Object Technology for the Integration of the Infrastructure, Data Manager, and Tracker for Command and Control Applications


The MITRE Corporation, Burlington Road, Bedford, MA 01730

ABSTRACT

MITRE’s Evolvable Real-Time C3 (command, control, and communications) initiative attempts to develop an approach that would enable current real-time systems to evolve into the systems of the future. In our previous paper presented at WORDS96, we described the design and proof-of-concept object-oriented implementation of the infrastructure and data manager. In this present paper, we describe the object technology-based integration of the Real-Time Software Infrastructure, Data Manager, and tracking application.

1. INTRODUCTION

Between now and the early part of the next century, significant portions of today’s real-time C3 systems will become either functionally inadequate or logistically unsupportable. Such systems need to change as new requirements are imposed in any of the many component systems. MITRE’s Evolvable Real-Time C3 systems initiative has developed 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 Pre Processing Product Improvement (P3I) 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 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.

We have published a series of papers on the project. In [bens95], we provided an overview. In [bens96a], we described the requirements. In [bens96b], we described the design and implementation of an infrastructure and data manager based on the object-oriented approach. In this paper, we describe the integration of the infrastructure, data manager, and tracking application. We have used an object request broker for the implementation. In section 2 we provide some background information on the project. In particular, we describe our approach to evolving systems. We have designed and developed two major components to support the evolution of command and control systems: the infrastructure and the data manager. These two modules are described in sections 3 and 4. Section 5 describes the particular application subsystem of awacs we have chosen to host on the infrastructure. Integration aspects are discussed in section 6. Summary and directions are given in section 7.

2. BACKGROUND

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 commercial products for the infrastructure. Figure 2-1 illustrates the infrastructure.
The services provided by the infrastructure include operating systems services such as memory management and scheduling, communication services such as interprocess communication, and data management services such as data sharing, querying, updating, transaction management, and enforcing integrity 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 project has chosen AWACS as an example to incrementally test out the concepts and architectures to be developed. It has a centralized database which is based on a closed architecture with monolithic custom software. It does not have a state-of-the-art hardware architecture. Processing upgrades to the system is time-consuming and expensive. The project has chosen to focus on the Surveillance function as the starting point for transitioning AWACS to an open architecture. This is because the target tracking algorithm within the Surveillance function is a prime candidate for system improvement due to recent advances in multi-sensor integration (MSI) trackers.

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 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 mission computing system. This is not feasible within current budget constraints. 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 MSI processing within the surveillance subsystem is hosted on the new infrastructure.
4. DATA MANAGER

4.1 OVERVIEW

Our data manager implements a priority ceiling protocol that uses the semantic information of objects in a real-time database. This protocol is called the affected set priority ceiling (ASPC) protocol. This protocol is compatible with semantic concurrency control techniques that support data logical consistency in real-time object-oriented databases. Details of the protocol with proofs are given in [SQUA96]. This section described the implementation of the data manager.

The ASPC protocol was designed and implemented as part of a prototype real-time data manager being developed at MITRE. This implementation was used as a testbed for evaluating the ASPC protocol. The entire prototype design developed at MITRE is comprised of many components, including a real-time infrastructure class library, real-time monitoring and control processes, a data manager, a real-time database, and a test application. This paper is concerned with the implementation of the data manager design, which consists of a user interface, query manager, meta data manager, transaction manager, constraint manager, object manager, storage manager, and a persistent database. The data manager in this prototype is responsible for controlling the concurrent access of the objects in the database. This paper is concerned with the design of the meta data manager, transaction manager, and the object manager. The meta data manager stores and controls access to the meta data for all of the objects and transactions in the database. The transaction manager uses the meta data manager to determine the concurrent interaction of transactions. Transactions are executable code that access the objects in the database. The object manager stores and retrieves objects from the database.

