Object-Oriented Implementation of an Infrastructure and Data Manager for Real-time Command and Control Systems

Edward Bensley, Peter Krupp, Ruth Ann Sigel, Michael Squadrito, Bhavani Thuraisingham, Thomas Wheeler
The MITRE Corporation, Burlington Road, Bedford, MA 01730

ABSTRACT

MITRE’s Evolvable Real-Time C3 (command, control, and communications) project attempts to develop an approach that would enable current real-time systems to evolve into the systems of the future. The project has chosen AWACS (Airborne Warning and Control System) as an example to test out the concepts and architectures to be developed. In this paper, we describe an object-oriented implementation of an infrastructure and data manager, for next generation real-time command and control systems.

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 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 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 P31 (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 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 (Airborne Warning and Control System) as an example to test out the concepts and architectures to be developed. 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 the 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 identify open software technology applicable to AWACS and other real-time C3 systems. The successful execution of this project would substantially reduce the risk of transition to open systems.

In an earlier paper [BENS94], we discussed issues on developing the infrastructure requirements. In particular, we first examined the features of current real-time C3 systems and then determined the systems of the future. Next, we defined various candidate architectures for future systems. Finally, we developed requirements for the infrastructure. The main focus was on operating systems, data management systems, and communication systems.

In this paper we describe the object-oriented implementation of the infrastructure and the data manager. Our infrastructure has focused on the operating system and its associated extensions. Note that while the data manager is part of the infrastructure as far as the application is concerned, it can be regarded as an application by the operating system. Since an evolvable design was a major consideration, we were influenced by object-oriented design and implementation approaches as well as our initial investigation of real-time issues for distributed object management systems. Our infrastructure is essentially a collection of objects interacting with each other. This way existing applications as well as new applications could be encapsulated as objects. The features supported by our real-time infrastructure (RTIS) include real-time scheduling and constraint enforcement, real-time predictability, admission control, overload management, real-time tasks and threads, interprocess/interthread communication, and fault-tolerance and group communication. These features address many of the problems that we have observed in existing C3 systems. The most important problem is the cost of maintenance (evolution and extension). A significant factor driving the cost is the use of the cyclic executive. It 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 in the form of RTIS design requirements will ameliorate this problem significantly. Through our prototyping of a RTIS, we will be able to test our hypothesis.

Our real-time data manager (RTDM) is an application hosted on the RTIS and makes use of the features offered by the RTIS. 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, an RTDM must also provide the facility to represent the entities, relationships, and constraints of the application. Be able to analyze the application for data dependencies, and determine potential inconsistencies.

One of the early tasks was to choose the implementation platform, COTS (Commercial Off the Shelf) products, and the language for implementation. This selection was also constrained by the resources that were available to us. After examining the various options, we decided to carry out the implementation of the RTIS on the Lynx platform [LYNX]. The Lynx operating system provides some of the basic support required by the RTIS. Since an object-oriented approach influenced much of our design, we chose to implement the RTIS in C++.

For the implementation of the data manager, we initially chose ZIP-RTDBMS (product of DBx Inc. [ZIP]) and encapsulated it as a C++ object. However, this product does not support some of the essential features such as transaction management and concurrency control. Therefore, we implemented a shared memory data manager with concurrency control support ourselves, and ZIP-RTDBMS is being used for storage purposes.

Note that in this paper we describe only the implementation. We have also carried out a detailed design of the infrastructure and data manager. Details are given in [BENS95].

2. INFRASTRUCTURE IMPLEMENTATION

2.1 APPROACH

Some of the critical requirements that have influenced the design include: real-time scheduling/constraint enforcement, fault tolerance/redundancy, management, object management, and distributed systems support. We started initially with a real-time distributed object management reference model illustrated in figure 2-1. In this model, applications and resources are encapsulated as objects. The real-world constraints that influenced this approach were the following. Most software/hardware technology does not directly support real-time processing, although there is some support for POSIX 1003.4 (Real-Time Standard) [IEEE] in emerging operating systems such as LynxOS. There is inadequate transparent distribution support in current operating systems. Furthermore, current operating systems do not support heterogeneity (machines, languages, etc.). Nor operating system supports timing-constraint enforcement directly (timing violation detection and recovery). Any near-term, open, real-time infrastructure must address all these constraints and issues.

