Adaptable Real-Time Distributed Object Management
for Command and Control Systems

John Maurer, Roman Ginis, Richard Freedman,
Michael Squadrito, Steven Wohlever, Bhavani Thuraisingham
MITRE Corporation
Bedford, MA 01730, USA

Abstract
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.

This paper describes our Adaptable Real-time Distributed Object Management (ARTDOM) approach for command and control systems. We have constructed a testbed environment and experimented with a Common Object Request Broker Architecture (CORBA) Object Request Broker (ORB). This project is a three year initiative and is expected to be completed at the end of 1999.

1. Introduction

1.1. Overview
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.

Between 1993 and 1996, MITRE’s Evolvable Real-Time C3 initiative developed an approach that would enable current real-time systems to evolve into the systems of the future. The candidate evolution approach is to leverage off near-term system upgrade and/or P3I (Pre Planned Product Improvement) activity to put a new architecture framework in place. The emphasis was on transitioning to open architectures that 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.

MITRE’s evolvable real-time systems initiative designed and implemented an infrastructure and data manager so that various applications could be hosted on the infrastructure. We chose a distributed object management system to integrate the various components. The emphasis was on meeting real-time constraints. We experimented with Object Management Group’s Common Object Request Broker (CORBA) based distributed object management systems. During 1996, we started a follow-on effort to design flexible adaptable distributed object management systems for command and control systems. Such an adaptable system would switch scheduling algorithms, policies, and protocols depending on the need and the environment. Both initiatives were carried out under MITRE’s Mission-oriented Investigation and Experimentation (MOIE) programs sponsored by the United States Air Force.

The report describes our approach to developing Adaptable Real-time Distributed Object Management System (ARTDOM) for command and control systems. In particular our work carried out on this initiative during the first year of the project is described. This initiative is a three year initiatives and is expected to be completed at the end of 1999.
The organization of this section is as follows. Section 1.2 describes the MOIE context and technology transfer plan. Section 1.3 describes some key aspects that would influence our project. These include quality of service, integrating object request brokers with object database systems, and the Defense Information infrastructure Common Operating Environment. A summary of this report is the subject of Section 1.4.

1.2. MOIE context and technology transfer plan

Through its interaction with the OMG Real-time SIG, the ARTDOM MOIE has a forum in which it can influence the development directions of ORB vendors. In particular, the specifications for real-time CORBA in development within the SIG will have an impact on DII COE, the government’s specification for a defense information infrastructure that vendors will be required to adhere to for future defense contracts. This technology is also being transferred to the Open Group/MITRE-Rome AWACS ATD, which is using best-effort scheduling to get predictable tracker performance. Figure 1.1. illustrates the flow of this technology transfer.

![Figure 1.1. MOIE context and technology transfer plan](image)

1.3. Some key aspects

This section describes some of the key aspects of the project. They are: quality of service, integrating ORBs (Object Request Brokers) with ODBMSs (Object Database Management Systems), and the DII COE (Defense Information Infrastructure Common Operating Environment).

1.3.1. Quality of Service (QoS). Quality of service [Sluman97] is a general term that covers system performance, as opposed to system operation. A system will be built to perform some set of functions for its users. These functions can be called the operational or functional features of the system; they include holding information as well as doing things of various kinds. But the performance of each function will take time, require system resources and be subject to occasional system errors or failures; these and other similar features are performance or non-functional features of the system, which come under the heading of QoS. In general, the users of a system will have requirements both for the functions that are to be performed and for the QoS with which they are performed.

Another approach to defining QoS is to identify the system properties of concern by means of examples. From this point of view, QoS can be said to be concerned with the properties of system functions and information such as delay, response time, throughput, stability or usability of, continuous media output, the freshness and accuracy of information, the coherence of distributed information, availability and so on. A useful, but somewhat simplified, summary is to say that QoS is concerned with timeliness, accuracy and integrity. In practice, the distinction between the functional and the QoS aspects of a system is not as black-and-white as that description might suggest, and it depends in part on how the system is being viewed at a given time. For example, a user's QoS requirement for rapid access to information may be satisfied by means that involve a negotiation process between system elements. At the user level, that process is part of QoS, but at an engineering level it may be seen as a system function that must itself be performed with a particular QoS, such as a bound on the time allowed to complete the negotiation. It is important in other ways: for example, a delay requirement at the user level (as in the example just given) may translate into throughput and storage requirements at a more detailed engineering level. Or a user requirement for reliability and coherence may translate into an engineering requirement to use transaction processing techniques.

