one needs to understand the requirements of

0-8186-7614-0/96 $5.00 © 1996 IEEE

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

current real-time C3 systems and how they designed. The major goals of the initiative include determining the software infrastructure requirements and identifying 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 Commercial-Off-the-Shelf (COTS) products for the infrastructure. Figure 2-1 illustrates the infrastructure.

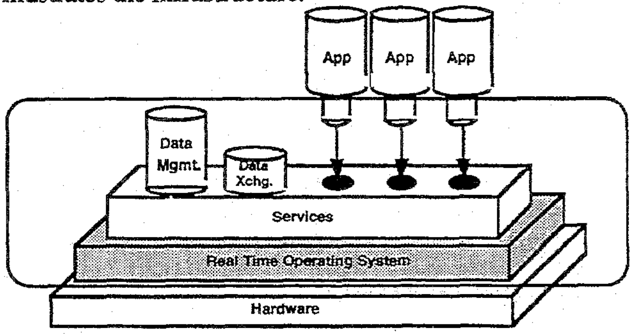

Figure 2-1. Infrastructure

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, real-time data management services such as data sharing, querying, updating, transaction management, and enforces 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.

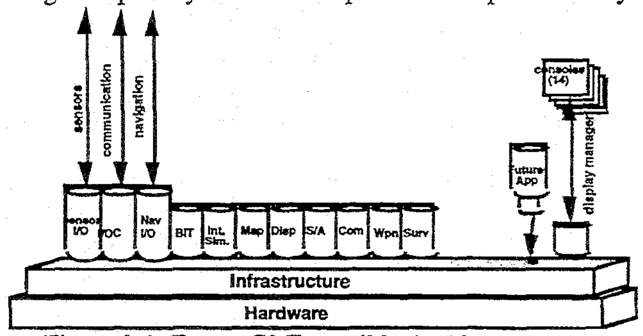

Figure 2-2. Target C3 Extensible Architecture

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 data as well as to communicate with each other. Implementing such a system will mean re-architecting the entire AWACS system. This is not feasible within 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 intermediate architecture is illustrated in Figure 2-3 where MSI is hosted on the infrastructure.

2.2. Architecting a Robust Real-Time System

We have considered three architectures[BENS95]. One is a centralized database architecture, the second is a distributed message passing architecture, and the third is based on a distributed-object client-server paradigm.

2.2.1. Centralized Database Architecture

The central database computer system architecture for C3 systems consists of subsystem functional units operating on possibly separate computer nodes each with their own local data manager and database (see Figure 2.4). 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.

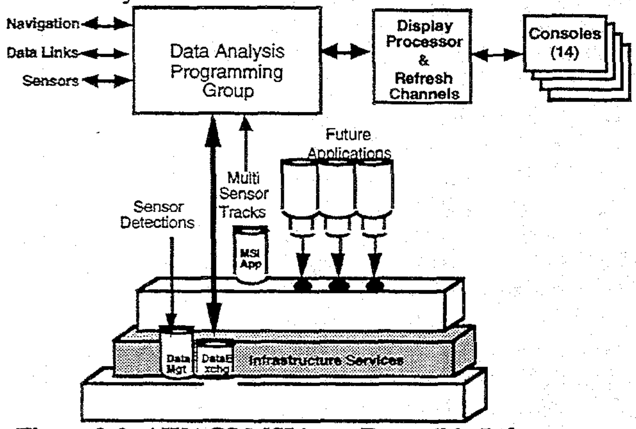

Figure 2-3. AWACS MSI in an Extensible Infrastructure

2.2.2. Message-Based Distributed Architecture

In this architecture, each subsystem has its own individual database of all required data, and different subsystems share/exchange data via messages (see Figure 2-5). Note that there is no central database to synchronize the local databases. Therefore, the local databases have to ensure that they are synchronized by exchanging messages.

2.2.3. Distributed Object Management Architecture

In a distributed object management system (DOM), the components are encapsulated as objects and the objects communicate with each other through some from of object request broker. An example of a DOM is the Object Management Group's (OMG) Common Object Request

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

Broker Architecture (CORBA) [OMG94]. Each component of the DOM is treated 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. 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 as well as distributed message-based architecture can just as well be implemented within a DOM architecture. Figure 2-6 shows the broadcast from the central database with distributed database managers at each component. Figure 2-7 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 that are sent to the objects [WOLF95].