Our design is based on real-time objects which combine C++ objects and Posix 1003.4 threads. Objects are treated as common real-time system components/entities. Polymorphism is used to map real-time operations supported by multiple implementations onto their correct run-time implementation. Abstract classes represent common characteristics of real-time activities (deadline, schedulability, workload, etc.). Concrete classes represent realizations (threads with hard deadlines invoked periodically, message streams with periodic arrivals and hard deadlines, sporadic tasks, etc.). Container classes are used to encapsulate non-real-time components.

Figure 2-1 Real-time Distributed Object Management Reference Model

The RTIS Application Programming Interface (API) consists of a C++ class library. This is needed to encapsulate the creation of schedulable units of work (processes and threads), provide abstractions for application development, and to utilize polymorphism and inheritance to reduce real-time system complexity. API functions (within a single environment) include create and destroy constrained threads, create and destroy synchronization objects, provide interprocess/thread communication, time constraint enforcement and exception processing, register and unregister threads, and synchronization of objects with priority service. Priorities are assigned as directed by a priority service.

2.2 REAL-TIME INFRASTRUCTURE LIBRARY

We have chosen LynxOS as our operating system platform for the development of the Real-Time Infrastructure Library. Lynx gives us the following features:

- Real-Time pre-emptable kernel.
- POSIX compliance for most of the POSIX real-time extensions.
- POSIX threads.
- Support for Priority Inheritance and Ceiling.
- Multi Hardware Platform availability.
- Availability of per-thread timers.

We created the Real-time Infrastructure Library to encapsulate and extend the use of the POSIX and Lynx Real-time features. Using POSIX threads and the LynxOS Real-time features can be confusing and requires a large quantity of setup code before using the features provided. We have eliminated this setup code with the class library. We have also extended the functionality of threads by allowing them to have their priorities set by a server that calculates the priority based on the period of a task, have timing constraints based on CPU usage or other timing criteria enforced, and to be controlled by events generated by other processes or threads.

The library encapsulates the POSIX and real-time portions of the operating system. The library also emulates some features we believe are important that are not yet in the standard or are not directly supported by Lynx. Examples of these are POSIX threads which are not yet in the standard. Lynx POSIX threads (pthreads) are based on Lynx system threads and as such do not quite meet the proposed standard. Semaphores, mutexes and condition variables which support priority inheritance are a part of the
2. Priority Server

The priority server is a stand-alone RPC program. Although theclient and server stubs and the server itself are not part of the Real-Time C++ infrastructure library, there is a class within the infrastructure library to interface with the server.

Any thread that is created using the Infrastructure Library and that requires a priority other than the system default will have its priority calculated and assigned by the Priority Server. This calculation is based on the periodic rate at which the thread runs as well as various system wide parameters. Technical details and interface information on using this server can be found in the Infrastructure Implementation Appendix.

Processes and threads that do not use the Infrastructure Library may use the server directly.

NOTE: We assume that all users of the system will set priorities, other than system default, through the use of this server. It is essential, to ensure the schedulability of threads and processes, that all processes use the server. There is no way to enforce the use of the priority server by a process and/or threads unless the program is using the Infrastructure Library for all of its thread code. The user must never make operating system calls to change priorities either from within the code or from their terminal.

3. CPU Time Device Driver (timdrv)

The device driver, timdrv, monitors CPU usage on a per-thread basis for all threads on the system. At every clock tick (currently every 10 msecs), the driver increments a time count for the thread id that is currently running. All threads, whether created by the library or directly by the user will be timed by this driver. Threads that use the Infrastructure Library will get CPU usage counts (in ticks) by querying the driver. We need to further investigate whether the granularity of this timing method is fine enough for our purposes or that this method of timing CPU usage is accurate. This will be examined during fiscal year '96.

2.4 REAL-TIME INFRASTRUCTURE LIBRARY CLASSES

NOTE: At present, there are three groups of classes: miscellaneous, threads, and synchronization. The Interface, interaction, and usage of the classes are described in the Infrastructure Implementation Appendix.

The terms base class and abstract base class which are used below are two variants of classes which may be inherited by other classes. A base class contains functions and data which are completely defined and although the class may not be complete, may be used directly. An abstract base class is one in which one or more member functions are declared but not defined. This forces any classes which derives from them to implement a function with the exact name and calling sequence.