The implementation of the MITRE prototype is now completed. The prototype is being developed on 486DX2 66 computers running the Lynx 2.3 operating system. Lynx is a POSIX compliant operating system, having the features required by POSIX.1, POSIX.4, and POSIX.4a standards. The objects and meta data manager are implemented in
shared memory. Shared memory is a POSIX.4 feature, and allows multiple processes to access the objects and meta data as if the memory were in their own address space. The prototype is currently designed for multiple applications running as multiple processes. Within each process, multiple transactions may run and have access to the shared memory objects and meta data. Each transaction is a thread as described in the POSIX.4a standard. A thread can be thought of as a light-weight process, each thread in a process having access to that process’s memory. A process gains access to the shared memory by instantiating a transaction manager in its own address space. The transaction manager class maps in the shared memory. The process and transactions have no direct access to the shared objects or meta data. A transaction must acquire an exclusive, read, write, or method lock. The transaction is given a shadow copy of the attributes in the shared object which are specified by the lock’s read/write affected sets. Once the method or methods have finished, the transaction releases the lock, but must commit any writes if the changes are to be reflected in the shared memory object.

Two models were used in the overall design and implementation for this project. The RTSORAC (Real Time Semantic Objects, Relationships And Constraints) model [PDPW94] was used as the basis for the object and transaction descriptions. The ASSET (A System for Supporting Extended Transactions) [BDGJR95] facility was used as the basis for the transaction manager design. In the remaining subsections, we discuss the details of the implementation.

4.2 MEMORY MANAGEMENT

Since the priority ceiling protocols require a priori knowledge of the objects and transactions, the objects and object meta data structures are statically instantiated in shared memory. An overloaded new operator is used for this purpose. In addition, all structures used by the meta data manager are also placed in shared memory at this time. The correct number of meta data manager structures is determined by using the knowledge of the maximum number of objects, transactions, and locks that will be used in the worst case.

Shared Memory: Objects, object meta data, and structures used by the meta data manager are stored in shared memory. To ease the use of the POSIX shared memory feature, the MITRE prototype used a dynamic shared memory manager developed by John K. Black at the University of Rhode Island. This C++ class allows the dynamic allocation and deallocation of shared memory objects. Each object is retrieved by use of an application assigned object identification number. When the MITRE project was started, it was not known that static allocation of the shared memory would be sufficient. The dynamic memory manager class is currently only being used during initial allocation of the shared memory objects, at which time the meta data manager obtains pointers to all shared objects. No overhead is added while the application is running. The port of the dynamic allocation/deallocation implementation requires more work if dynamic objects are required. Currently, all objects must be allocated in shared memory at one time.

Object Manager Implementation: The object manager acts as a static memory allocator, allowing the application to flag objects in shared memory as being active. Although all objects are pre-allocated in shared memory, these are just place holders for when an object is actually used in the application. For example, an application may need an array of 100 objects of a certain type which would be pre-allocated. During execution, the application is able to “activate” objects as they are needed. Having the object manager keep track of active objects allows applications to select all objects, and be assured that only objects with actual information are returned. The object manager may be used in a few ways. The first way allows the application to activate all objects before the application begins execution. To do this, the application declares one object manager capable of indexing all objects in shared memory. The application then activates the objects. The second way is to declare a separate object manager for each object type. During execution, the application activates objects as they are needed. The object managers keep track of the number of available objects for each type. The third use of the object manager combines the previous two, allowing the application to activate selected object types, while reserving activation of others during run-time.
4.3 TRANSACTION AND TRANSACTION MANAGER IMPLEMENTATION

Transactions request locks on objects or the methods of objects. These requests are either granted or denied by using the priority ceiling protocol. Transactions in the prototype are C++ programs that execute as threads. The MITRE prototype requires that all threads be instantiated by using an infrastructure thread class. An infrastructure thread is a C++ class used to encapsulate the operating system calls needed to execute a thread. Use of the infrastructure class ensures that the thread is assigned an appropriate priority, and that the thread can be monitored if required. The infrastructure classes, including threads, mutexes, and semaphores, as well as the priority server, were implemented. Each transaction has access to the transaction manager instantiated in the process, and thus has access to the shared memory objects. The transaction manager uses the priority ceiling protocol to control the concurrent execution of the transactions. The transaction manager is implemented as a C++ class and contains various primitives as public member functions. The class has a private meta data manager, which allows the transaction manager access to the shared memory objects. The request_lock and release_lock methods execute the priority ceiling protocols.