A question that is often asked is whether security is part of QoS. Security characteristics are identified as QoS characteristics in the ISO/IEC QoS Framework standard, for the reason that security concerns non-functional aspects of systems – though it is not intrinsically concerned with performance; and it will be important to provide a unified treatment of security and performance characteristics in OMG (for example in terms of requests, negotiation, asynchronous feedback and so on). On the other hand, there are significant differences between security and performance-related QoS: security requirements are usually non-negotiable and are discrete (i.e., expressed in levels) rather than continuous; and the mechanisms and expertise needed to provide security have little in common with those needed for other types of QoS.

The OMG adopts the practical approach of treating them as quite distinct, though within a common architecture that identifies the appropriate interfaces needed between security and other QoS functions. Then
from a performance QoS point of view, security functions can be treated just like any other application functions; they may have QoS requirements, and the choice of security mechanisms may affect the QoS with which other system functions have to be performed in order to meet user requirements for QoS.

1.3.2. Integrating ORB and ODBMS. Our work necessitates an approach for integrating ORB and ODBMS, which is non-trivial because commercial products in these two categories have not been design to work together seamlessly. We would like to base our infrastructure on standard mechanisms, but real-time requirements must be met across integrated mechanisms and adaptation mechanisms should be common to both ORB and ODBMS behavior.

The OMG has adopted the Persistent Object Service (POS), which allows CORBA objects to be stored and retrieved using many possible means (files, databases, etc.) [Coss94]. Although some implementations exist [e.g., Secant97], POS is limited as a means for integration of real-time ORBs and ODBMSs [Session96]. For example, POS ignores the role of the ODBMS in efficient transfer and caching of fine-grained objects. The OMG is currently revising POS to address these issues.

ODBMS vendors have taken an adapter approach, where the ORB is used to locate and bind to the ODBMS adapter and ODBMS mechanisms are then used to access database objects. Although adapter mechanisms combine CORBA and ODBMS standards, the mechanisms themselves are proprietary and specific to the ORB and ODBMS used. Francisco Reverbel has investigated these adapters in the context of medical information systems [Reverbel96]. It remains to be seen whether adapters could be modified for use in real-time systems. Adapters are modeled on an approach outlined by ODMG that uses CORBA for communication between coarse-grained objects and ODMG for fine-grained objects [ODMG97]. The ODMG approach seems to be common among CORBA developers. AWACS has extended this approach to take advantage of the fact that the ORB and ODBMS are newly constructed [Hughes97a]. In AWACS, the ODBMS uses the ORB for distributed communication but an ODBMS mechanism is used to move objects from machine to machine.

1.3.3. DII COE. The Defense Information Infrastructure Common Operating Environment (DII COE) is a Department of Defense Initiative to design and develop an environment that can be used across the different services. Figure 1.2 illustrates the DII COE.

Various working groups are involved in specifying the DII COE for various components like distributed computing, data management, and multimedia. Until recently, there were no requirements for real-time processing. Around August of 1996, interest began on examining real-time issues for DII COE. The work carried out under MITRE’s evolvable real-time systems initiative as well as the work of OMG’s real-time special interest group have influenced the preliminary work on examining real-time issues for the DII COE. A working group has now been formed to explore real-time issues for the DII COE. The work under the ARTDOM initiative will be fed into the work on Real-time DII COE.

