Turk J Elec Engin, VOL.10, NO.1 2002, © TÜBİTAK

An Efficient Middleware Architecture Supporting Real-Time Distributed Object Programming

Erhan SARIDOĞAN
Turkish Navy, Software Development Center
Arastirma Merkezi Komutanligi
81504, Pendik, Istanbul-TURKEY
e-mail: esaridogan@yahoo.com
Nadia ERDOĞAN
Computer Engineering Department
Electrical-Electronics Engineering Faculty
Istanbul Technical University
80686, Ayazaga, Istanbul-TURKEY
e-mail: erdogan@cs.itu.edu.tr

Abstract

With the increasing demand for distributed real-time systems, the need for programming tools and execution platforms useful in development of such application systems is widely recognized. This paper presents CORD-RTS, an efficient middleware architecture that provides support for real-time distributed object programming. The communication infrastructure and various components of the middleware, which support several modes of interactions among distributed real-time objects, along with its real-time features and services, are discussed in detail.

Key Words: *real-time, middleware, distributed object programming, object interactions*

1. Introduction

Real-time systems have become increasingly important in a growing number of application domains, such as telecommunication networks, multimedia, military command and control, manufacturing, and finance. The tasks in a real-time system must produce logically correct results by their deadlines. The correctness of such a system depends not only on the logical correctness of the result but also the tasks being completed by their the deadlines. That is, real- time systems require that deadlines of all jobs be met. However, for many applications, this is an overly stringent requirement. An occasionally missed deadline may cause decreased performance, but is, nevertheless, acceptable. Such real-time systems that may miss some deadlines are called "soft real-time" to distinguish them from "hard real-time" systems where all deadlines must be met to ensure the safety and correctness of the system.

Distributed real-time applications are those in which there are end-to-end timing constraints across a distributed system. Unfortunately, highly skilled programming is required to achieve real-time performance in such systems. The challenge is to develop *mechanisms* that allow a distributed real-time system that

meets its deadlines to be built on standard hardware and operating systems. The operating system API is typically an inflexible procedural interface that addresses a single machine's requirements. Its use usually limits evolutionary development and complicates application design for distributed real-time systems. Recent decades have shown that technology evolves faster than application requirements. Technological improvements in hardware lead to changes in operating system functionality and these changes must be reflected in their API. However, the adoption of existing software to a new API is a difficult task. Therefore, real-time programmers need execution platforms that isolate them from the underlying hardware and software, to develop real-time applications.

CPL (Cord Programming Language) [1], [2], [3], is the language we propose for real-time distributed object programming. CPL aims to reduce the difficulty in developing real-time systems and allows distributed real-time programs to be designed and tested as easily as single sequential programs, independent of underlying architecture. CPL is based on the object oriented programming paradigm with enhanced object behavior. Periodic method invocation, iteration statements with timing constraints, specification of deadline both at the statement level and on object communication, and an advanced exception control mechanism constitute the high-level at high precision real-time attributes the CPL provides, which contribute to the development of reliable real-time applications.

CORD-RTS (Concurrent, Object-oriented and Real-time Distribution Run-Time System) provides execution support for CPL applications. It is a middleware architecture implemented on a multi-node computer network of standard hardware and operating systems (Sun OS 4.1.3 and Solaris 2.5). Its place in the software abstraction hierarchy is between a CPL application and the operating system, providing an interface for which the CPL compiler generates code. It provides communication infrastructure, naming mechanisms, remote method invocation facilities, and real-time features and services are essential for the object-based architecture on which CPL applications execute.

CPL and its supportive middleware CORD-RTS is applicable to a broad class of real-time systems. Among these, "Distributed Command and Control Systems" are especially of interest to the authors and have closely guided the design choices of CPL real-time features. Both industrial and military command and control systems require an efficient infrastructure capable of handling large amounts of high frequency data. Automated control systems collect data from various sensors or input devices, evaluate them and remotely control some actuators, preserving real-time constraints. In addition to efficient device control, fast and reliable access to shared data is also required. As the target domain of our work includes systems with dynamically changing resources and loads, off-line analysis techniques cannot be applied to make definitive claims about deadlines; therefore CPL applies best to soft real-time systems. The static priority based on-line scheduling scheme of the underlying operating system, coupled with CORD-RTS time management mechanisms, insure the fulfillment of soft real-time requirements.

This paper is structured as follows: Section describes the related work, Section gives background information on CPL and also presents CORD-RTS with its goals and architecture. Section 3 and Section 4 describe the CORD-RTS communication infrastructure and components. Section 5 elaborates on inter-object communication, while Section 6 gives details on the implementation of real-time features.

2. Related Work

While the field of object-oriented real-time programming is young, it is growing quickly because it offers a wide range of applicability, from complex real-time systems to the next generation of computing and communication devices [4]. Java, an object-oriented programming language, is highly suitable for extension to

real-time and embedded systems. The Real-Time for Java Experts Group (RTJEG) is working on developing a real-time specification for Java (RTSJ) [5]. A similar initiative has come from the real-time CORBA community. In [6], the authors discuss the Real-Time CORBA approach to defining quality of service attributes for distributed objects. Another approach would be defining an API for real-time distributed objects. The TMO model [7] proposed for real-time object programming is supported with such an API. We may expect middleware supporting distributed real-time objects based on DCOM [8] architecture become available in the near future.

3. CPL

The CORD System is a framework that has been designed to allow the development of real-time distributed object programs in CPL independent of computer hardware, operating system and the network topology [1], [3]. Application software developed in CPL can easily be ported to a new environment by modifying only the parts of CORD-RTS that are in close interaction with the operating system. Therefore, CORD-RTS acts as a middleware for the software architecture as shown in Figure 1. It is also possible to access the underlying operating system through system calls or CORD C++ libraries.