Three priority ceiling protocols were implemented as part of the data manager using the same functions and meta data structures. Once the implementation was completed for the ASPC protocol, the implementation was able to support all three protocols. Each object sets a conflict priority ceiling for each method, an exclusive priority ceiling, a read priority ceiling, and a write priority ceiling. A transaction then has the flexibility to request an exclusive lock, read lock, write lock, or method lock.

4.4 META DATA MANAGER IMPLEMENTATION

The meta data manager contains all of the structures required to implement the priority ceiling protocol. The meta data manager is implemented as a C++ class and is based, in part, on the implementation described in [BDGJR95]. The structures and implementation required for the priority ceiling protocols was obtained from [BR89].

Support Structures: The ASSET [BDGJR95] implementation describes three major structures: the Transaction Descriptor (TD), the Lock Request Descriptor (LRD), and the Object Descriptor (OD). Each of these C++ structures required additional fields in order to implement the priority ceiling protocols. We implemented all the structures. Since the priority ceiling protocol is being implemented within the data manager, and not the objects, it made sense to store the meta data required for the priority ceiling protocols within the meta data manager structures and not directly in the object.

The Meta Data Class: The meta data manager has as private members, a hash table for TDs, an array of OD pointers (one for each object in shared memory), a priority queue for granted requested locks (GRL), and a last-in-first-out (LIFO) list for pending transactions (PTL). The user of the prototype initializes constants which enable the meta data manager to allocate the proper number of TD, LRD, and OD structures.

The hash table is used to locate a transaction by its id, which is the same as the thread id assigned by the operating system. The ASSET [BDGJR95] implementation, which was not being used for real-time, suggests the use of a hash table for locating the TDs because of its excellent average case performance. Although a hash table has poor worst case performance, it can be bounded given a certain number of transactions, and it can be easily replaced if the performance is not acceptable. The array of OD pointers is used to locate an object in shared memory. The array indices are the same as the object ids.

The GRL queue and LIFO PTL are described in the implementation of [BR89]. The GRL is a priority queue of currently held locks (LRD structures) enqueued in highest to lowest order based on the priority ceiling of the lock. This queue is used to determine if a transaction may be granted a lock. The LIFO PTL is a list of pending transactions (TD structures). Each time a high priority transaction is blocked by a lock held by a lower priority
transaction, the high priority transaction is placed in this LIFO list until the lower priority transaction releases the blocking lock. Use of the GRL and PTL structures will be explained in detail in the next Section since they are directly related to the ASPC protocol.

The meta data manager also has a mutex and condition variable which are used to control access to the meta data. The mutex provides mutual exclusion access of the meta data, and the condition variable is used to signal other transactions when a lock has been released. Currently, the GRL queue and LIFO PTL are implemented as part of the meta data class. These structures should be implemented as separate classes for a more object oriented approach, in the same manner the hash table was implemented.

4.5 OBJECT TYPE IMPLEMENTATION

The object type in the MITRE prototype is a subset of the RTSORAC object type. The schema is specified by C++ classes that are of the handle/body idiom [Cop92]. The body contains the object’s attributes and meta data, and is stored in shared memory. In the conventional handle/body design, the handle contains a pointer to a body object, which is assigned when the handle is instantiated, along with the methods used to access the body. The RTSORAC model uses this idiom, however, the MITRE implementation required atomic transactions. Therefore, the handles are given a shadow copy of the object to enforce atomicity, and a pointer to the object’s meta data in shared memory. Currently, the copy is instantiated in the process’s own address space. This could be changed so that the copy is instantiated in shared memory by using the dynamic shared memory manager.