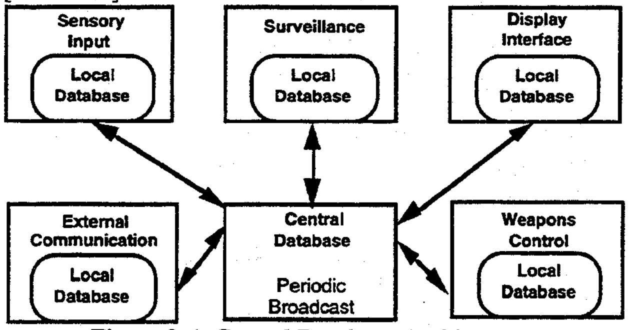

Figure 2-4. Central Database Architecture

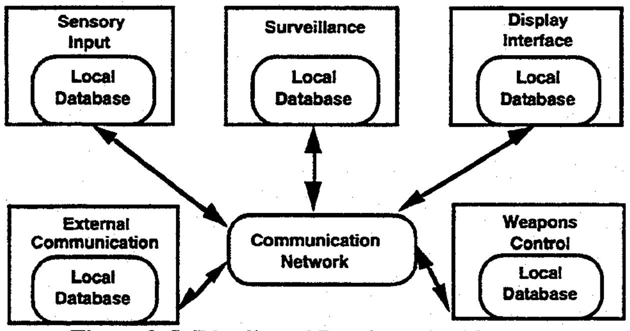

Figure 2-5. Distributed Database Architecture

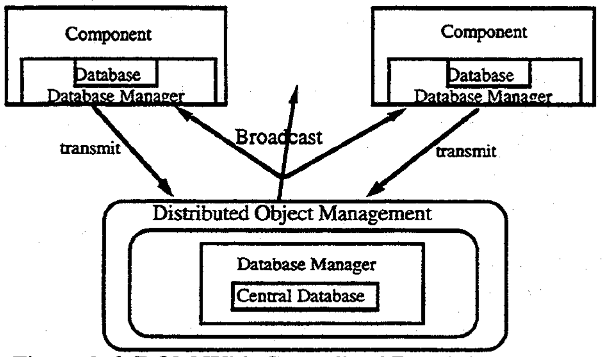

Figure 2-6. DOM With Centralized Data Management

3. Real-Time Infrastructure Requirement

3.1. Overview

The following real-time infrastructure requirements (RTIS) have been identified and discussed in this chapter:[1]

- Real-Time Scheduling,
- Time Constraint Enforcement,
- Transaction Management,
- Scheduler Encapsulation,
- Real-Time Predictability,
- Admission Control,
- Overload Management,
- Real-Time Tasks,
- Real-Time Threads,
- Interthread Synchronization and Mutual Exclusion,
- Interprocess/Interthread Communication,
- Fault-Tolerance and Group Communication,
- Portability,
- Global Time,
- Evolvability and Extensibility
- Object-Orientedness.

Also data management requirements are discussed separately in chapter 4. These requirements are intended to address many of the problems that we have observed in existing C3 systems [GINIS96]. The most important problem is the cost of maintenance (evolution and extension). A significant factor driving the cost is the use of cyclic executives, that results in a global, system-wide dispersal of the knowledge of real-time constraints and behavior of the system. We believe that some of the proposed techniques described below in the form of RTIS design requirements will ameliorate this problem significantly. Through our prototyping of RTIS, we will be able to test our hypothesis.

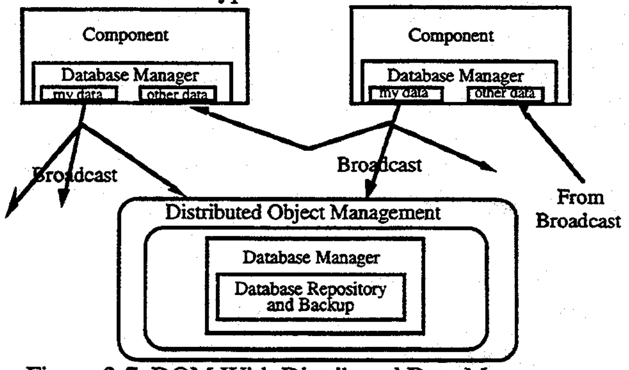

Figure 2-7. DOM With Distributed Data Management

3.2. Real-Time Scheduling

- *RTIS will provide an application with real-time scheduling support services, that include the assignment of operating system scheduling parameters on the basis of the application real-time workload to processes and threads with real-time constraints.*

[1]Note that some of these requirements are design goals and constraints. These will be stated where appropriate.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

Current real-time operating systems do not provide services to support the determination of scheduling priorities, on the basis of workload and timing constraints, which is left to an implementor. By applying existing scheduling theory, we can provide an RTIS service that will calculate scheduling parameters (priority) on the basis of workload and timing constraints.