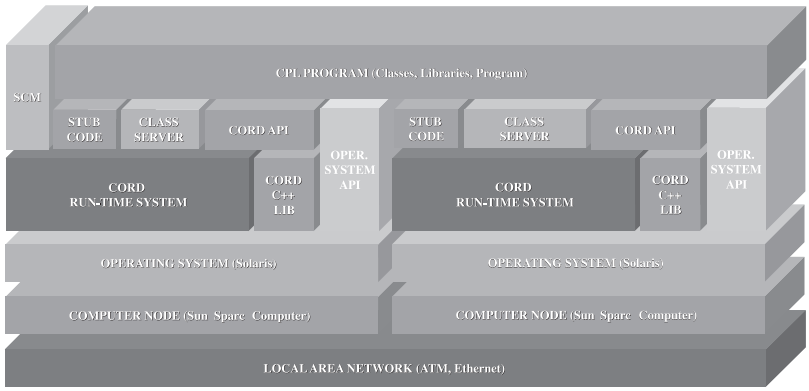

Figure 1. Software Architecture

CPL allows the construction of object-oriented programs without regard to traditional object boundaries, such as address spaces and location of objects in a distributed system. It introduces three new types of classes in addition to regular C++ classes: *active*, *passive* and *device* classes, from which active, passive and device objects are created respectively.

***Active class*:** Object instances of this class are the primary computational units of a distributed application, capable of calling methods of other CPL objects. An active object, in contrast to a passive object, is associated with an independent thread of execution, which is implemented by a Unix process in CORD-RTS. Methods of remote active objects are called transparently through special messages generated by the CPL preprocessor.

***Passive class*:** Object instances of passive classes are used as a means of fast access to data shared among active objects. They perform storage management of and provide fast access to a shared data store located on the main memory of a node. The data store of a passive object is implemented as a segment of Unix shared memory. Passive objects act as storage servers and are not capable of calling methods of other objects. Only active objects located on the same node can access these objects for very fast read or write operations to shared data.

***Device class*:** A device class defines a standard and coherent interface for input and output devices. The class definition specifies the mode of access, priority, data type, buffer size and speed of data transfer.

A number of classes and a main module constitute a CPL program. The main module declares the objects, and registers itself and all its classes to CORD-RTS during elaboration. With the start command, the application begins to execute.

4. RTS Communication

Interaction among middleware components and application program elements is supported by a high level communication mechanism provided by CORD-RTS. The system uses basic Unix IPC facilities in order to provide an efficient, fast and reliable medium for communication. Inter-object communication is established by means of messages, while inter-node communication uses datagrams.

Message structure: The unit of inter-object communication is a message that has a standard header part and a variable length data part of maximum 1000 bytes. The header part of each message contains certain fields for program, source and destination identification, message type, size, priority, time of initiation and timeout value. The data part of a message is interpreted according to the message type and contains parameters relevant to the message request. For example, a message instance of ACTIVE_OBJECT_METHOD_CALL carries "*method_id, parameter_length, parameter_list*", while a message instance of type CMD_ADD_NODE carries "*node_name, net_address*" information in the data part.

Datagram structure: Inter-node communication is through UDP datagrams, executing a connectionless protocol to achieve fast data transfer with broadcast option. A datagram consists of a header part and a data part. The header part contains fields for source node identification, size, number of messages and time of initiation. The data part contains a variable length byte string carrying a number of messages. CORD-RTS limits data length to 5000 bytes to speed up data exchange and processing.

The basic components of the communication mechanism are the *Object Output Queue*, *Object Input Channels* and the *Passive Data Storage*. Network Channel and Error Channels are simple socket-type communication mechanisms. Figure 2 depicts the communication between program elements and the middleware components with the implemented messages.

4.1. Object Output Queue

Object Output Queue is a Unix message queue that is created by the Object Manager. It has a predefined name and identification that CORD-RTS and CPL program components recognize and connect to, in order to set up a unidirectional communication with the Object Manager. Even though the message queue is maintained as a FIFO list, CORD-RTS makes use of the *msgtype* argument to convert it into a *priority_queue*. Every message on the queue has a long integer type attribute. A message is read from the queue using the *msgrcv* system call, where the *msgtype* argument specifies which message on the queue is desired. If a message of a certain type is not available, the reader blocks on the queue. A *msgtype* of zero specifies that the oldest message on the queue is to be returned. On the other hand, if *msgtype* is less than zero, the first message with the *lowest* type value that is less than or equal to the absolute value of *msgtype* is returned. CORD-RTS treats the priority values of messages as their *msgtype* attribute and the Object Manager attempts to read messages with a *msgtype* value of "-1", which always returns the first message with the lowest *msgtype* value, corresponding to the "oldest message in the queue with the highest priority". Figure 3 shows the internal structure of the Object Output Queue and Passive Data Storage.