Any shared memory object or shadow copy object class that is to be used by the prototype must be derived from a base class. This base class contains fields that are required for all schema classes, such as the number of attributes. Attributes are themselves specified by C++ classes, and are parameterized (C++ templates) so that any C++ basic type can be used. The attribute class has a private value variable that may be accessed in two ways. The first way is with two public methods, one to read and one to write the actual private value variable. The second way to access the value member is by using the overloaded equal operator for the class. The meta data is stored in shared memory at a known location for each class. Therefore, each object instantiated for a class shares the meta data, such as the number of methods and attributes, and the read/write affected sets. In addition, the transaction manager needs to know the offsets of the attributes within the body to transfer values between the shadow copy object and the shared memory object. The user is required to specify a constructor which takes as arguments, pointers to the attributes within the body. When the meta data is placed in shared memory, this constructor is called, and an offset table is initialized that contains each attributes’ offset with respect to the object’s location. The get_attr(object pointer, attribute number) method of the base class is used to convert the relative offset into a pointer to an attribute.

Instantiating an Object: First, the user must specify a schema object in the handle/body form, along with the class meta data (number of attributes, number of methods, and the read/write affected sets). Base classes were implemented for attributes, bodies, and meta data. The handle does not need to be derived at this time. Second, since the priority ceiling protocols need both affected set information and transaction priority information, the user must also specify the highest priority transaction that will access each method of the object.

The class meta data is determined by using a parser, or in the case of the testing analysis, the class meta data was generated automatically. The parser was written using GNU flex (lexical analyzer) and bison (parser), and takes as input the object schema files (both the header and source files). The only requirement is that the attributes be accessed using the read and write member functions of the attribute class.

The meta data for the objects is placed in shared memory first. When the meta data constructor is called, a temporary object is instantiated, and the attribute offset table is created. The object can now be instantiated in shared memory using the parameterized class called OD. The OD class takes as parameters, the type of the body class, and the number of methods in the handle class. The
constructor of the OD class takes as arguments, a pointer to the class meta data, and an array of the highest priority transactions that will access the methods. Each array index/lociation is associated with a method. Presently, the user determines the highest priority transaction that accesses each method. In the future, a parser can be implemented which could analyze the transactions to determine these priorities.

5. MULTISENSOR INTEGRATION TRACKING

The current goals of the AWACS project are to upgrade the onboard CC-2E mission computer system to, first, provide current system operational capability, then secondly, provide additional functional capability. The upgrade of the AWACS central computer system is planned to take place in stages. The first stage would attach an adjunct computer system to the existing CC-2E central computer. Functional components of the C3 system would incrementally migrate from the current CC-2E to the adjunct computer.

The first functional area to be migrated to the adjunct computer would be the multisensor integration (MSI) tracking system. Replacing the current tracking system with a MSI tracking system would significantly improve the tracking performance of the AWACS while reducing the operator workload. Ultimately, the CC-2E computer would be replaced with the open architecture-based adjunct computer.

The MSI algorithms attempt to associate and correlate all sensor data with existing system tracks. The tracks are then smoothed with the correlating sensor data using an Extended Kalman Filter (EKF), which accommodates the differing dimensionalities of the sensor reports. Sensor identification information is fused with track identification to determine the best possible estimate of target type, status, and intent. Sensor types supported by our tracker include: Pulse Doppler Radar (PDR); Beyond The Horizon Radar (BTH); Identification Friend or Foe (IFF); and Electronic Support Measures (ESM).

After the kinematic track smoothing and identification estimates, tracks are updated for track maintenance. Track quality scores are assigned to the tracks based on how consistently they are updated. If the track quality drops to zero because of a lack of consistent updates, the operator is alerted. The design of the Multisensor Integration (MSI) tracking system did not change significantly this year.