1.4. Summary of the paper

This report describes our work during 1997 on adaptable real-time distributed object management for command and control systems. Our approach to adaptable middleware is the subject of Section 2. Motivation for our work as well as discussions on metaobject protocols and reflection are given. Our adaptable ORB experiment is presented in Section 3. Adaptable real-time CORBA is discussed in Section 4. Future work will be built on the discussions in this section. Impact of our work as well as future directions are addressed in Section 5.

Our work for 1997 was built on our previous work on evolvable real-time systems for command and control systems. These papers have been published in WORDS 96 and WORDS 97 workshops.

2. Approach to adaptable systems

2.1. Goals

The goal in designing an adaptable system is to develop a system that:

- Supports unanticipated configurations
- Works correctly (in unanticipated configurations)
- Balances safety and ease of use
- Provides a good default configuration
- Supports user-supplied components
2.2. Motivation

On both the Advanced Technology Demonstration and Evolvable System MOIE projects, the need for a reflective capability has become apparent. A few examples of areas that could be addressed using an open implementation are given below.

2.2.1. Communication protocols. TCP/IP is standard protocol for most ORBs. Real-time systems will need new communication protocols. If ORBS are to interoperate, they must use compatible communication protocols. Currently, CORBA 2.0 mandates the IIOP (Internet Interoperability Protocols) as the default means for ORB interoperability. IIOP is based on TCP/IP. If a C3 system requires a protocol other than TCP/IP, ORBS are not guaranteed to interoperate. Many issues are involved to accomplish successful interoperability, especially with real-time and fault-tolerance requirements. Interoperability of real-time and non-real-time systems are an example that requires careful consideration. Question: Do you wait for the vendor to provide a “protocol” for your “specialized” application? C3 Systems may never dominate the market. Using metaobjects allows the ORB architecture to be opened up without the vendor providing source code. Another C3 “metaobject” vendor can provide.

2.2.2. Real-time data manager and transaction services. Standard transaction services do not support real-time. Evolvable system MOIE project implemented real-time transactions using priority ceiling and real-time scheduling.

2.2.3. Object-oriented frameworks. An object-oriented framework is essentially a skeletal application: the user plugs his code through inheritance and composition. This works well unless the fundamental operations and concepts need to be overridden: method invocation, properties of objects, “implementation details” such as internal queuing (FIFO vs. priority), new concepts such as “object groups” and “multicast method invocation” may not be implementable correctly without the introduction of new communication protocols.

2.2.4. Real-time high performance computing. Suppose you want to use Real-Time CORBA on a symmetric multiprocessor or a massively parallel processor. Currently MPI, a low level, highly efficient, message passing interface, is used. This increases the complexity of software implementation for high performance computing since low-level abstractions are used. The trade of complexity for performance is made because standard CORBA implementations are not efficient enough. Metaobjects would permit method invocation to be optimized for high-performance computing by using MPI as the transport mechanism for method invocations. This would open up the use of CORBA for high-performance computing.

The most important example given above is that of an object-oriented framework. It is here where most of our hopes for source code and design reuse lay, together with the concept of “openness” extended to the level of the application. With an object-oriented framework for sensor fusion, we would be able to add sensor sources and sensor data processing with much less effort than today. CORBA is a necessary but not sufficient technology for accomplishing this because of the stringent performance requirements. We will prototype a simple sensor fusion framework, based on an ORB with an open implementation. In some sense, this is an extension of the traditional notion of an object-oriented framework, since the existence of metaobject protocols allow the framework to redefine aspects of what it means to be an object (e.g., semantics of method invocation).

2.3. Open system approach

Since a component framework supports the integration of software from diverse sources, we think of it as an open approach. We believe such an approach is essential for future applications, but it changes the roles of people who build and use these applications. A component framework blurs the distinction between programmer and user roles. A vendor developing a component has less ability to lock in future business by providing a new de facto standard and so must cope with a different profit model. We hope that this technology will also lead to more useful software documentation, preferably executable as part of the delivered system. In addition, we wish to see components with more focused specialization rather than breadth of functionality.