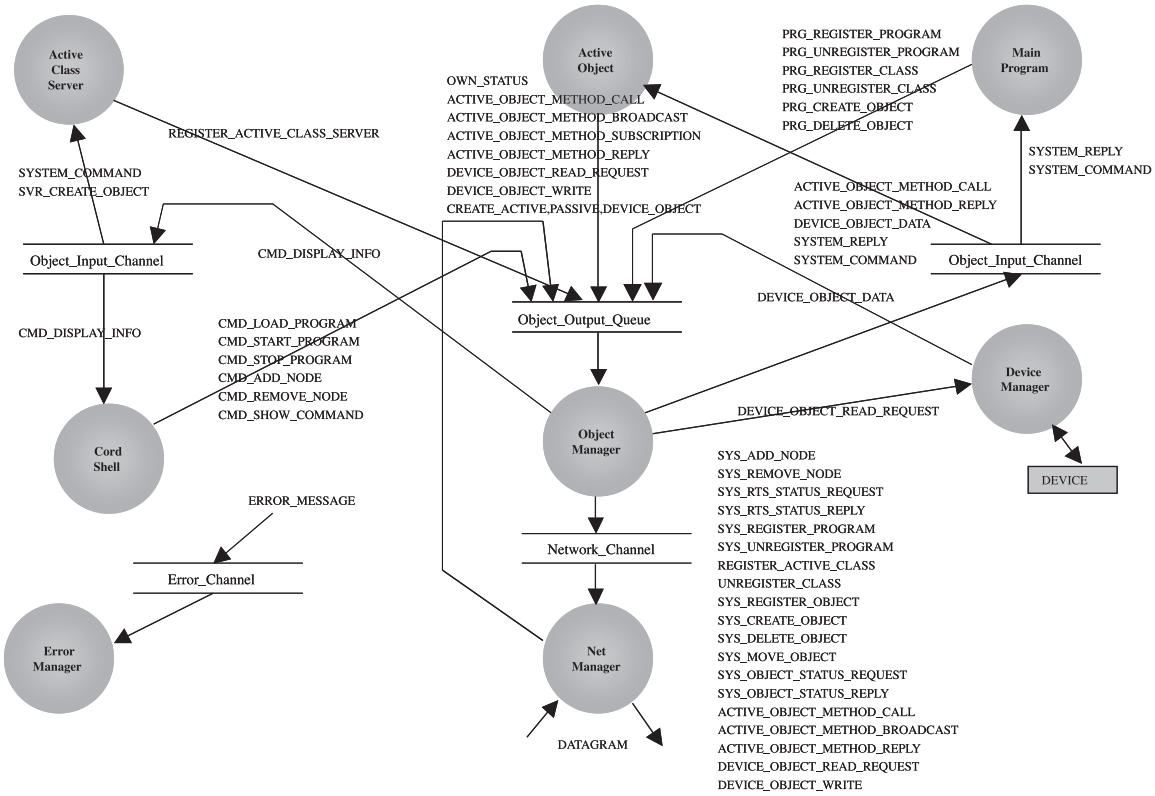

Figure 2. CORD-RTS Message Communication Infrastructure

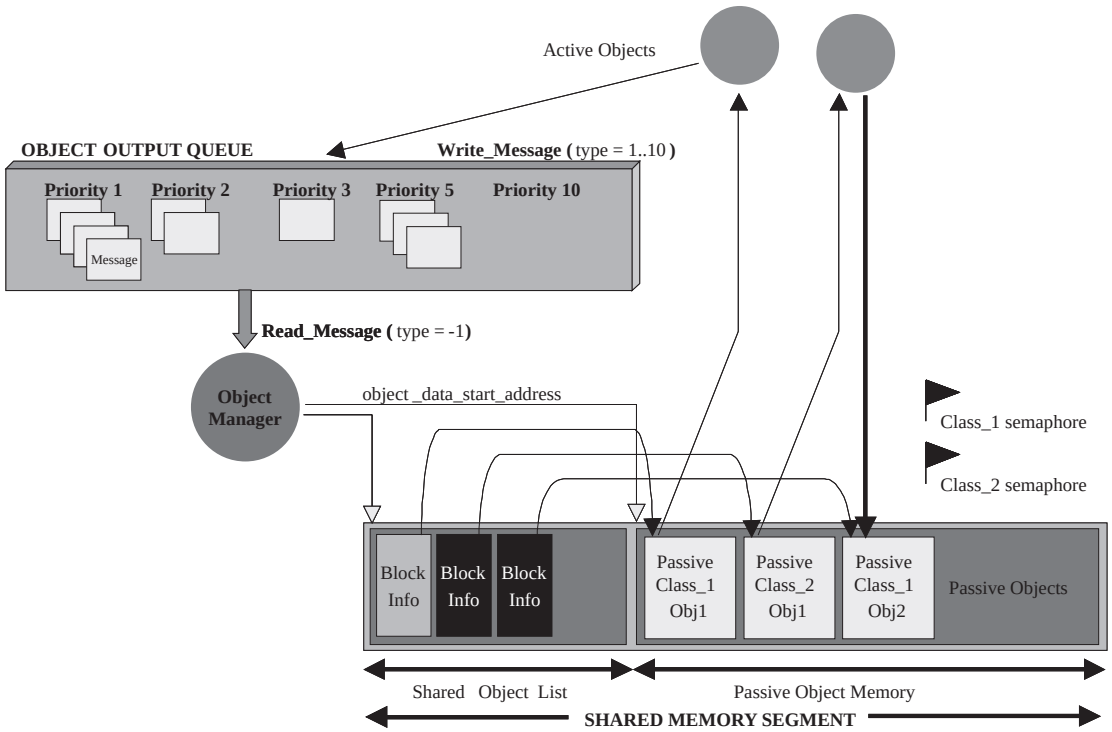

Figure 3. Object Output Queue and Passive Object Storage

4.2. Object Input Channel

Object Input Channel acts as the communication address of an active CPL object. A newly created active object adds its process id (pid) value to a constant, currently 5000, to obtain a unique port number and opens a UDP socket for local input. The active object also registers itself to the Object Manager supplying its pid. The Object Manager, receiving the registration information, opens a socket for output with the same parameters and directs its messages into this socket to communicate with the active object. Active objects receive all kinds of method invocation or reply messages through their Object Input Channels.

4.3. Passive Data Storage

CORD-RTS allows active objects on a node to communicate through shared memory, thus preventing excessive data copying through messages. This approach optimizes memory management and speeds up processing. Passive objects are instances of passive classes encapsulate shared data, whose implementation relies on Unix IPC Shared Memory. During the initialization of a CORD node, the Object Manager creates a shared memory segment and organizes it into two parts. The first part, the *Shared_Passive_Object_List*, is an array of records, each containing information about the program, class and object identifications, start address and in-use condition of a passive object. The second part is a set of memory blocks reserved for passive objects. When the Object Manager receives a request to create a passive object, it allocates the first free memory block of size specified in the passive class definition and places the relevant information into the Shared_Passive_Object_List (Figure 3).

When a passive object is created (or bound to an already created one), the caller attaches itself to the shared memory segment, locates the target passive object in the Shared_Passive_Object_List, gets the start address of the memory block allocated for that passive object and connects its local pointer to this address. When method calls for this object are issued, the caller performs memory operations without any context switch.

Each passive class is associated with a Unix semaphore to provide mutually excluded access to its instances. The *CPL_LOCK* and *CPL_UNLOCK* macros provided to the CPL programmer issue *Acquire()* and *Release()* functions of the class semaphore, respectively, for appropriate action. Thus, the programmer can enforce mutual exclusion through these primitives if necessary. It should be noted that using these primitives increases data access time from the nanosecond level to a few hundred microseconds.

5. CORD-RTS Components

All active components of a CPL program communicate with CORD-RTS, which provides network transparent inter-object communication with real-time constraints. CPL objects invoke methods with input and output parameters and are not aware of the underlying message exchange. CORD-RTS consists of five components handling the core functionalities of the middleware: the *Object Manager*, the *Net Manager*, the *Device Manager*, the *Error Manager* and the *Shell*. Interaction among middleware components and application program elements is supported by a high level communication mechanism provided by CORD-RTS. Each node on the distributed system runs a copy of CORD-RTS, whose architecture is illustrated in Figure 4.