Figure 5-1 is a high level data/control flow view of the tracker and its environment. The emphasis this year was on the integration of the tracker with the data manager and infrastructure. We refer to Section 6 of this paper for information on the integration. An overview of MSI is given in [Bar93].

![Figure 5-1. Tracking System Control and Data Flow](image)

6. INTEGRATION

6.1 OVERVIEW

Over the last year the Multisensor Integration (MSI) tracking system has been integrated with the real-time software infrastructure and data transaction manager. The software infrastructure includes an interface to a POSIX real-time operating system (Lynx operating system) which spawns processing threads for bundles of track processing work and schedules them as a function of their timing requirements and expected CPU usage. The data transaction manager makes consistent and atomic data transactions between all software modules which access the common shared memory. This section describes the effort involved and lessons learned in integrating these three "components" into what we call a demonstration configuration. We have used an Object Request Broker (ORB) to integrate the various components. A high level integration architecture is illustrated in...
Figure 6-1. Detailed discussion of the reasons for selecting an ORB for integration as well as implementation details are also given (for a discussion on some real-time issues, we refer to [THUR96]). The organization of this section is as follows. Demonstration configuration is discussed in section 6.2. Various aspects of the integration are discussed in sections 6.3 - 6.8. Finally lessons learned are discussed in section 6.9.

![Figure 6-1. High-Level Integration Architecture](image)

### 6.2 DEMONSTRATION CONFIGURATION

Figure 6-2 is a diagram of the demonstration configuration. The components are the Sensor Model, AWACS Emulator, Operator Display, three multisensor integration (MSI) tracking systems, the Shared Memory, and the Utilization Display. Descriptions of the Sensor Model, AWACS Emulator, Operator Display and MSI tracking systems can be found in our previous papers. The Shared Memory and Utilization Display are described in Sections 6.6 and 6.7.

The Sensor Model, AWACS Emulator, and Operator Display run on Silicon Graphics workstation computers. The MSI tracking systems and the Shared Memory run on an Intel x486 based PC running the Lynx/OS POSIX real-time operating system. The Utilization display runs on a separate Intel x486 based PC running the Lynx/OS.

![Figure 6.2. Demonstration Configuration](image)

### 6.3 DATA FLOW AND SYSTEM TIMING

The AWACS Emulator receives sensor data and AWACS navigation data from the Sensor Model every 3 seconds. The Operator Display receives sensor and track data from the AWACS Emulator and sends user commands back to the AWACS Emulator every 3 seconds. The AWACS Emulator imposes the AWACS system timing requirements on the adjunct computer. The C3 components (three MSI tracking systems) running on the adjunct computer must receive the sensor data periodically from the AWACS Emulator, and the track and command data must be exchanged during the particular time slots allowed by the mission software’s cyclic executive.

The three MSI tracking systems operate at different processing intervals (3, 9, and 12 seconds). The AWACS Emulator communicates with the tracking systems every 3, 9 or 12 seconds. Only the 12 second period tracking system sends tracks back to the AWACS Emulator. The tracking systems send their CPU usage and elapsed time to the Utilization Display.

### 6.4 INTERPROCESS COMMUNICATION

The AWACS Emulator communicates to the C3 components via the Interface Language Unification (ILU) software, which is a freeware CORBA product from Xerox. ILU does not implement all of CORBA 2.0 features, but we only are using the static bindings for interprocess communication. The interfaces between the AWACS Emulator and the three tracking systems are described in Section 6.6.
systems are defined in the Interface Design Language (IDL). Because IDL is the standard way to define module interfaces for all CORBA products, to substitute another CORBA product for ILU would be a relatively simple matter because the interfaces and service implementations would not change. Only the makefile used to build the inter-communicating modules would change to accommodate the particular stubs to be compiled and linked. It has been found that the interprocess communication between modules over the MITRENET network has been reliable, even more reliable than a similar remote procedure call (RPC) implementation developed as a precursor to this CORBA based demonstration system.