A major criticism of the middleware is that it dictates particular implementation decisions that may not be appropriate for all applications. Some aspects of the middleware may not deliver the required performance or lack the necessary functionality. For example, we found that the current commercial CORBA implementations fail to fulfill many of AWACS real-time requirements. Because the middleware is often implemented as a black box, it allows little or no flexibility in adapting it to the application. If one were to choose one of the commercial ORBs as the baseline, they currently have two choices: either (1) request that the middleware vendor make the desired change (which may or may not happen based on market considerations) or (2) purchase the source code and customize the middleware, which undermines the rationale for using Commercial Off The Shelf (COTS) approach adapted by the DoD. The alternative is only to develop an ORB from scratch, incurring all the costs of the development and support.
The problem with commercial CORBA implementations is the black box software engineering approach taken by many middleware vendors. A solution is middleware designed using open techniques. These Open Implementation principles [Kiczales96, Zimmermann96] allow users or applications to customize some aspects of the middleware and make it more suitable for their needs. For example, if an ORB’s Protocol/Transport is not fast enough or does not support a legacy standard used by the present system, it can be replaced without affecting other parts of the ORB. In this way, the middleware can adapt to the needs of the application. This can only be accomplished if the middleware design supports a specific set of features that make it sufficiently open. These include object-oriented design (clear, well-defined interfaces between modules) and the use of either object-oriented framework or metaobject protocol approaches described later.

The advantages of an open ORB are similar to that of any open system. First, users (e.g., ORB developers) are not married to one particular CORBA implementation: various vendors can write the modules for the ORB in the same way as they write services. This offers a higher flexibility of choice of components, since users only buy the parts they need. The new modules can be purchased separately, developed in house, or contracted outside. For example, an open implementation would allow an ORB vendor to sell a scaled-down ORB at a cheaper price, and to sell extra components for additional fees. Moreover, the customer can buy the additional components from different companies, thus encouraging competition in the component market. In this way, the ORB upgrades can also be cheaper, since the customer does not have to upgrade the entire ORB, just the modules that have grown obsolete.

Consider a customer with unique ORB requirements (e.g., real-time). Basic method invocations and a naming service are usually a part of the standard CORBA implementation. However, a customer who wants real-time method invocations (i.e., with timing constraints) and a real-time database for naming should have the option to buy them separately. If the same customer wants specialized components such as real-time scheduling and priority-ceiling concurrency control protocol (that are not necessary for a non-real-time CORBA user), then it should be possible to either build or buy these components (see Figure 2.1).

Currently, neither the CORBA standard nor most commercial implementations support such features. The open implementation approach addresses this need.

### 2.4. Reflection approach

Systems that can monitor their own behavior and performance and change based on that information are called reflective. It is of interest to develop an adaptive system that is also reflective, such that it would be capable of adapting dynamically with the changes in real time. For example, if a selection of resources of different quality were available (e.g., a set of radars), a reflective system would be able to choose different radars depending on the application as well as additional user constraints imposed on a specific operation (e.g., timing constraints on getting current information about a contact).

If one were using metaobject protocols to implement such a system, it could have a metalayer (set of metaobjects) whose job it is to monitor the rest of the objects in the system to determine whether or not they are meeting their QoS requirements. Depending on the result, the metalayer might choose a different set of objects (that represent groups of resources) to handle certain tasks, sacrificing something (time vs. quality) in order to get back a result within its specified parameters. In a sense, a reflective ORB would reconfigure itself to work better.

### 2.5. Metaobject approach

Metaobject Protocols and Object-Oriented Frameworks are two prevalent techniques used to implement open systems today. We are studying the two techniques to use the best of each in our design.

A metaobject protocol is a powerful software engineering approach to designing complex object-oriented systems. It allows a designer to build scaleable, evolvable and adaptable ORBs that can be dynamically configured and monitored. In such a system, the flow of operation can be changed at run-time to meet new QoS requirements. This change is accomplished using additional objects called metaobjects that can control and monitor other objects that compose the system. A baseobject is any object executing application code and implementing interfaces that are used by other baseobjects. A metaobject is an object that can control and introspect certain aspects of the execution of a baseobject. The interfaces to metaobjects are called metaobject protocols (MOPs).