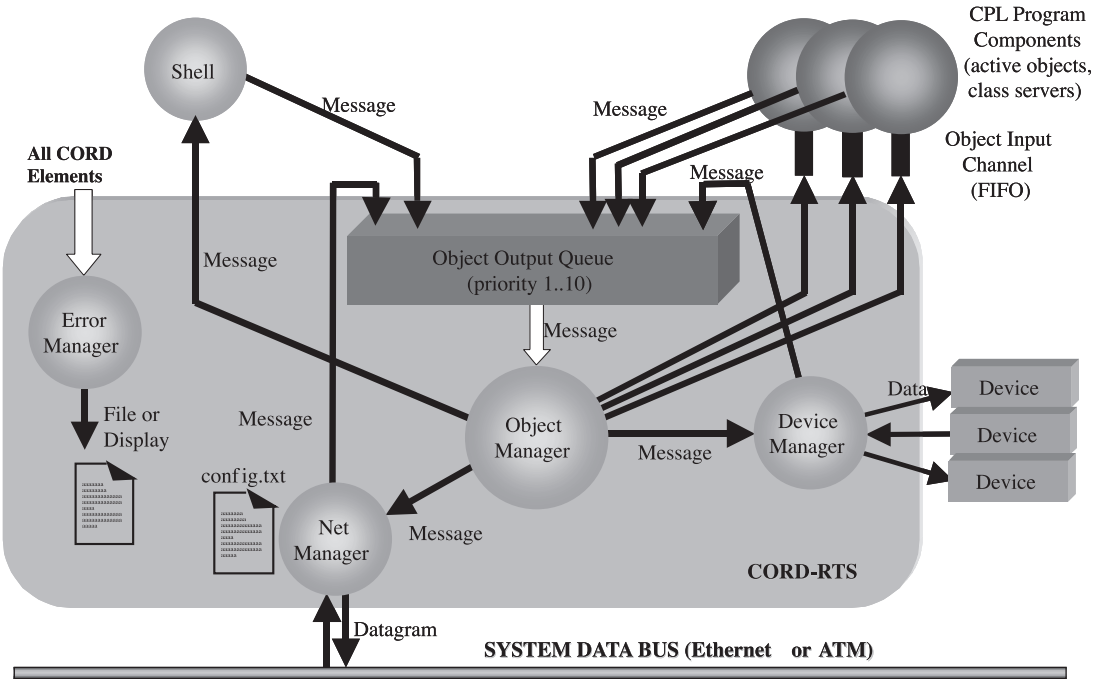

Figure 4. The CORD System Architecture

5.1. Object Manager

The Object Manager is explicitly started on each node of the distributed system and acts as the main system controller on its node. It manages the activities of object servers and running programs, routes messages to appropriate destinations and executes commands from the user through a shell interface. On start-up, the Object Manager creates communication channels and initiates the other manager components of CORD-RTS. It first creates the Object Output Queue, its input channel through which it receives messages from all other components. Next, it creates the dedicated output channels, the Network Channel and the Device Channel, for communication with the Network Manager and the Device Manager, respectively. The initiation of the nodal Passive Data Storage, the Network Manager and the Device Manager completes the start-up phase.

The Object Manager maintains a data structure called the program-table for application programs and keeps it up to date at all times throughout the CORD network. Each record on the program-table points to separate lists of active, passive and device class constituents of a CPL program. When a new CPL program is loaded onto one of the nodes, it registers all its classes to the local Object Manager, which then broadcasts this information to all nodes, requesting all other Object Managers on the system to update their program-tables. When a new object is created, the Object Manager inserts a record containing its identity and addressing information into the list maintained by the particular class of which the object is an instance. Thus, the program-table, which is replicated at each node, is kept consistent, providing a backup facility for fault-tolerance.

An Object Manager is initiated either as a Master or as a Slave, as there has to be a master manager in a system of several nodes. The master manager is responsible for program-table updates should a new node be added to the system. If a manager receives a system message requesting for a table update, it responds to this message only if it is the master. Since this request message is broadcast, the managers need not recognize each other.

After start-up, the Object Manager takes appropriate action on one of the following request types:

- Registration/unregistration of programs, classes, class servers, objects
- Creation/deletion of active, passive or device object instances
- Active object method call, reply, publish/subscribe
- Device object read/write request, publish/subscribe
- Generation of system reply messages to active objects
- System commands:
  - Add/remove node
  - Register/unregister RTS
  - Terminate RTS
  - Send/receive RTS status
  - Update look-up table
  - Load/start/stop/restart program

The Object Manager checks the initiation and time-out value of a message before starting to process it. For this purpose, a predefined system parameter is used to indicate average round-trip communication and processing delay. If the difference between the current time and the message initiation time is lower than the time-out value after subtracting the round- trip delay, the manager accepts the message. If the computed value exceeds the deadline, the manager rejects the message and sends a system reply message to its originator.

The Object Manager also acts a message router among components of a CPL program and CORD-RTS. To redirect a message, it first locates the destination object on the program table. If the addressing information points to a local object, then the message is written to the object's Object Input Channel. Should a remote object be the destination, the addressing information is placed in the message header and the message is directed to the Net Manager.

The Object Manager maintains a list of subscriber objects located on the node, controls the publish-subscribe mechanism and distributes incoming data to each of the interested subscribers. When it receives a published message from a local object, it scans the Subscriber List for that published data to locate a subscriber object and forwards a method call message for its registered method with the published data as the input parameter. This process is repeated for all subscriber objects on the list. The Manager keeps the subscriber list up to date by inserting new subscriptions upon request and removing existing ones when the subscriber objects terminate.

If an object is to be moved from one node to another, a hand-shaking takes place between the Object Managers of the source and destination nodes. After the old object status is marked as a moving object, a new object representation is created on the destination node and a new object is created from the local active class server. Next, the old object process is terminated and its representative data is removed from the program table. Meanwhile, incoming messages for that object are buffered in the Object Output Queue of the source node and as soon as the old object is deleted, these messages are forwarded to the destination

node. It is the programmer's responsibility not to move active objects that use passive objects. Special application design has to be made to provide passive object support to moved active objects.

The Object Manager also maintains a list of Command Shell representations that are activated by the user. When a shell is started, it registers itself to the Object Manager in order to open a communication channel using its pid. When the shell is terminated the manager clears its registration.