We believe that scheduling techniques with "predictable" behavior must be used to achieve highly reliable, open real-time systems. One interpretation of "predictable" is that some type of simple analytic model must exist which when given a workload will provide us with a determination of its schedulability. Another desirable property would be a prediction of system behavior under overload conditions so that we can design the system to gracefully degrade when overloaded.

One of the design goals of RTIS is to take advantage of the recent developments in the area of real-time scheduling theory as implemented in recent COTS real-time operating systems such as LynxOS [LYNX] and as defined in emerging real-time operating systems standards such as POSIX 1003.1b, 1d and 1j [POSIX]. The trend is towards preemptive, priority-based scheduling (dynamic and static priorities); including rate-monotonic scheduling and earliest deadline first. Best-effort scheduling (as in the real-time extensions to OSF/Mach developed at the OSF/Research Institute) is closely related, but not strictly speaking a priority-based scheduling scheme. The trend is away from time-driven cyclic executive schedulers and towards event-driven schedulers based on various priority schemes.

It is possible to apply existing scheduling techniques to provide both schedulability analysis and graceful degradation. One example of a technique that has these properties is rate-monotonic scheduling. To do so requires assigning scheduling priorities to each thread (task) in the system according to its period (for periodic tasks) or minimum inter-arrival time (for sporadic tasks). To properly assign priorities to a particular task requires knowledge of the entire processor workload. This is an example of a service that RTIS can provide that will ease the difficulty in using modern real-time scheduling techniques.

3.3. Time Constraint Enforcements

- *RTIS will provide time constraint (real-time and CPU-time) enforcement services. An application will be able to request notification of time-constraint violations (e.g., deadline faults) and then elect to specify fault recovery processing which may include application routines.*

In our initial exploration of how to implement this requirement, we found that a significant design issue was the problem of aborting the execution of a thread cleanly. The POSIX 1003.1c, 1j standards is of little help in this area since it does not provide for the automatic cleanup of a thread when it is aborted. The standard however does provide for user specified cleanup handlers that are executed when a thread is aborted. This, while certainly necessary, seems unsatisfactory in its level of support. Drawing an analogy between a UNIX-style process and a POSIX thread, it would be convenient if threads were cleaned up (resources reclaimed) as when a UNIX process is terminated. The operating system kernel cleans up in most cases and reclaims process resources when a process is terminated. This does not happen with a thread, and if a thread is simply aborted resources such as semaphores, mutexes, and the heap that are process-wide can be left in an ill-defined state.

A second issue is how to carry out the timing fault recovery processing without causing a cascade of other timing faults. If the fault-recovery processing consumes additional processor time, other threads may fail to meet their constraints because of reduced processor time availability or because the thread that has failed also fails to release other needed, shared resources on time. Our current strategy to resolve these conflicting requirements is to require the application to reserve time for recovery by setting a high water mark for both CPU-time constraints and real-time constraints. When the high water mark is met, the application is notified, it then has time remaining to carry out its recovery. This we believe is a simple, effective way to enforce real-time constraints and provide for clean recovery from timing faults.

3.4. Transaction Manager Support

- *RTIS will provide the services needed to support the implementation of a real-time transaction manager as part of a real-time data manager.*

From our exploration of the problem of enforcing timing-constraints, we believe that there is a need for a real-time transaction manager that supports atomic transactions. Transactions implemented as threads must be cleaned up by the RTIS without significant operating system support. The data manager component of RTIS must provide this support (see section 4). We found that the problem of clean termination was a significant barrier to the use of COTS real-time data managers such as ZIP Real-Time Management System (ZIP-RTDBMS) [RTZIP]. Since ZIP-RTDBMS did not provide transactions or any means of cleanly terminating in-progress operation on data under its management, we could not easily use it as a basis for implemention hard real-time database operations.

3.5. Real-Time Scheduler Encapsulation

- *An application shall have no explicit knowledge of the scheduling technique being used by the real-time operating system.*

An important design objective of RTIS is to reduce the cost of the maintenance (extension and evolution) of distributed real-time applications. What we have observed in existing applications is that knowledge of the real-time scheduling technique is typically distributed throughout the application source code. This increases the cost and complexity of the evolution of real-time systems. The

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

application should not be concerned with how real-time constraints are met, only with what the constraints are and what should be done if they are not met. The application will define its scheduling requirements (time/resource constraints), pass them to RTIS which will be responsible for selecting the appropriate scheduling technique and calculating the appropriate scheduling parameters. By providing scheduling support services that keep track of all the application real-time constraints, RTIS can calculate the scheduling priority using rate-monotonic priority assignment rules. Later on if it is desired to use dynamic scheduling with the EDF (earliest deadline first) rule, (and the underlying real-time operating system supports it), the RTIS could substitute that method of scheduling without requiring any changes to application source code. We plan on seeing how far this idea can be extended. Our experience this year supports our claim that knowledge of the operating system scheduling mechanism can be encapsulated within the RTIS. The only real issue is the fact that different schedulers have different levels of efficiency in scheduling and different behavior under overload conditions. We will need to determine through experimentation if this tradeoff of encapsulation vs. scheduling efficiency is worthwhile. We expect that it is.