6.5 C3 COMPONENTS

Ideally we would have used three different C3 components for our demonstration, such as a tracking system, a weapon control component, and a communication component. Since we have only operational code for the MSI tracking system we duplicated it three times to place realistic communication and data transaction demands on the system. The three tracking systems all share the same tracks through the same shared memory. This emulates three C3 components all using the same database. For example, in a real C3 system the tracking system provides the updated tracks, the weapons control function requests specific tracks, and the communications function reads all the system tracks and writes external tracks.

6.6 USE OF THE DATA TRANSACTION MANAGER

The interface to the Data Transaction Manager (DTM) is made easy having been written as a class in the C++ programming language. Processing objects in the tracking system are instantiated classes written in C++ that are derived from the DTM classes. That is, the tracking system processing objects inherit the functional capability of the DTM. When a processing object is created it has access to the shared memory through the functions of the transaction manager. When tracks are read from or written to the shared memory the functions of the DTM perform all the necessary locks to maintain data consistency and atomicity.

The integration of the data manager with the tracker took approximately two weeks to complete. The first step was to identify the object types to be stored under the control of the data manager. These types needed to be converted to objects the data manager could use. The only data type to be stored in shared memory was the “Sub_Track” structure. Since the tracker implementation already accessed the data members directly, it was easier to convert the structure using an array called AttrBase to reference the existing data members.

Once the data was converted, the second step was to alter the tracker to obtain new track data from shared memory in lieu of its local memory as was done in the past. The change ended up being relatively straightforward to implement. Originally the tracker was designed to request new “Track” structures, and saved “Sub_Track” structures for use by other applications. Since we only needed to store the “Sub_Track” structures in shared memory, a small fraction was required to convert the Sub_Tracks into Track structures.

6.7 USE OF THE OPERATING SYSTEM INTERFACE INFRASTRUCTURE

The operating system interface infrastructure (OSII) interfaces the tracking system processing objects to the POSIX real-time operating system. The OSII interfaces easily to the track processing objects, having been written as classes in the C++ programming language. The track processing objects inherit the OSII functionality by deriving their classes from the OSII classes.

When a track processing object is instantiated, the OSII constructor is run. The OSII constructor creates a processing thread (a separate processing entity which shares the same address space and variables with the parent process) and establishes its priority based on its assumed processing time. The idea is if all the processes running on the computer have their priorities set as a function of their predicted processing times, in accordance with rate monotonic scheduling theory, and all processes do not overrun their predicted processing times, then the computer system will meet its timing requirements without using a cyclic executive to schedule processes.
6.8 THE DISPLAY

In order to analyze the system, each tracker process reported its CPU usage and total elapsed time to a graphical display server running on a separate computer. The display server was written using Python, which is an object-oriented scripting language. Python incorporates Tcl and the Tk toolkit, which is used to generate Xwindows graphics, including buttons, scrollbars, text windows, etc. The display served two purposes. The first was to show the actual CPU utilizations of each tracker. CPU utilization is equal to the execution time of the task divided by the rate (period) of the task. This indicates when a tracker over-used the CPU. Additionally, the elapsed time to complete the task was reported to indicate when a tracker missed its deadline. The second purpose was to allow a person to use rate monotonic analysis (RMA) and determine how much more CPU utilization was available for another task.

The display was written as an ILU distributed server. The original idea was that the trackers would run on one computer and send their reports via asynchronous RPCs through ILU method invocations. This would have been ideal, since the trackers’ performance would not be affected by the display. However, several problems with this set-up occurred. The first was that the information transmitted through ILU from the trackers was not received correctly by the display. The cause of this problem was not pursued because of another limiting problem. The display could only handle input from one process at a time. The display could handle multiple inputs from threads within a single process, but could not service more than one process. Since the trackers were implemented as independent processes, an alternate method had to be determined to get the trackers timing reports to the display. The solution was to have each tracker write its timing reports to a file. A process was written that was connected to the display via ILU and opened the trackers’ reports from the NFS files. This reporter process read the timing reports and sent them to the display.