2.4.1 Miscellaneous Classes

EVENT_SIG

Class EVENT_SIG handles all POSIX user defined events to be sent to processes or individual threads. This does the following:
- Setting up POSIX events
- Waiting for an event
- Sending an event

MUTEX_ATTR

Class MUTEX_ATTR is the base class from which the standard Lynx mutex is derived. It handles the initialization and deletion of the Lynx data structures for a Lynx mutex. This class is not visible to the user and is to be used only by derived classes.

PTHREAD_ATTR

Class PTHREAD_ATTR is the class which handles setting up all of the attributes of a thread. It is a requirement of Lynx that all threads have a set of attributes created and initialized before the thread is created. This class will never be used directly, it is accessed only by Class THREAD. The class implements the following attribute features for a thread.
- Set stacksize
- Set inheritance policy (does the thread inherit the attributes of its parent or does it have its own).
- Set scheduling policy.
- Set priority.
- The class allows queries which will return the various values of attributes to the user.
- Deletion of the attributes from the operating systems tables when a thread exits.

**PRIORITY_SVC**
The class PRIORITY_SVC enables the user of the Real-time Infrastructure library to interface to the priority server without knowing about the RPC server or client. The user does not have to understand how RPC works, only what arguments and return values are expected by the PRIORITY_SVC class. The class supports the following priority server functions:
- Assigning priority
- Releasing priority
- Calculating priority

**THREAD_ERR**
Class THREAD_ERR is nothing more than a place holder that will be replaced when a more robust fault tolerant error handler has been designed. At this point in its development it accepts a defined constant describing the type of error and whether the error is fatal or not. The class prints out an error message and exits or returns. During the next phase of development the implementation will be changed to handle errors in a manner appropriate for Real-Time threads and fault tolerant systems.

### 2.4.2 Thread Classes

Figures 2-2 and 2-3 illustrate the thread classes. A description of these classes is given below.

**THREAD**
Class THREAD is an abstract base class which can be used only when defining a new class which is derived from it. This class is the highest level class from which all thread class types are derived. The class supports the following features:
- POSIX thread creation
- Thread Exit
- Cancel Properties: Ability to cancel thread synchronously or asynchronously. Synchronous cancellation allows the thread to prohibit cancellation while it is executing critical sections of code.
- Query thread’s CPU time
- Push and pop cleanup routines on the threads stack which can be executed before the thread exits.

**EVENT_THREAD**
Class EVENT_THREAD is the abstract base class for all threads which require and are controlled by an event handler. This allows a thread’s execution to be controlled by events sent to it by other processes or threads. In addition to the attributes inherited from threads, this class supports:
- Creating an event driven thread
- Sending an event to a thread.

**TIMED_THREAD**
Class TIMED_THREAD is the basic class of threads used in the real-time infrastructure library. It may be used as implemented, or it may be used as a base class. In addition to the functionality of a basic thread it encapsulates the properties of real-time constraints and deadlines. The thread manages a series of thread instances which perform the required actions on a periodic basis. The priority server assigns a priority to this thread based on its period rate. Optionally, it can implement a maximum time that each thread can run before it is forced to exit. Before the thread is forced to exit it is required to cleanup any locked resources that it may be holding. The class supports the following:
- Creation of a periodic timed thread.
- Creation of a periodic timed thread with an execution deadline.
- Specification of an abort routine if a thread has missed or will miss its proscribed deadline.

**TIMED_CHILD**
Class TIMED_CHILD is a thread class that can only be created by a TIMED_THREAD instance. This creates an actual periodic thread which is constrained by the periods and deadlines given to the TIMED_THREAD at creation. The abort routine specified to the TIMED_THREAD is used when this thread misses its deadline.
to ensure that all system resources are freed before the thread exits. Without an appropriate abort routine the thread might leave synchronization objects locked, causing the system to behave unpredictably.

### 2.4.3 Synchronization Classes

Figure 2-4 illustrates the synchronization class hierarchy. A description of the classes are given below.

**SEMAPHORE**

Class SEMAPHORE is an abstract base class from which standard Lynx counting semaphores and Lynx fast (priority inheritance) semaphores are derived. This class contains the virtual functions of wait and post that all semaphores must have. This class can never be directly used by the programmer.

**FA_OBJ**

Class FA_OBJ is another support class found in the library. This class does the set up in the Lynx data structures for Lynx fast semaphores, mutexes, and condition variables used to support priority inheritance.