3.6. Real-Time Predictability

- *RTIS will support the implementation of "predictable" real-time applications.*

The principal design issue is what do we mean by "predictability." Our interpretation here is that we will only use real-time scheduling algorithms in RTIS for which tractable theories exist: simple schedulability models as in rate-monotonic scheduling, deadline-monotonic scheduling, or earliest-deadline-first; or schedulers that empirically have acceptable behavior such as best-effort scheduling. Standard "feedback-queue" schedulers used in most mainframe time-sharing and UNIX systems, emphasizing fairness and deterministic timing behavior, are not acceptable and will not be used. Currently, only static priority-based preemptive scheduling techniques are universally supported in COTS real-time operating systems.

3.7. Admission Control

- *RTIS will support the optional use of admission control. When admission control is enabled, all requests for the creation of threads with real-time and CPU-time constraints are checked for admissibility and are created only if the new workload is schedulable.*

This requirement is imposed because in hard real-time applications, timing faults lead to system failure. Admission control enables the application to monitor the workload through RTIS. Additional units of work (threads) will be rejected when they are created instead of when they fail or trigger other failures. This capability is not needed for all real-time applications, only those that have critical timing constraints. The application will enable or disable admission control as part of its *initial configuration.* We have made this decision in order to simplify the design of RTIS and see no need to dynamically enable or disable this capability. When enabled, it will permit admission control to restrict the creation of real-time threads to those that constitute "analyzable" architectures for the underlying real-time operating system. This means that an application that uses this capability will not necessarily be portable to other platforms that support other (different) analyzable real-time scheduling algorithms.

3.8. Overload Manager

- *RTIS will support the "graceful" degradation of an application under overload.*

An RTIS application gracefully degrades missed in the order of criticality when the real-time constraints of an application are. When a workload is not schedulable, there is an overload condition. The application will enable or disable overload management when it is initialized. A design issue that we have not resolved, satisfactorily, is how to specify the "criticality" of a thread. We will initially simply permit the application to specify a ranking with an integer. Another issue is that overload management is only well understood for a few real-time scheduling techniques: rate-monotonic, deadline-monotonic, and best-effort scheduling. RTIS will only support overload management for these scheduling techniques.

3.9. Real-Time Tasks

- *RTIS will support the creation of a real-time task (a native operating system process) that provides the address space for a set of RTIS and application threads that contain per-task (per-process) resources.*

This is analogous to the Mach concept of a task as a container for threads (a UNIX process without its single thread of control). This is RTIS notion of a process and easily maps on to processes in all modern operating systems. It is a convenient way of managing per-process (per-task) resources such as application-created communication ports, RTIS-created communication ports (for task control and monitoring), and RTIS service threads needed to carry out RTIS control functions.

3.10. Real-Time Threads

- *RTIS will support the creation of time-constrained real-time threads: both periodic and sporadic. RTIS will also support the creation of non-real-time threads for activities that have no critical time-constraints.*

The real-time threads described here are intended to map on to POSIX 1003.1b,1c threads. If appropriate scheduling algorithms are supported, they could be mapped onto other types of operating system supported threads, such as Mach threads. Two types of real-time threads will be supported, periodic and sporadic. A periodic thread will have a default deadline that coincides with the end of the period. A deadline that is earlier or later than the period may be specified. If admission control is enabled, a

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

CPU-time quota for the thread must be specified. A sporadic thread will have a minimum inter-arrival time specified (it will not automatically be invoked periodically). If a request for a sporadic task arrives earlier than permitted it will be either dropped or queued as specified when the thread is created. Non-real-time threads with have no time-constraints can be created (periodic and aperiodic—no minimum inter-arrival time). RTIS will ensure that they are scheduled in such a manner that they do not conflict with the execution of real-time threads.