5.2. Net Manager

The Net Manager basically handles node-to-node communication on the network. It directs the transfer of system messages that are exchanged among the RTS running on different nodes to support interaction among CPL objects. It also directs messages coming from the network to the Object Manager.

When the Net Manager is initiated, it first reads a configuration file to obtain the logical names and network addresses of the nodes that exist on the CORD System and constructs an internal data structure called the Node-List, which is basically the list of nodes on the network. Each node is associated with a priority-based message list. The broadcast address is added by default as if it were a separate node and messages with a broadcast tag are inserted into the list pointed at by this symbolic node.

The Net Manager receives messages from the Object Manager through the Network Channel and converts them into datagrams. It is initiated with a parameter that indicates the time period in milliseconds it has to wait before packing messages into a network datagram for a specific destination. This interval is overridden for highest priority messages. Incoming messages arriving within this period of time are buffered in the message lists of destination nodes in the Node-List until the period of time expires or the maximum size of a datagram is reached. Then messages are packed into a datagram and sent through the network to each destination node.

The Net Manager also receives datagrams from other nodes through the network. They are decomposed into messages and directed to the Object Manager after checking their timeout values to find out if the remaining time is sufficient for possible processing. The sufficiency of the remaining time is a system parameter and is currently set to a constant value. However, its value is planned to be computed periodically by the Net Managers in future versions. The manager provides status information and updates its Node-List in accordance with certain system messages. Figure 5 shows the message flow through a Net Manager.

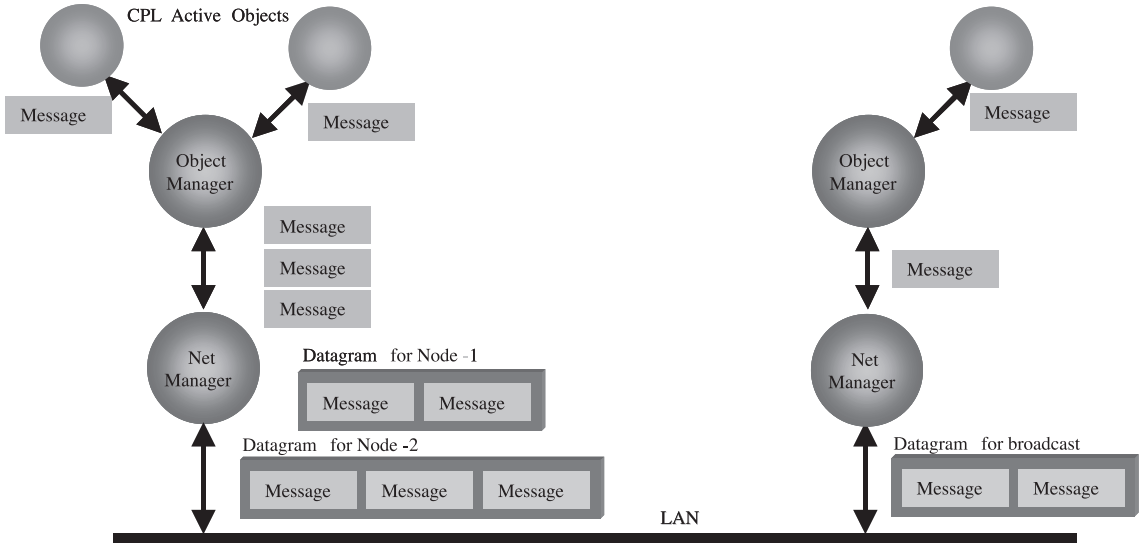

Figure 5. Message Communication Through Net Manager

5.3. Device Manager

The Device Manager controls hardware input-output devices. It activates a device through a device specific kernel driver that establishes communication between the operating system and the hardware. A system may have default devices that can be accessed via installed drivers. When a new device is to be added, its driver specific to the operating system has to be installed on the node. A high-level abstraction can be very helpful to develop software interacting with various types of hardware. The CPL device class specification provides such a high level interface for device I/O.

The Device Manager maintains a list of registered device classes and object instances located on the node. For a new registration request, a new entry is created, but the actual device is not activated unless an instance is created. The Device Manager provides access to nodal devices through their descriptors. After successfully opening a device, it sets the I/O controls according to settings specified in the class definition to prepare the descriptor. Since there may be several descriptors to be watched at the same time, the Device Manager uses the Unix "select" mechanism. A wait-list of descriptors to be read are constructed. The main execution control of the Device Manager waits blocked at these descriptors until one of them is ready. Then data is read from the device based on the specification kept in the object representation and stored in the data buffer of the associated object.

The manager receives device read or write request messages from the Object Manager via the Device Channel (see Figure 4). Each device object representation keeps a local buffer to store data along with the time of last update. There may be three types of read activity on this data buffer as specified in the request message. Aged Read returns the data in the buffer if it has been updated within the specified time. Immediate Read returns the content of the local buffer immediately. Subscribed Read adds the requesting object information into the await list and provides call back when data is read.

The manager also provides a means for a publish-subscribe mechanism. Active objects may register one of their methods to a specific device object created for device input. The manager keeps a list of registered active objects and their methods for a specific device object. When new data is read from a device, the manager checks the list and issues asynchronous-method-call messages supplying the read data as the parameter for the registered method of each active object on the list. This mechanism provides fast data distribution capability. The Device Manager can logically connect several objects to a single physical device, thus providing concurrent data retrieval, even from remote nodes.

Should the CPL device class specifications not be sufficient to control a device, a conventional C or C++ program module with appropriate device settings can be used. This module is then linked to a CPL active class to be accessed by the rest of the distributed program. If a piece of special hardware has to be used, then its driver has to be adapted within the Device Manager, which requires modification and re-compilation.

5.4. Error Manager

The Error Manager is a message collector that has to be initiated before all other CORD- RTS components. It is initiated with a command line option indicating the destination of errors messages, either a file, standard output or both. The manager simply receives error messages through the Error Channel and directs them to the specified output destination in a standard format. All elements of the CORD System, including CPL programs, report their errors in a specific format to this manager. When a program is activated, it first initializes its internal global object, *Error*, with its program, class and object name, and establishes a connection with the Error Channel. The CORD System API provides a number of routines to report errors

in a standard format, which includes a text part and a source identification through program, class and object id's.