The baseobjects of an ORB perform functionality in terms of the algorithm (e.g., message routing or database lookup). Baseobjects make up the code that runs the application. Metaobjects have two purposes. First, they provide a means to manipulate the operation of the

![Figure 2.1. Open ORB approach](image)

---

Authorized licensed use limited to: Univ of Texas at Dallas. Downloaded on April 19,2010 at 17:27:20 UTC from IEEE Xplore. Restrictions apply.
baseobjects. In this way, by calling methods on a metaobject’s interface, it is possible to change the flow of control over the ORB. Second, metaobjects can implement some functionality that is commonly used by baseobjects (e.g., persistence), simplifying the baseobject implementation when exported to the metalevel [Zimmerman96].

An easy way to understand the baseobject/metaobject relationship is by following a use case. Consider a CORBA ORB implementation. Every ORB comes with some form of a naming service that allows CORBA clients to locate CORBA servers by name, IP address, domain, etc. Suppose we are the developers of the ORB and instead of just one naming service we decide to implement two: the First-Found service that quickly searches for the first server with the requested name, and the Load-Balanced service that can find a server with the least current load, but requires more time (see Figure 2.2.). If the services are both written using object-oriented paradigm, both service implementations consist of a set of baseobjects that perform all the necessary work to do a name lookup. Suppose now that we want to be able to switch between the two types of naming services depending on some external requirements, e.g., timing constraints. That is, if we have more time to perform a name lookup, we would use the Load-Balanced service, and if the timing is short, then First-Found would suffice. Thus, the ORB would be able to adapt to the requirements of the applications that are using it.

To implement the service choice functionality, two aspects are required: the code that does the switching and an interface that exposes the choices to the user and/or some other application. In this example, the objects that implement the switching code are the metaobjects with respect to the objects that implement the naming service functionality. The metaobject methods are the interface for the switch. If one of the metaobjects has a method called ServiceSelect(service type), then by calling this method it is possible to choose one of the services available. The ServiceSelect is a part of the metaobject protocol for the naming service.

To demonstrate a case of code simplification using metaobjects, imagine that we decide that version 2 of the naming service would benefit from caching some of the server location information. Instead of changing the naming service implementation, we can simply write a new type of metaobject that performs data caching and add it into the metaobject hierarchy (Figure 2.3.). In this way, we are able to evolve an existing naming service into a new version with minimal effort, since virtually none of the existing code had to be changed in the process.

Examples of metaobjects are the metaclasses that control the behavior of classes at object creation time. For instance, a metaclass might intercept the new call (as in C++) that creates an instance of a class (a template for an object in C++) and instead load a previously stored object from disk intact with all the state data. Another example is metaobjects that are responsible for method invocations, via a transparent transfer of control to the metaobject by the run-time of the programming language. Here, the metaobject intercepts a method invocation to add some additional functionality to the basic call. For example, this would allow the metaobject to execute some code before and after the invocation, or even pass the call parameters to a completely different function.

Metaobject protocols can be used by either baseobjects or other metaobjects to manipulate the operation of a program. There are three categories of metaobject protocols: explicit, implicit and inter-metaobject.

Explicit metaobject protocols are used by baseobjects to communicate with the metaobject in order to make a change in the behavior of the baseobject (for example changing the status of an object from volatile to persistent). The change is realized by an implicit metaobject protocol. Implicit metaobject protocols are in charge of controlling baseobject behavior transparently (e.g., calling a method on another baseobject or the creation of an instance of a class) and inter-metaobject protocols are used for the communication between individual metaobjects [Zimmerman96].

Using the metaobject pattern, it is easy to add a control interface to an ORB that is semantically separate from the standard ORB interface. This interface, called metainterface, provides a user or an application with an ability to tailor the objects in the ORB towards specific requirements. The metainterface is orthogonal to the standard interface, because the actions that can be
performed on it do not change the functionality of the ORB, only its mode of operation (Figure 2.4).