3.11. Interthread Synchronization and Mutual Exclusion

- *RTIS shall provide interthread mutual exclusion and synchronization services that are integrated with real-time scheduling such that a real-time thread with high criticality cannot be delayed indefinetaly by a less critical real-time thread.*

The synchronization constructs are intended to map onto POSIX 1003.1b,1c synchronization and mutual exclusion services: semaphores, condition variables, and mutexes. The key issue here is the avoidance of unbounded delays due to tasks with low priority holding resources that high priority tasks need (unbounded priority inversion). Unbounded priority inversion may be avoided through the use of the priority inheritance and priority ceiling protocols. POSIX 1003.1c,1j draft standards recognize this problem and provide support for its solution. The real-time data manager requires this support for its implementation of atomic transactions and semantic locking.

3.12. Interprocess/Interthread Communication

- *RTIS will provide IPC/ITC services by supporting communication objects called ports.*

The communication model we have adopted from Mach is simply an extension of the existing model implemented by COTS real-time operating systems. It has been extended so that thread-to-thread communication can take place. Process-to-process communication does not provide a sufficiently fine level of granularity. IPC/ITC will follow the Mach model: RTIS will provide services necessary to create/destroy communication objects called ports (a port is similar to a POSIX/Lynx OS message queue). A thread can send a message using a port as a target and can also receive a using a port as a receiving endpoint. Both blocking and non-blocking sends and receives will be supported. A blocking send will wait until an acknowledgment from the receiving point is received (as a blocking RPC does). RTIS will support sends and receives between threads in the same process and different processes on either the same processor or different processors.

RTIS will provide a name service for the registration of ports. A thread can then advertise the availability of a port by an agreed upon name. RTIS will support the transmission of ports (as objects) from any thread to any other thread. In a heterogeneous environment RTIS will make the machine-dependent representation conversions of messages (byte-ordering, integers, floating point).

3.13. Fault-Tolerance

- *RTIS will support fault-tolerance by providing group communication services (reliable multicast, atomic multicast, causal multicast, and group membership services).*

The ISIS model demonstrated support for fault-tolerance and distributed computing outside of the normal point-to-point communication model, which we adopted for RTIS [BIRM96, ISIS, HORUS]. This supports fault-tolerance by permitting replication of data, real-time processes, and threads. The following group communication services will be supported: reliable multicast, atomic multicast, causal multicast. The associated orderings in message delivery will be enforced with respect to changes in group membership. RTIS will provide support for thread groups: the creation and naming of a thread group, the destruction of a thread group, and the addition and removal of a thread from a thread group. The name of a thread group will be associated with a thread group address. A distributed name service will permit any thread to find the address of a thread group by asking the name service for its address. When a thread group is created, it will be registered with its specified name with the thread group name service. When a thread group is destroyed, it will be deregistered from the thread group name service.

3.14. Global Time Service

- *RTIS will provide a global time service using clock synchronization protocols to keep processor clocks synchronized within a specified upper bound. The application will be able to query RTIS as to the accuracy and maximum skew of the time service.*

A distributed time service will be provided to synchronize clocks across the network. For this project we will use publicly available NTP (Network Time Protocol) that has been ported to a variety of UNIX-like platforms. Its accuracy should be sufficient for this project. Another choice is the Flavio-Christian's fault-tolerant clock synchronization service [CRIS90] that had been implemented by the OSF as part of an x-kernel distribution. To use it we need to port the x-kernel (a network protocol implementation kit) to our current real-time operating system (LynxOS).

3.15. Portability

- *RTIS will be constructed as a layer above POSIX.1,1b.*

If this requirement can be met, RTIS should be portable to any POSIX.1,1b compliant platform. As part of our project we will test this hypothesis. An RTIS application will be portable if it obtains its services through RTIS API.

3.16. Evolvability & Extensibility (Design Goal)

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

- *RTIS will be designed such that if additional capability needs to be added, existing code does not need to be modified.*

This is really a constraint on the design of RTIS. We have adopted an object-oriented approach in the hope that through the use of classes and inheritance we will be able to design and implement RTIS in a manner that permits us to add functionality by inheriting from existing RTIS classes and overriding or adding methods to new RTIS classes. Through the mechanism of inheritance and object composition only new classes need to be added. This goal (while not easily or clearly attainable) is an ideal that should be striven for, if and extensibility are to be properties of RTIS. This is an hypothesis that has been shown to be true for other applications of object-oriented design. We intend to test its applicability to the design and implementation of RTIS.

3.17. Object-Oriented (Design Goal & Constraint)

- *The API for RTIS will be implemented as a C++ class library. RTIS will serve as an object-oriented framework for MSI applications.*

This is needed to support the design goal of evolvability and extensibility. To attain evolvability and extensibility of RTIS and applications using RTIS, we believe we need a CORBA-like distributed object approach [ALLEN96, KOP96]. We are investigating some implementations of distributed object managers that are available from other projects, like ILU (Interlanguage Unification) from Xerox PARC. That would allow RTIS to support a number of languages and extend the object-oriented paradigm to include distributed objects.