5.5. CORD Shell

The CORD-Shell interprets user commands interactively in order to enable on-line system control and monitoring. It is started from a Unix shell by the *"cord_shell"* command, upon which it displays the prompt *"cord>"* and waits for user input. The commands fall into two classes: those that display information and those that allow users to interactively control CORD-RTS and CPL programs. Commands in the first class provide information on the system and status of programs, their classes or objects. On information requesting commands, the shell acquires and displays the relevant data. Commands in the second class allow for the interactive control and re-configuration of the system. Users can register, load, start and terminate programs; register, create, delete or move classes/objects; add, remove or change the status (master/slave) of a node; terminate CORD-RTS or issue Unix commands preceded by an exclamation mark "!".

6. Inter-Object Communication

CORD-RTS relies on remote method invocation as a basic abstraction for inter-object communication. This abstraction simplifies distributed programming by making communication with a remote object resemble communication with a local object.

Method calls are either synchronous or asynchronous depending on the parameter type declared in the class body (*in/out/inout*). Methods that return any kind of information are called synchronously, forcing the client to wait until execution is complete. Methods that do not return any information are called asynchronously, which allows a call to overlap with local computation. CORD-RTS also supports group communication through the publish- subscribe mechanism, where a publisher object implicitly issues method calls to a group as a whole, rather than having to know group membership and communicate with members one to one. Dynamic data structures and pointers must not be used as parameters in any of the communication mechanisms.

The data part of a message may have a variable length with a maximum of 1000 bytes, because this is an optimized value for Ethernet communication. In addition, typical command and control systems do not require big chunks of data, instead small pieces of frequently changing data are used in order to meet real-time requirements.

6.1. Call Processing

CPL allows method invocations on distributed objects regardless of object location. A remote object is accessed just like a local object, through the *object.method(argument)* mechanism. Objects only experience the added overhead of an indirect remote method call. Client stub and server skeleton form the interface between the application layer and the CORD communication layer. The CPL compiler generates the stub and skeleton code automatically in C++.

The client stub provides a strongly typed, static invocation interface that marshals application parameters into a standard message-level representation and transparently directs a method call request into CORD-RTS. A method call request includes associations that produce an object reference, a destination object address that uniquely identifies the target object in the CORD System network. CORD-RTS locates the appropriate end point in the distributed system and sends a message to initiate call processing. If the

destination is the local node, the message is forwarded directly to the object itself. Otherwise, the message is sent to the RTS of the remote node which, in turn, transfers the message to the target object.

The destination object may be in a blocked state waiting for messages. In this case, the new message is accepted immediately; its time-out value is evaluated to determine if there is sufficient time for processing. If the scheduling priority of the object is lower than the priority of the incoming message, then the process increases its own priority up to the message priority. After the message is processed, the scheduling priority is lowered again. For the reverse case, the object does not change its priority.

The callee may also be waiting for a specific reply message to its previous call when a new method call message arrives. In this case, it accepts the call message and puts it in a priority- based call message queue inside the object context. After the arrival of the awaited reply message or on time-out, the call messages in the priority-based queue are processed. The skeleton code de-marshals the message-level representation into typed parameters appropriate to the application and generates a call to the related internal function.

CORD-RTS active object method call is an implementation of remote procedure call (RPC). However, all parameter type conversions are handled during compile time. This speeds up processing immensely when compared to dynamic type conversion. A synchronous method call is the primary way of active object communication. This mechanism, including the CPL syntax, is illustrated in Figure 6.

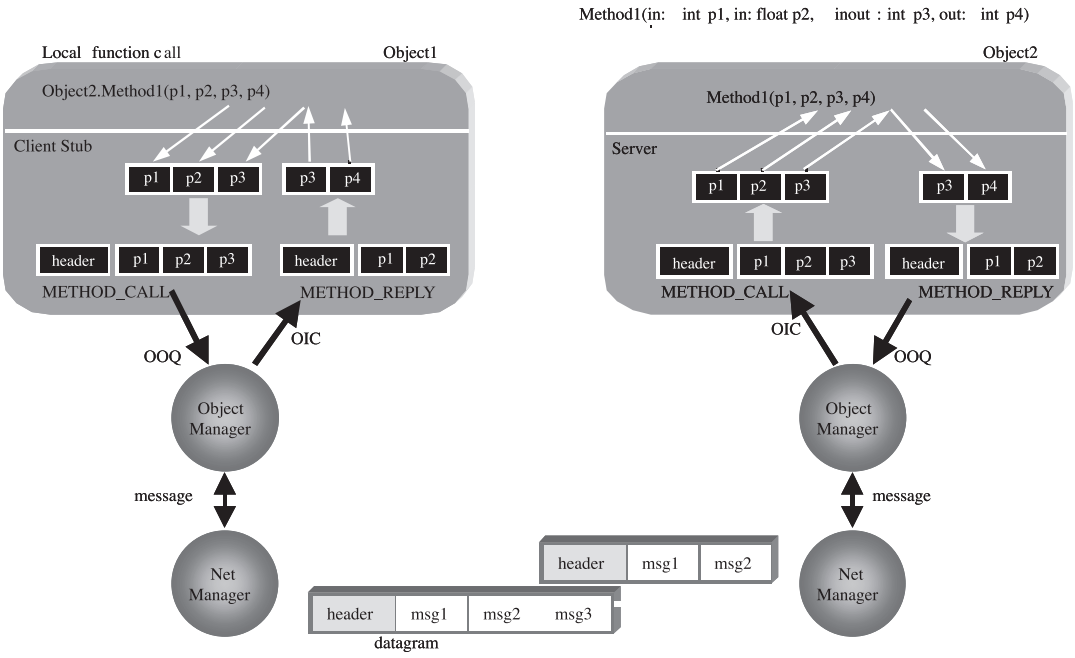

Figure 6. Synchronous Method Call

When Method1 of Object2 is called by Object1 the client stub converts all *in* and *inout* parameters (p1, p2 and p3) into an ACTIVE_OBJECT_METHOD_CALL message data part, appends a message header, sends the message to the local Object Manager and starts waiting for the ACTIVE_OBJECT_METHOD_REPLY message. When the Object2 server skeleton receives the call message, it de-marshals the message data part into parameters, calls the associated member function, obtains the out parameters when that function returns, marshals them into an ACTIVE_OBJECT_METHOD_REPLY message and sends it back to the caller via its local Object Manager. When the caller receives a reply message, it checks the message id to confirm that this message is the awaited reply, and then de-marshals the message data part into *out* parameters. At

this point, the call in Object1 returns.