The upper half of the interface displays the actual CPU utilization percentage used by each tracker. The lower half of the interface is where a user can analyze the system using RMA. The trackers and the display may be started in any order. When the display is first started, the Total and Blocker interfaces are also generated. The Blocker represents the running task with the highest blocking time. Blocking time is the time a low priority task prevents a high priority task from running while holding a common resource. In the case of the trackers, the common resource is the track table in shared memory. RMA analysis requires the blocking time for an accurate analysis. Both the Total and Blocker indicators are initialized to zero. Once the display is running, a process is executed which registers the trackers with the display. When a process is registered with the display, an upper and lower interface are added to the display. The ring indicator and slider for each task is set equal to the initial execution time (extime) passed into the display. The Blocker indicator is adjusted if the registered task has a higher blocking utilization than the current highest blocking time.

When the slowest rate tracker has written to its time reporting file, the display reporter process may be started. This process continuously reads from the trackers’ time reports and sends the elapsed time and execution time to the display via the ILU ReportTime method. The ring indicator of each tracker moves at the same rate as the respective tracker rate, moving up at the end of a tracker’s execution, and back down to zero at the start of a tracker’s period. Because the reports are read from the NFS file, there is some lag between the display and the trackers.

6.9 LESSONS LEARNED

Threads Running Over Predicted Execution Time: In order to implement a rate monotonic scheduling policy, there must be a way to know when a particular process overruns its predicted processing time, and terminate it. No monitor of CPU usage was implemented due to problems writing the device driver. If a processing thread is terminated because it took too much time, there must be a strategy to deal with completing the work of the processing thread. This was not realized, but could be by rescheduling the thread with a greater predicted processing time. The system could be designed to adjust its processing estimates the more experience it gets.
Single CPU Only: The current demonstration system runs the adjunct computer on a single CPU system. This was done to allow us to study the scheduling characteristics of the system. This configuration does not fully take advantage of the separate processing threads that are created to process units of track processing work. Also the three tracking systems (emulating three C3 components) are running on the same single CPU computer. A more effective tracking demonstration would have been to run on a symmetric multiprocessor (SMP) containing multiple CPUs with a shared memory. This type of parallel processing is the most likely proposed by contractors for an adjunct computer. Although this type of computer was available, it did not have an operating system that supported a pre-emptable kernel. Only now does the Silicon Graphics IRIX operating system support the pre-emptable kernel and POSIX threads.

7. SUMMARY AND DIRECTIONS

The paper has provided a description of our work on evolving complex real-time command and control systems. In particular the design and implementation of the data manager, infrastructure, and tracking application are described. Then a detailed discussion of the integration of the three modules is given. Future directions include the use of real-time middleware for integration. In this direction, we are examining the real-time aspects of the Object Management Group’s Common Object Request Broker Architecture. This will enable the encapsulation of complex components as objects and for these objects to communicate with each other in real-time. Another direction is to examine object-oriented frameworks for integrating various components for real-time C3 systems. We believe that our project shows how to evolve legacy real-time C3 system as well as to transfer the real-time systems research to operational systems.

ACKNOWLEDGMENTS

We gratefully acknowledge the Air Force Mission Oriented Investigation and Experimentation Program for supporting the work described in this paper. 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. We would especially like to thank Edward Bensley for his vision and wisdom in the direction of this project.

REFERENCES


[BENS96b] Bensley E. et al, Design and Implementation of Object-Oriented Infrastructure and Data Manager, Proceedings of the WORDS 96 Conference, Laguna Beach.