**C_SEMAPHORE**

Class C_SEMAPHORE supports standard Lynx counting semaphores. These are ordinary UNIX semaphores which allow an initial count to be set when the semaphore is created. In addition to wait and post, which it inherits from class SEMAPHORE, it adds the functionality of try_wait. Try_wait allows the thread to do something else if it cannot get the semaphore.

**PI_SEMAPHORE**

Class PI_SEMAPHORE supports priority inheritance semaphores for use within a single process. They appear to the user as ordinary semaphores but have the property of priority inheritance to prevent cases of priority inversion on the semaphore.

**BIN_SEM**

Class BIN_SEM supports named priority inheritance semaphores which are capable of use across processes running on the same CPU.

**MUTEX**

Class MUTEX supports standard Lynx mutexes. The mutex will work within a process. Unlike the semaphore, a mutex has the concept of an owner. Only the thread which locked a mutex may unlock it. Care must be exercised when a thread is killed or does a normal exit that the synchronization objects that it holds have been released.

**PI_MUTEX**

Class PI_MUTEX supports priority inheritance through inheritance of FA_OBJ and works in a similar fashion to the PI_SEMAPHORE. It cannot be released except by the thread that locked it. This may be used in conjunction with the PI_COND_VAR.

**PI_COND_VAR**

Class PI_COND_VAR supports condition variables, which must be used in conjunction with a PI_MUTEX. It supports condition variable functions such as:
- Wait
- Signal
- Signal and Unlock the associated mutex
- Broadcast
- Broadcast and Unlock the associated mutex
- Lock associated mutex

### 3. DATA MANAGER IMPLEMENTATION

#### 3.1 APPROACH

A real-time data manager manages persistent and temporal data, ensures that the queries and transactions meet the timing constraints, enforces temporally correct serializable schedules, and combines techniques from data management and real-time scheduling.

Our goal was to implement a two-level data manager; shared memory data manager to manage the shared memory database needed for real-time applications; persistent storage data manager to manage persistent database. The persistent database is larger than the shared memory database and is needed for long-term storage. Interaction between the two data managers is as follows: The shared memory data manager manages the data for the operation of the application subsystem. If data is not available, the shared memory data manager communicates with the persistent storage data manager to retrieve data. Caching data may be performed in advance. At every interval, the shared memory data manager sends all updates to the persistent storage data manager. Figure 3-1 illustrates the two data managers.

Our initial plan was to use ZIP-RTDBMS for both data managers and make the necessary changes. However we found that implementing transaction management and concurrency control with ZIP-RTDBMS could not be accomplished within the...
resources allocated to the project. Therefore, we used ZIP-RTDBMS as a persistent storage manager and encapsulated it as a C++ class. This implementation is discussed in section 3.4.2. We implemented the shared memory data manager. This implementation is discussed in sections 3.4.3 - 3.4.6. In particular, the implementation of the data manager and the associated C++ classes are discussed. Note that we have not implemented the interaction between the shared memory data manager and the persistent storage data manager.

Figure 3-2 is a representation of the current Data Manager implementation design. The unshaded boxes are those portions which have been implemented or obtained as a COTS product.

3.2 ENCAPSULATED Zip-RTDBMS

Zip-RTDBMS is a COTS Real-Time database management system. Typically a user acting as a server will create a database for use by themselves and other client users who may connect to the database. A database is instantiated by inputting a schema file into Zip which specifies all relations in the database. Queries are written using Zip 'C' functions and compiled. All data manipulation language (DML) update and select queries must be written in post-fix form. The encapsulation was done to make Zip more user friendly and evolvable. The encapsulation consisted of C++ wrapper classes and an SQL to Zip parser.

The Zip Wrapper Classes

Three C++ classes were implemented to hide various Zip features.

1) Class Zip_rtdbms contains all of the DML functions which Zip offers, such as insert and select. Each wrapper function corresponds to a Zip function, i.e., wrapper function Zip_rtdbms::insert_into calls the Zip function insertDB. Each wrapper function also handles errors returned by the Zip functions, which simplifies the users programs. This class also establishes the connection to the Zip server database when a Zip_rtdbms object is instantiated.

2) Class relation contains and initializes all of the attributes of a given Zip relation. The user creates a Zip_rtdbms object which has as members, relation objects that are contained within the database. Since the database is instantiated with the schema file, a client or server may elect to only specify those relations specific to their current queries.