Since our project is about the evolvability of C3 systems and MSI is our test application, an objective of RTIS is to act as on object-oriented framework for MSI applications. This provides a significantly higher level of reuse and support for an application than a simple class library. It will enable us to build a collection of collaborating classes which we believe is important if we are to support complex real-time, distributed MSI applications. Since this is desirable and not necessarily achievable with the time and resources available, it is only listed as a design goal and not a requirement.

4. Features of Real-Time Data Managers[2]

C3 systems cannot be just a simple blend of real-time and data management requirements. While a real-time data manager (RTDM) must have many features of database management systems that service complex, multi-user systems, such as multithreading and locking, it must also support real-time systems requirements, such as the performance of transactions within predicted execution times. In addition, it must support the data representation requirements of the applications. For example, not only should the entities of the application and the relationships between the entities be represented, it should also be possible to represent the data dependencies between the processes which operate on the data.

Thus, there are a set of additional issues which arise which are distinct and particular to the maintenance and access of temporal data combined with real-time constraints imposed by priorities, deadlines, and fault tolerance. These issues lead to approaches such as fuzzy responses to satisfy query deadlines and relaxation of serializability. Basic features that data managers are expected to provide are:

- Multi-user access with data locking for serializability, preventing inadvertent corruption of data.
- Data views and data manipulation language on logical views of persistent data.
- Isolation from physical layout and storage of data.
- Indices and/or other optimized mechanisms for fast data access.
- Query and update capability.
- Atomic transactions to maintain data consistency and integrity.
- Backup and recovery facilities.

Many modern data managers also provide additional features which the users have grown to expect as part of a basic data manager:

- Report generators.
- Ad hoc query capability.
- GUI (Graphical User Interface) builders.
- GUI front end.
- Fourth generation application builders.
- Distributed databases.
- Multiple platform support.
- Bridges to other data managers.

RTDMs have to meet additional requirements that do not apply to traditional data managers. Data is time-stamped and transactions are time-constrained; they have a deadline and they have a value to the system which varies depending on whether or not they met that deadline.

In some cases, missing the deadline may be catastrophic. Other new aspects which must be considered when scheduling transactions are: What is the expected execution time, and is the required data sufficiently up-to-date? While temporal data managers also contain time-stamped data, these data managers do not require the kind of scheduling that is necessary to meet strict deadlines, as required for functionally complete RTDMs. In addition, a RTDM must also provide the facility to represent the application and be able to analyze the application for data dependencies and determine potential inconsistencies.

A basic RTDM may not have to supply the additional features in the second list above, since it is generally in a specialized environment and not expected to be a general purpose data manager. In addition, some of the traditional basic features listed for data managers such as data consistency, locking, and transaction serializability are modified when time-constraints and priorities must be considered.

4.2. Discussion of Requirements

[2] [RAMA93] provides good introduction to RTDMs.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

4.2.1. Representation and Analysis

Representing the application is a major requirement. The entities of the application, the relationships between the entities, the constraints, and the data dependencies have to be specified. Constraints include logical consistency constraints, temporal consistency constraints, and timing constraints. Data dependency constraints include statements like transaction T1 can read data item O only after T2 has updated O. An appropriate data model is needed to represent the application.

4.2.2. Temporal Data Management

The temporal data requirements of the C3 systems we have been concerned with, when expressed explicitly, will affect both the data and the transactions to that data. Time-stamped data will have an acquisition time-stamp and a time-decay attribute which describes its interlude of validity. This may be a value, another time-stamp, or a function. Not all the data in the system will be temporal but all data could be time-stamped. A ground installation might be stamped as valid forever, for instance. When data is no longer valid, it could be removed from the database or ignored, depending upon the policy decided upon.

Temporally consistent data is required in order to derive other data or conclusions, or to display a coherent picture of a situation. What is considered temporally consistent is determined by the semantics of the application but must be explicitly specified so that the system can enforce it. There are two kinds of temporal consistencies to consider: (1) consistency with the external frame of reference (absolute consistency); (2) consistency with other data in database (relative consistency). Note that, unlike temporal databases, a composite object in a RTDM may have data attributes with different timestamps. Thus, the altitude of an aircraft may have been reported at a somewhat different point in time as either its speed or latitude-longitude.