6.2. Publish and Subscribe

The publish-subscribe model serves as an alternative interaction mechanism among distributed real-time objects, establishing one-to-many communication between them. The methods listed in the *publish* section allow an object to specify events, and the declarations in the *subscribe* section determine how it will respond to external events. An event may be an external stimulus, such as new data available on a device, or internal conditions, such as a property value changing. The published method is the signature of its response when the condition occurs, actually, a declaration of information that will be produced. The publisher object does not know how this information is used or which objects are interested in it. Subscriber objects are responsible for registering their interest and providing a local method to be triggered on the occurrence of the condition.

CORD-RTS implements the publish-subscribe mechanism through the multicast mode of communication, where the source is the publisher object and the destination is a list of subscriber objects that have registered for a particular published method. The publisher object delivers copies of a single message to each object on a specified list of destinations, without holding an explicit reference to each one. The delivered message actually activates a remote call to the registered method of the subscriber object with the published information passed as input parameters. It is also possible to declare a published method without any parameters and use this mechanism for synchronizing a number of objects to a certain point of execution.

Depending on the class-type of the publisher object, either the Object Manager or the Device Manager implements the publish-subscribe mechanism. If it is an active object, the Object Manager is responsible. Otherwise, the Device Manager carries out the necessary actions. Both managers maintain lists of subscriber objects with their registered methods. On the invocation of a published method, they generate asynchronous method call messages to registered methods of each subscribed object, providing the published information as the input parameter.

7. REAL-TIME FEATURES

CORD-RTS meets timing requirements of soft real-time distributed object applications through several mechanisms.

7.1. Time Synchronization

Distributed real-time computing necessitates the establishment of a global time base to supply a common time reference to all nodes executing a distributed application. The current CORD System development environment uses Network Time Protocol (NTP) Version 4.0. During the tests, 50 to 200 microseconds of accuracy was achieved in synchronizing workstation clocks on an ATM network by using the internal clock of one of the workstations as the main time reference. During system design, the developer should take into account the time synchronization requirements in the application domain and choose a time distribution implementation that is capable of meeting the given time alignment constraint. If this constraint is in the 1000 milliseconds range, any simple algorithm may be used. If 1- to 10-millisecond range is required then NTP is quite suitable. Further accuracy can only be achieved by separating the soft real-time and hard real-time parts of the system and distributing time with point-to-point connections or special, proprietary hardware devices for those system components that require high accuracy.

7.2. Timing

CORD-RTS enables the supervision of timed-events through timing controls in the millisecond domain. Thus, it becomes meaningful to specify timing assertions at the CPL statement level, such as timed-statements, timed loops, and periodic method invocations. CORD-RTS also controls the timing constraints for method calls and the arrival of results from invoked objects. Each manager in the system checks the initiation and time-out values of incoming messages to find out if they can be processed within their deadline. If not, the manager rejects the message and sends a system reply that generates an exception at the user level.

7.3. Scheduling

CORD-RTS does not perform off-line scheduling analysis since it is not a requirement for soft real-time systems. The priority based scheduling scheme of the underlying operating system is sufficient to meet timing demands. CPL allows the application to specify a static object priority in the range 1 to 10 in a class declaration. CORD-RTS maps this value to a native operating system priority during object creation. CPL objects possess static priority but during the execution of a high-priority method call, priority inversion feature causes temporary changes in their priority value.

7.4. Message Priority

CPL method calls are implicitly converted into message level representations with priority and timeout values. CORD-RTS uses a prioritized message handling scheme that adjusts processing time according to incoming message priority. Method calls generate messages with an assigned priority in the range 1 to 10 and a timeout value indicating the life-time of the message. A programmer should assign suitable priority and timeout values to an active object method call explicitly to meet real-time requirements, otherwise system defaults are used. Unless otherwise specified in the code, active objects send their messages with a priority equal to their own scheduling priority. The default priority of a method call message is equal to the priority of the active object instance and the default time-out is infinite. A 10- level priority mechanism is used for the system for the time being. However, if it becomes essential, the number of priority levels may easily be increased by changing the related system constants.

A priority inversion feature is provided for objects that are running with low priority. When a high priority message for a method call is received, the receiving object increases its scheduling priority up to the priority of the incoming message in order to get higher CPU utilization while processing this urgent message. After the message is processed, the priority value is lowered again. Since messages are subject to communication delays, in some cases, active objects may send messages with priorities higher than their own priorities.

8. Fault Tolerance

Fault tolerance is the ability of a system to recover from failures. In a distributed system, the probability of a failure increases with the number of processors. The cost of a single failure also increases because the whole distributed state may be lost.

The goal of a fault-tolerant system is to recover state data and continue operation despite failures in software units or in the underlying platform. Fault tolerance can be provided by combining hardware and software solutions. The first level recovery can be achieved using exception handlers embedded in the language. CPL provides C++ exception handlers with extensions for communication and time constraints.

Another approach for software fault tolerance is to keep redundant copies of data and processing elements against a software unit failure. In order to support this type of recovery, the CORD-RTS activates a copy of each active class server on each node regardless of its object instances. In addition to fast object creation, these servers act as "hot stand-by" processing elements that support fault tolerance in case of an object failure. Since the current version of the CORD-RTS does not have any embedded failure detection mechanism, when an active object crashes, its absence should be detected at the application level and a new instance would be created.

If an active object has to store internal state variables, it is usually not sufficient just to create a new object, because the new object needs initial data to recover its previous state and continue execution correctly. It is necessary to keep these kind of essential state variables outside of the object context. We propose three methods to save state during execution. The first one uses active class dynamic data members, which are stored outside of the object context, in a safe memory area. Once an object is initialized, it can maintain its state even after a crash. The second method is to use passive objects to store state data. This method provides only intra-node fault tolerance with inexpensive state saving. The third method uses a central active object residing on a reliable node. The methods of this object are called by the client objects who ask to save their state information. This method is relatively expensive as it requires network communication, but it is more flexible for vulnerable active objects. Passive objects containing important state data can be replicated over the nodes if the necessary data distribution is provided by convenient active objects.