3) Class attr_t<class Type> hides the Zip information required for each attribute (column) in a given relation. A user of a Zip database is required to specify four fields for each attribute, the connection id of the database, the character name of the column in the relation, a pointer to the actual variable being used as the buffer, and a pointer to a structure containing the size and type information of the variable. Class attr_t<class Type> contains all of the information that Zip needs for a users buffer. All that is required is to instantiate an attribute of the required type, i.e., attr_t<int> column1_buffer.

To use these classes a server or client would derive their database from the Zip_rtdbms class and specify any relations that are going to be accessed by their DML, as shown below.

```cpp
Figure 3-1 Implementation Approach
```

When the stu_adv_server class is instantiated, the variable masterDBid is assigned a connection address which is used by the relation class to initialize the attributes. The class student_t is derived from the class relation and contains all attributes required by the users DML queries, as shown below.

```cpp
Figure 3-2 Functional Modules of the Data Manager
```

The SQL to Zip Parser

The C++ wrapper encapsulation hides many Zip specific details but the user is still required to express the DML in terms of Zip functions and in post-fix notation. The SQL to Zip parser enables the user to express the DML in embedded SQL and use the parser to convert the query into Zip wrapper function calls which can then be used by a C++ compiler. The parser also generates the appropriate derived relation classes from the data definition language (DDL) schema file. An example SQL query and its parsed Zip wrapper equivalent are shown below.

Example SQL SELECT query:

```sql
SELECT column1, column2 FROM table;
```

Output of previous SQL query in post-fix form:
3.3 CONCURRENCY CONTROL

Before discussing the other major components of the Data Manager, it is necessary to discuss the various techniques that are and will be used to manage multiple executing transactions. The following classes make use of the basic infrastructure concurrency control classes, which include the class BIN_SEM and the class PI_COND_VAR.

Preemptable Mutex / Condition Variable Class

Derived from the infrastructure PI_COND_VAR class, the preemptable class PREEMPT_PI_MUT_CV allows a transaction to be interrupted and aborted without leaving any data structures in an inconsistent state. It provides the ability to lock and unlock a mutex, which may be used in conjunction with a condition variable. A condition variable is used to signal other users of the mutex that a particular condition has been satisfied and that the mutex is available for use. After a mutex is unlocked or a condition variable is signaled, a test is done to find out if an abort signal was issued to the transaction while the mutex was locked. The test function is a built-in LYNX feature. Only one user may lock a mutex at any time.

Readers / Writer (Latch) Class

This class allows many readers or one writer to lock an object. It can use either the binary semaphore class or the priority inheritance mutex / condition variable class, which are both available in the Real-time infrastructure class library. When a user performs a read lock, a count is incremented to indicate a reader is accessing the guarded data structure. More readers are allowed to access the data structure, but writers must wait until the reader count is zero. Only one writer is allowed access at any time. This class also provides the ability to upgrade a lock from a read to a write.

Semantic Locking Mechanism (SLM)

This implementation was acquired from the University of Rhode Island and ported from the Solaris operating system to Lynx OS version 2.2.2. The SLM allows object locking on the method level, affording more concurrency than basic object read/write locking. The SLM is actually a family of C++ classes which operate together. Each object must be specified in a predetermined manner, deriving from a Base class. Objects must use the Attr class to specify each attribute that will be manipulated and accessed by users of that object. Each method of the object will access the attributes through the Attr member functions Read and Write.

The object is parsed to determine the read/write affected sets for each method. The read affected set is the set of attributes read by a method and the write affected set is the set of attributes written by a method. From the read/write affected sets, a compatibility matrix is defined which identifies which methods can run concurrently with other methods of the same object. Each method pair has a compatibility function which is used to determine if the methods are compatible. The user is allowed to manipulate the compatibility functions, therefore creating specific semantics under which the methods may run.

Read/Write Affected Set Priority Ceiling Protocol

This algorithm is an extension of the Basic Priority Ceiling Protocol and the Read/Write Priority Ceiling Protocol. Both of these protocols operate on the object locking level and provide prevention from deadlock and bound priority inversion to one lower priority transaction. Priority inversion occurs when lower priority transactions hold locks which prevent a higher priority transaction from running.

The Read/Write Affected Set Priority Ceiling Protocol provides the same benefits of the previous protocols, except that it works at the method locking level. This allows for greater concurrency than the previous protocols, while at the same time offering the same benefits of no deadlock and a bounded priority inversion of one.