For example, the metainterface could present a method SetStrategy(x) that selects one of several implementation strategies found inside the ORB. While the baseobjects inside the ORB continue executing their standard functionality tied to the standard ORB interface, the metainterface impacts the behavior of the system by controlling which of baseobjects get invoked.

The metaobject approach gives an ORB developer a two-level control over the ORB’s operation. One at the functional level and one at the operational level. Depending on the requirements of the application, the developer may choose to use the metainterface to configure the ORB to one of the existing (implemented in the ORB) operational strategies or to write their own and include it in the selection.

2.6. MITRE’s approach to and adaptable ORB

2.6.1. Goals. The goal of this project was to identify the desired features of an open ORB, create a design and to implement a prototype in coordination with the AWACS effort. This implementation is tailorable to demanding real-time and fault-tolerance requirements through the use of metaobject protocols and reflective techniques. Section 2.6.2 describes the approach taken in designing an adaptable ORB. Section 2.6.3 discusses ongoing related work in the area of adaptable ORBs.

2.6.2. Approach. The general approach taken by this MOIE to developing an adaptable ORB is illustrated in Figure 2.5. See Section 3 for more details.

The components of the system are the ORB, services such as scheduling and data management and the application frameworks. Each component consists of various subcomponents and protocols. The idea is to switch the subcomponents and protocols without disrupting the operation of the system. For example, one could switch the scheduling protocols or the transport protocols or even the application components.

Figure 2.6. illustrates how new versions of a component can be derived such that new functionality, such as performance and system monitoring, can be added. The fact that these modifications can be made relatively easily represents a tremendous economic value. Section 3 describes the approach taken by this project in more detail.

2.6.3. Related efforts. The effort that we have described has been influenced by the work on metaobject protocols carried out at Xerox Palo Alto Research Center. More recently, there is also work on adaptable ORBs at Washington University in St. Louis. We have combined the work described in these efforts together with our previous work on evolvable real-time command and control systems. Some of the work carried out at Lockheed Martin on integrating ORBs with ODBMSs have also impacted our work. An excellent discussion on object-oriented frameworks and component integration is given in [ACM97].

3. Adaptable ORB experiment

MITRE presents a Technical Symposium every June. We presented the ARTDOM demonstration at the Bedford Symposium. It was intended to be sufficiently non-trivial to be a meaningful demonstration, but not a showy sidetrack from our real objectives. It is intended to convey technical methods used and at least hint at future applications for those methods, but does not require
The four components will now be explained in more detail.

3.1. The server

The server was written using ILU (SYLU). The server’s skeleton code was modified along with the ORB code to facilitate changing protocols. The interface of the server consisted of RPC method calls and HTTP POST() and GET() methods. During the demonstration, the server could be switched between the RPC and HTTP modes. This was accomplished by terminating the current server and starting a new server to take its place. When a server started, it referred to a configuration file called ‘switch.’ The server was always started in RPC mode. However, when the switch file was read, the settings were changed accordingly. The switch file contained the protocol mode and port number for the server. For the demo, it was either “RPC; sunprerm;tcp_0_0; sunrpc” or “httpd; http; tcp_0_3333; cgi”.

In RPC mode, the RPC methods were called directly by the client. There were two methods, that when called, would print text to the server’s and client’s window. Since the server had to be shut down in order to change modes, an additional method was put on the interface called shutdown(). When the operator of the demo selected the HTTP transport protocol on the GUI display, a process was started which connected to the RPC server and invoked the shutdown() method. On the server side, the shutdown method invoked ilu.ExitMainLoop(), which gracefully terminated the server.

In HTTP mode, the POST() and GET() methods were used by the protocol to transmit information between the browser (client) and the server. The user connected to the server by typing the server’s port number and file name as the URL. The user could access HTML pages from the server by clicking on links, just like a normal HTTP server.

When the operator of the demo selected the RPC protocol on the display, a process was started to shut down the server. However, this process did not connect to the HTTP server. Since the server was in HTTP mode, it only accepted calls on the GET() method interface. We were unable to determine how to access the server in HTTP mode to notify it to shut down. The shut down process instead simply looked up the server in the list of processes running on the host computer, and sent a ‘kill’ signal to the server. This was probably not the best way to shut the server down, but it was sufficient for the demonstration.