Additional fault tolerance capability is provided by the CORD-RTS architecture as synchronized copies of the RTS execute on each node within the system. Each Object Manager maintains the same program table to keep track of running CPL programs and their components. If one of the nodes crashes entirely due to hardware fault or severe operating system error and reboots after fixing, it is possible for its RTS to initialize itself by obtaining program table contents from the master node. An application program in charge of system management can detect the nodal failure and allocate those objects on other nodes. When the failed node recovers and becomes operational again, the stem manager can reallocate objects on this node again.

Soft real-time systems that must be dependable typically replicate application objects on different processors within the distributed system. Since it is possible to create several active object instances using the same class server, a fail-stop mechanism can also be used. When an object fails, another one resumes service. Thus, n number of objects can afford $n-1$ failures without recreation. However, an efficient detection and naming mechanism has to be implemented at the application level. CORD-RTS features together with CPL programs may be used to support one or more of these mechanisms attempting to recover from hardware or software faults.

9. Perpormance Analysis

We carried out various tests to determine overall system performance and also to detect and remove system bottlenecks, the most time-consuming parts of the software. The first test environment where we initially developed the software consisted of Sun Sparc4 (100 MHz) workstations with SunOS 4.1.3 on 10 Mbs Ethernet. The second test environment was a stand-alone Sparc notebook (70 MHz) with Solaris 2.5. This computer was used just for demonstration purposes. The third one consisted of Sun Ultra60 (300 MHz) workstations with Solaris 2.5 on 155 Mbs ATM network with LAN emulation.

The tests we carried out measured the cost of local/remote object communication (method calls),

direct and mutually excluded access to shared data encapsulated by a passive object, active and passive object creation and active object migration. The Table shows some performance figures in microseconds obtained by averaging the time consumed by a method call. Each test was performed by invoking a call 10 thousand to 1 million times and measuring the total elapsed time. Thus, the execution time of a single invocation was calculated as an average value. During the tests, a 4-byte data type (an integer type) was used as the method parameter. The computers were idle and dedicated for the tests. Therefore, these performance tests indicate the "best-case" figures of the CORD System. The results show the cost of communication and context switches, as well as the search latency of the data structures in the Object Manager. We observed that inter-object communication time can be reduced dramatically if passive objects were used. Active object communication latency increased as the number of active objects requiring parallel execution increased.

Table Performance Figures

Test	Sun Sparc 4 Ethernet	SparcBook Stand-alone	Sun Ultra 60 ATM
Asynchronous method call, local node	850	1300	550
Asynchronous method call, remote node	2400	-	1100
Published method, local node	900	1300	600
Published method, remote node	2200	-	1200
Published device data, same node	1100	1500	700
Synchronous method-call, local node	2600	3500	1550
Synchronous method-call, remote nodes	6700	-	2250
Reading passive object with Lock/Unlock	180	220	50
Reading passive object without Lock/ Unlock	0.18	0.25	0.15
Asynchronous read from a local device	2200	2400	1800
Active object creation, local node	< 8000	< 10000	< 5000
Passive object creation, local node	< 4000	< 5000	< 2000
Active object migration	-	-	< 15000

10. Conclusion

In this paper, we described the design and implementation issues of CORD-RTS, a middleware for management of real-time distributed objects. CORD-RTS fulfills its core functionalities through a set of manager components. It provides a high level infrastructure for inter-object and inter-node communication. Interactions among real-time objects are either in the form of location transparent method calls or the publish-subscribe mechanism. One of the main contributions of CORD-RTS is its ability to instantiate real-time objects anywhere throughout a distributed system. Objects only experience the added overhead of a remote method call.

CORD-RTS meets the timing requirements of soft real-time applications through a prioritized message handling scheme and supervision of timing constraints for method calls and arrival of results from invoked objects. It does not perform off-line scheduling analysis and relies on the priority based scheduling scheme of the underlying operating system. Our experience and performance results reveal that a commercial general-purpose operating system can be extended with a suitable middleware to approach the structure of a real-time kernel and thus be capable of supporting the timing demands of soft real-time applications.

The fault tolerance mechanism that CORD-RTS currently provides is quite primitive, consisting mostly of exceptions returned if a request fails. The programmer needs to implement the necessary fault

tolerance mechanism within the application to meet its reliability needs. As future work, we consider to add components that handle fault detection and fault recovery in a manner that is transparent to the CPL application.

This system has been implemented at the Turkish Navy Software Development Center. We are continuing to both research and develop CORD-RTS software, adapting it to new requirements. Enhancements to the language constructs may also be expected.

References

[1] E.Saridogan, "Design and Implementation of a Concurrent, Object-Oriented, Real- Time and Distributed Programming Language with its Supportive Run-Time System", Ph.D. Thesis, Istanbul Technical University, Institute of Science and Technology, January 2000.

[2] E.Saridogan, N.Erdogan, "CPL: A Language for Real-Time Distributed Object Programming", Elektrik, TUBITAK, Vol. 10, No. 1, pp. 1-21, 2002.

[3] E.Saridogan, N.Erdogan, "A Real-Time and Distributed System with Programming Language Abstraction", International Conference on Parallel and Distributed Processing Techniques and Applications, June 28-July 1, Las Vegas, USA, 1999.

[4] E.Shokri, P.Sheu, "Real-Time Distributed Object Computing: An Emerging Field", Computer, June 2000, p. 45-46.

[5] The Real-Time Experts Group, "The Real-Time Specification for Java", Version 0.8.1, 1999, http://www.rtj.org/rtj.pdf

[6] D.C.Schmidt, F.Kuhns, "An Overview of the Real-Time CORBA Specification", Computer, June 2000, p.56-63.

[7] K.H.Kim, "Object-Oriented Real-Time Distributed Programming and Supportive Middleware", Proc. ICPADS 2000, Japan, July 2000, pp. 10-20.

[8] N.Brown, C.Kindel, "Distributed Component Object Model Protocol-DCOM/1.0", Microsoft Corp., 1998, http://www.microsoft.com/dcom/

[9] RFC 1305, 1992. Network Time Protocol (V3), IETF.

[10] Object Management Group, The Common Object Request Broker: Architecture and Specification Rev.2.1, Aug. 1997.