4.2.3. Storage and Main-Memory Databases

Other characteristics of RTDMs have led to a rethinking of storage and backup techniques. As mentioned above, one cannot effectively schedule a set of operations which will complete within a given time period without knowing what the worst case execution time of each of these operations are, and what dependencies they have upon each other. Thus, when performing transaction scheduling, execution time and resource usage need to be known or computable so that a guaranteed schedule can be produced a-priori. When this is not possible, a best-effort schedule is made with provisions for aborting, delaying, or omitting transactions which will cause violations of the most serious constraints. There are algorithms for choosing and strategies to choose from. However, there are no guarantees.

One technique that has the advantage of predictability is to maintain all the data in main memory, or, at a minimum, all the data which is used by any of the time-critical transactions. This may, indeed, be reasonable for some of the C3 systems we are currently examining, given the increasing availability of large, relatively inexpensive memory and the possibility of using multiple processors. Use of main memory for data has other advantages; it enables transactions to execute quickly and prevents Input/Output (I/O) operations from adding a large degree of uncertainty to the system.

When main memory is being used for data storage, specialized hardware or software is needed to ensure continuing system operation during power-down is needed, such as battery back-up of power sources, rapid logging devices, and checkpointing that runs off shadow copies of the database, so as not to interfere with on-going data processing. Analysis of main-memory databases has identified storage and access algorithms which perform better for memory database than when disk I/O is the critical factor. For instance, B-tree type indexing is useful with disk-based data managers to minimize I/O. On the other hand, hashing may be used for main memory data managers to minimize CPU usage. New designs, with and without caches, have been developed for RT memory usage which can be applied to RTDMs. There are also RT access algorithms for disks which are sensitive to priorities and deadlines.

Ultimately, we believe that a multi-level storage management policy will be used by next-generation RTDM, using main memory, shared memory, local processor memory, disks, tapes, and new devices.

4.2.4. Transaction Management

As has been iterated before, in order to perform hard-real-time scheduling, transactions need predictable, computable execution times. If the execution of the transaction is not time-critical, then in addition to the worst-case execution time, the average time and a measure of variance from the average could be used to schedule. There has been much work on scheduling periodic tasks for real-time systems. When aperiodic tasks can be treated as if they were periodic ones, this work also applies.

Most of the work on scheduling of real-time tasks is relevant to the scheduling of real-time transactions. Note that the arrival of periodic (every x secs) transactions is predictable. The arrival of aperiodic transactions are not predictable. Transactions may arrive based on conditions in system or from external world. Constraints may be violated and this will cause transactions to be initiated for repair. This may occur either via an application, a data manager constraint checker, or an active RTDM.

Transactions will have time constraints, i.e., deadlines, and priorities, with some policy such as: cannot miss deadline (sometimes called hard); abort if can't be executed by deadline; has some value even if executed after deadline. If the system is expected to guarantee that no hard deadlines will be missed, then executable schedules for hard deadline transactions must be predetermined and a plan to ensure that they will be executed, despite unexpected events, must be in place.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

As mentioned, transactions may have a value (importance of execution) function, and may have predetermined priorities. If a priority doesn't preexist, and the transaction does not inherit it from the task it belongs to, priority will be computed on basis of deadline, value, etc.

4.2.5. Common RTDM and Operating System Needs

Most of the work of RT scheduling is performed within some layer of the real-time operating system. The RTDM scheduling issues are so close to those of the operating system (OS), that we feel that collaborative scheduling algorithms will emerge in the next several years, where the OS will consult with the RTDM about the data resources needed for a task to run. Tasks are units which typically perform a number of database transactions in the process of their execution. With a little extra information about the transactions within a task, the initial schedule generated could avoid data conflicts, which are not considered now.

Some traditional large data managers avoid using the services of the OS they are layered on top of, managing as much of the memory management, buffering, I/O, and threads as they can. Since they have specialized needs and do not have to provide a general service to all applications, they optimized these critical functions for the purpose of maximizing data manager throughput and average response time. Real-time operating systems (RT OSs), however, are designed with most of the same goals as the RTDM, and if the RTDM can use the services provided (and they are real-time POSIX compliant), the total C3 system will be simpler, as well as more portable.

4.2.6.Conflict and overload management

Conflict resolution is needed when competing for resources, such as data locks, prevent priority inversion with some form of priority inheritance, abort (and restart) all but one of the transactions, wait until completion of blocking transactions, and relax serialization requirements.

Another issue is whether to prevent conflicts rather than resolution, and if so, what are the techniques for conflict prevention. Optimistic locking avoids conflict for some time but may add some overhead. The system needs to identify and shed transactions or whole tasks to reduce overload. However, it must be ensured that the hard real-time transactions meet deadlines.