3.2. The RPC client

The RPC client was a process that could be started manually by the user to access the RPC server. Once started, the client accessed the server’s two RPC methods, invoking each method three times. Each invocation was performed after a five second pause, to allow the onlookers time to see the output written to the window. The output was simply the name of the method called and some additional information concerning the method parameters used.
3.3. The HTTP client

The HTTP client was a Netscape browser. To connect to the HTTP server, the user would type in the URL of the server, which was the port number and computer the server was on. Once the user was connected, the server would return an HTML page. The user could then access other HTML pages by clicking on links, just like a normal HTTP server.

3.4. The controller/GUI

The controller determined which server was running and the GUI displayed what was happening over the network. The graphical representation did not accurately depict network traffic, but simply let users see which protocol was in use. The display showed several protocol modes to indicate potential changes which could be made to the server. Two types of protocols that were examined are:

- Messaging Protocol Components: IIOP, HTTP, SUN RPC
- Transport Protocol Components: TCP/IP, UDP, IPX/SPX.

For the demonstration, the only messaging protocols used were HTTP and SUN RPC. The only transport protocol used was TCP/IP. To control the mode of the server, the operator clicked on either an RPC or HTTP selector button. The controller would then perform the necessary operations to change from the current protocol to the new protocol.

3.5. Changing to HTTP mode

If the server was in RPC mode, the controller would shutdown the server using a process to call the shutdown method on the server. The server would then call `ilu.ExitMainLoop()`. Next, the controller would write the switch file with the string “httpd;http;tcp_0_3333;cgi” indicating that the server would run in HTTP mode, on port number 3333. The server was started, read the switch file, and changed its mode to the current settings. The display was then updated to indicate that HTTP network packets were being used. The user could then use a browser to access the server using the port number 3333.

3.6. Changing to RPC mode

If the server was in HTTP mode, the controller would shutdown the server using the system kill command. The controller would write the switch file with “RPC;sunrpcrm;tcp_0_0;sunrpc”, which indicated that the server would run in RPC mode, using a port number assigned by the system. After one second, which is enough time for the server to start, the controller started the RPC client. The display was then updated to indicate that RPC messaging was being used. Once the server was running, the user could also start more RPC clients.

Earlier work had been done on a network connection in which the message protocol could be switched between HTTP and one (or more) of the many protocols commonly used for client-server exchange in a local network, such as SUN RPC. It was decided that the multi-faceted capability could be indicated by depicting clients and servers as collections of components, each selected independently from various component toolkits. Even though only two components were to be activated, the existence of multiple sets would make it clear that the technique was not restricted to the one toolkit shown. A box was labeled “Messaging Protocol Components” and contained boxes labeled “SUN RPC”, “HTTP”, “IIOP”, and an ellipsis to indicate other components were possible. A second box was labeled “Transport Protocol Components” and contained boxes labeled “TCP/IP”, “UDP/IP”, “IIOP”, and an ellipsis. The two toolkits were themselves contained, along with an ellipsis, in a box labeled “Protocol Components.” When one of the boxes was selected (via drag and drop) its image would go to the client and server objects, and corresponding client and server processes would be started up. They would be exercised on a second screen, while their activity at a more abstract level would continue to be indicated in this primary screen. HTTP activity could be demonstrated by activating a Netscape browser.
already made (indicated by the TCP/IP box at the lower end of each) and with a Messaging Protocol choice still to be made (indicated by an empty box just above the Transport Protocol box).

Figure 3.2. Demonstration screen 2

Figure 3.2. shows the same system after the Messaging Protocol choice has been made. In the demonstration there was simulated animation showing the choice moving to the ORB (the horizontal connection between the client and server objects) and being replicated and disseminated through the network, followed by “traffic”, represented by a box moving back and forth between the two objects. In the picture this box is seen on the lower right portion of the ORB.