Although this algorithm has not been fully implemented, the algorithm for determining the priority ceilings of the methods has been implemented in the form of a parser as long as the objects are in conformance with SLM objects (as described above).

3.4. MEMORY MANAGERS

We first describe the dynamic shared memory manager which we also refer to as the object manager. To implement the Meta Data and Transaction Managers, it was also necessary to implement some other memory managers. We also describe these other memory managers.

Dynamic Shared Memory Manager

This implementation was acquired from the University of Rhode Island and ported from the Solaris operating system to Lynx OS version 2.2.2. It allows dynamic allocation and deallocation of objects in shared memory. The SharedHeap-int> class is currently being used as the Object Manager in a limited capacity. The Transaction Manager has access to the objects through the SharedHeap, however, since there is no Storage Manager, there is no connection to the Zip Database.

To use this class a process first creates a new shared memory heap or attaches an existing one to its virtual address space. A process allocates objects using an overloaded C++ new operator. The process then registers an object id with the Shared Memory Manager (done through the Transaction Manager). Any thread running within that process can retrieve an object from shared memory if it knows the object id.

Static Memory Manager

This class was created to take advantage of the a priori information in a Real-Time system. The class may be used with or without shared memory. The user pre-allocates the desired type and quantity. Once allocated, transactions can use an overloaded C++ new operator to retrieve a block of the desired type. This class cannot retrieve an object by 'id'.

Hash Table Class

This class was needed to keep track of transaction id’s in the Meta Data Manager. The class uses a template so it will work with any user defined type. The only requirement is that the user type contain a member named 'id' which is used by the hash function.
The class may be used with or without shared memory and uses a PREEMPT.PI_MUT.CV to allow concurrent and preemptable thread access to the Hash Table structures. The class can currently only hash int 'id' values, but through inheritance, any 'id' type may be hashed.

3.5 METADATA MANAGER

This implementation was based on [BILI95]. In its current form the Metadata Manager TXN_META_DATA class is simply a storage device. Any manipulation of the metadata contained within it is performed by external classes. Currently the only friends which use the Metadata Manager are the Transaction Manager and Transaction Clean-Up class. The Metadata Manager contains three major structures for storing information:

1) The Transaction Descriptor (TD) structure contains information for a transaction. A transaction is a single thread within a process. The status of the transaction is recorded, and can be either initiated, running, committed, or aborted. The TD stores the transaction id, which is used to locate a TD in the hash table. The TD id is the same as the thread id that is executing the transaction. A pointer to the infrastructure THREAD object which was instantiated for the transaction is also stored, so when the transaction status is changed from initiated to running, the thread's function may be executed. Finally, the TD keeps a list of all locked objects for the transaction.

2) The Object Descriptor (OD) structure contains information for an object. It contains the object's id, supplied by the user, and the object's size. This structure also contains a list of granted and pending locks for transactions on the object. Although the term object is being used, it actually could refer to any lockable entity, such as a method of an object. This means that this structure could be used with any of the concurrency techniques that were previously discussed.

3) The Lock Request Descriptor (LRD) structure contains information for a lock on a particular object. This structure contains a pointer to the TD structure of the transaction who owns the lock, a pointer to the OD structure of the locked object, whether the lock is a read or a write, and whether the lock is granted, pending, or released. This structure also contains a shadow copy of the locked object, which is used by the transaction until that transaction is committed.

The Meta Data Manager may be used with or without shared memory. This class uses the Static Memory Manager class to preallocate all of the required structures in shared memory and the Hash Table class to locate the TD structure for a particular transaction. It also contains a PREEMPT_PI_MUT.CV class to allow concurrent and preemptable thread access to the Meta Data Manager structures.

3.6 TRANSACTION MANAGER

This implementation was based on [BILI95]. Through the TXN_MAN class transactions manipulate the Metadata Manager to log their lock requests. This class supports both periodic and non-periodic Real-Time infrastructure threads.

Periodic Threads

The TIMED_TXN_THREAD class is derived from the infrastructure class TIMED_THREAD. It has as a TXN_CLEANUP class as a member.

Non-Periodic Threads

The TXN_THREAD class is derived from the infrastructure class THREAD. It has a TXN_CLEANUP class as a member.

Transaction Clean Up

Since both types of threads need to deallocate Meta Data Manager structures when they are destroyed, the TXN_CLEANUP class was made to centralize the destructor function without having to use multiple inheritance, which seemed unnecessary for this simple function. When a periodic or non-periodic thread is deleted, the destructor for the TXN_CLEANUP class is called to deallocate any Meta Data Manager structures still being used by the transaction.