Another issue is to reduce quality of responses rather than no response in a crisis situation. Some current research here is to use alternate algorithms such as sampling data, using older data, extrapolating, and executing partial and imprecise computations.

4.2.7. Recovery Management

An issue here is to recover only temporally consistent and meaningful data. Logging for recovery may add to overload or cause deadlines to be missed. Recovery algorithms must be part of schedulable tasks, suspendable temporarily and resumed later, yet maintaining required consistency of the database.

5. Summary --- Current Directions

This paper has built on our previous paper on evolvable real-time C3 systems. While our previous paper provided an overview of the project and described three architectures in detail, this paper describes the infrastructure requirements. In particular, the requirements for operating system, data manager, and communications are discussed. The architecture study as well as the requirements discussed in this paper have formed the basis of our continuing investigation on this project.

Our current directions are the following. We have carried out an evaluation of the various approaches proposed and have chosen a distributed object management CORBA-like architecture for the system. Since an evolvable design was a major consideration, we were influenced by object-oriented design and implementation. Our infrastructure is essentially a collection of objects interacting with each other. Existing applications as well as new applications can be encapsulated as objects. Furthermore, we have chosen a distributed database approach rather than centralizing the database. We have carried out a more extensive investigation of the requirements for the infrastructure. We have also carried out the design and implementation of the infrastructure, data manager, and are now integrating it with the MSI application. Some preliminary results are reported in [BENS96]. We have also made progress in investigating real-time issues for distributed object management systems [THUR96]. At present, we are using Xerox Corporation's ILU system to integrate the various components of the infrastructure and the application. Some of the details of our real-time CORBA work as well as the design and implementation of the complete system will be described in future papers.

Acknowledgments

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

[ALLEN96] D. Allen, Position Paper: CORBA Technology for Cross-Domain Interoperability in Embedded Military Systems, and Issues in Its Use, *Proceedings of 2nd Workshop on Object-Oriented Real-Time Dependable Systems,* pp. 173-178, 1996.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.

[BENS95] E. Bensley, et al, Evolvable Real-time C3 Systems, *Proceedings of the 1995 IEEE Complex Systems Conference*, pp. 153-166, Nov. 1995.

[BENS96] E. Bensley, et al, An Object-Oriented Implementation of an Infrastructure and Data Manager for Real-time Command and Control Systems, *Proceedings WORDS '96*, Feb. 1996.

[BIRM96] K. Birman, and R. Van Renesse. Software for Reliable Networks, *Scientific American,* May 1996.

[CRIS90] F. Cristian, B. Dancy, and J. Dehn. Fault-Tolerance in the Advanced Automation System, In *Proc.of Fault-Tolerant Computing Symposium,* pp. 6-17, June 1990.

[HORUS] R. Van Renesse, K. Birman, and S. Matteis. Horus, a Flexible Group Communication, *Communications of ACM,* pp. 76-83, April 1996.

[ISIS] K. Birman, and R. Van Renesse. Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, 1993.

[GINIS96] R. Ginis, V. Wolfe, and J. Prichard, The Design of an Open System with Distributed Real-Time Requirements, *Proceedings IEEE Real-Time Technology and Applications Symposium,* pp. 82-90, June 1996.

[KOP96] H. Kopetz and S. Polenda, A Node as a Real-Time Object, *Proceedings of 2nd Workshop on Object-Oriented Real-Time Dependable Systems,* pp. 2-7, 1996.

[LYNX] Lynx Real-Time Systems Inc. http://www.lynx.com.

[OMG94] Object Management Group. Common Object Services Specification, V 1, John Wiley and Sons, 1994.

[POSIX] Potrable Operating System Interface, http://stdsbbs.ieee.org/groups/pasc/standing/sd11.html

[RAMA93] K. Ramaritham, Real-Time Database Systems, *Journal of Parallel and Distributed Computing,* Vol. 1, 1993.

[THUR96] B. Thuraisingham, et al, Position Paper: On Real-time Extensions to Object Request Brokers, *Proceedings WORDS '96,* Feb. 1996.

[WALT90] E. Waltz, J. Llinas. Multisensor Data Fusion, 1990.

[WOLF95] V. Wolfe, J. Black, B. Thurasingham, and P. Krupp. Towards real-time method invocations in distributed computing environment, *Proceedings of the International Conference on High Performance Computing,* Dec. 1995.

[RTZIP] Zip Real-Time Database Management System Manual Dbx, Inc., Cherryhill, NJ, 1993.

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 16,2010 at 18:47:29 UTC from IEEE Xplore. Restrictions apply.