Use of Transaction Manager

Transactions are 'initiated' to allocate resources. At this time the appropriate thread class is instantiated, and the TD structure is placed in the hash table. If initiated, the transaction is executed by a call to the function 'begin'. The transaction manager changes the transaction's status to running and starts the threads function. The transaction manager presently issues read and write locks on objects stored in shared memory. Deadlock is avoided by explicitly ordering the locks. This could be changed in the future by implementing the read/write affected set priority ceiling protocol to allow locking on the method level with no deadlock.

The Transaction Manager uses two-phase locking to maintain serializability. The user may make as many locks as they wish, but cannot make any more locks after a lock has been released. The locks may be released one at a time throughout the transaction and/or released all at once when a commit or abort is performed. Releasing the locks individually allows more concurrency with incompatible transactions.

When a transaction finishes executing it can commit the changes to the database and release any remaining locks. If a periodic transaction does not meet it's deadline, a signal is issued and the transaction is aborted. An abort releases any locks the transaction may have had.

Another function related to the commit/abort functions is the 'wait' function. The function wait returns a true value if the transaction in question commits. It may be used to express dependencies between transactions. A transaction will wait to commit until a certain transaction or group of transactions commits. Depending on the logical expression used, any number of dependencies may be expressed. For example:

Transaction self will commit if (T1 commits OR (T2 AND T3 commit).
ORB will be a glue for the infrastructure, data manager, and the Request Broker (ORB) to be included in the implementation. We have described some preliminary ideas in our work. CORBA (Common Object Request Broker Architecture) is a standard for distributed object management systems such as OMG's (Object Management Group) CORBA. We are also investigating real-time issues for CORBA and will make use of the data manager for real-time data management.

Our current work is proceeding in the following directions. We are investigating object-oriented implementation of the infrastructure and data manager for real-time command and control systems. In addition to the work described in this paper, we have also carried out a detailed design of the infrastructure and data manager which includes several additional features not described in the implementation. One of the major contributions of this work is a concurrency control protocol which incorporates the semantic locking mechanism described in [WOLF94] with the priority ceiling protocol, described in [RAJK89]. We have also completed an object-oriented implementation of the MSI environment. Details of the design and implementation are given in [BENS95].

Our current work is proceeding in the following directions. We are porting the application implemented in the SGI environment to the Lynx platform. The application will be hosted on the infrastructure and will make use of the data manager for real-time data management. We are also investigating real-time issues for distributed object management systems such as OMG's (Object Management Group's) CORBA (Common Object Request Broker Architecture). We have described some preliminary ideas in [KRUP94]. Finally, we are investigating the use of an Object Request Broker (ORB) to be included in the implementation. The ORB will be a glue for the infrastructure, data manager, and the application.

4. CURRENT STATUS

In this paper we have described our object-oriented implementation of the infrastructure and data manager for real-time command and control systems. In addition to the work described in this paper, we have also carried out a detailed design of the infrastructure and data manager which includes several additional features not described in the implementation. One of the major contributions of this work is a concurrency control protocol which incorporates the semantic locking mechanism described in [WOLF94] with the priority ceiling protocol, described in [RAJK89]. We have also completed an object-oriented implementation of the MSI application. This implementation was carried out in the SGI [SGI] environment. Details of the design and implementation are given in [BENS95].

Since the data resides in shared memory, if the system were to lose power, the data would be lost. This situation will be remedied when a persistent store is added to the Data Manager. Another situation of failure is when the process running the transactions is terminated abruptly. When this occurs, a single process can be started to traverse all active TD structures and finish aborting and committing based on the current state of the TD status flags. Once the objects have been placed in a consistent state, the TXN_META_DATA may be reinitialized to reset the structures and replace any mutexes that were left in a locked state.

REFERENCES


ACKNOWLEDGMENTS

This work was carried out at the MITRE Corporation during FY95 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 Thomas 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 also thank Larry Fisher, Gary Gengo, Jim Houckens, Arkady Kanevsky, Soohee Kim, and Alice Schaefer for their inputs into this project. Finally, we thank Prof Victor Wolfe of the University of Rhode Island and his real-time systems group (John Black in particular) for many useful suggestions, comments, discussions, and also for supplying the code for part of the data manager implementation.