4. Adaptable real-time CORBA

4.1. Requirements

A major criticism of middleware (and CORBA in particular) is that it dictates a particular software architecture that may not be appropriate to all software systems development. The current CORBA implementations are not easily adaptable or open today. This criticism can be leveled at systems software (operating systems, compilers, language runtimes, databases) in general. If the operating system that was selected for a system development project has poor performance because of inappropriate resource management policies, there is little that the developer can do. As mentioned before, the developer can either (1) request that the vendor make the change (which may or may not happen based on vendor market considerations) or (2) purchase the source code and customize the vendor product undermines the rationale for using COTS products in the first place.

A major practical problem in applying high-level approaches such as CORBA to systems development is performance. General purpose protocols such as TCP/IP, for example, may be inefficient for use in embedded systems. Lockheed-Martin, as an AWACS contractor, is developing a Real-Time CORBA implementation, that while largely CORBA 2.0 compliant, has a design that is highly influenced by AWACS operational requirements and performance requirements, both real-time and fault-tolerance. A fact of life of this type of system is that to meet these stringent requirements, the implementation of the real-time operating system and middleware must be tailored to some extent. This is in part why CORBA is not sufficient as it stands today and we now have the Real-Time CORBA SIG.

Object technology is necessary if high-levels of reuse are to be achieved, but object-technology implementation must be “open” if it is to succeed in systems with stringent performance requirements such as high-performance (massively parallel) applications and real-time, fault-tolerant applications. High levels of design and code reuse have been achieved through the use of commercial object-oriented frameworks such as MacApp (Macintosh Applications Framework) and MFC (Microsoft Foundation Class Library). A new generation of CORBA-based object-oriented frameworks are becoming commercially available: OpenDoc from Component Integration Laboratories is a significant example. It reflects the evolution of the concept of an object-oriented framework from supporting the development of simple desktop applications to highly distributed object-oriented applications.

It, in fact, turns the whole notion of application on its head: OpenDoc supports components that the user integrates to produce his own custom “application.” The components are not required to run on the same platform but may be distributed over a heterogeneous network. The components are built without knowledge of each other and are assembled together by the user at run-time. This in itself is revolutionary: it shifts construction of the application to the user and focuses the effort of the developer on components. The user now performs what the application developer use to do: the user implements his “application” with off-the-shelf components which are objects.

This trend is widespread: Microsoft is moving in this direction with OLE/COM and DOM. The OMG has a number of Task Forces and Special Interest Groups developing industry-specific CORBA-based OO-frameworks (object-oriented frameworks). SemTech has developed and specified a CORBA-based manufacturing OO-framework. The trend is towards OO-frameworks capturing domain knowledge. These OO-frameworks are highly adaptable but are not sufficiently adaptable to support real-time, fault-tolerant, distributed applications.

OO-frameworks are necessary, but not sufficient. Real-time, Highly Available C3 Systems have critical requirements that cannot be met by the current generation of ORBs. The next generation of real-time ORBS will go a long way towards meeting these critical requirements. But this will only happen when the “out-of-the-box” ORB and underlying real-time operating system, as established
by analysis and benchmarking, meet the application performance requirements.

Of course, a general purpose COTS solution will make the necessary compromises to satisfy the largest segment of its target market. These compromises will inevitably mean the failure of COTS CORBA solutions to support the critical performance requirements of some systems. Other, perhaps more serious, failures will be the problem of interoperability with legacy software. A CORBA vendor may not be interested in helping a government user integrate current legacy software with a new CORBA-based system unless it represents a sufficient return on investment. In this case, it will be up to the user to provide a gateway, in some cases, that integrates the two systems.

Unfortunately, a gateway may not provide sufficient performance or may require extra hardware that will increase the cost of the system. In some cases, a solution may require access to the internals of the ORB to extend it to support specific application requirements. This has, historically, been an expensive and risky approach. The vendor will not usually support user modified systems software. Thus, the user is left with an expensive maintenance problem and is locked into a stovepipe solution that he attempted to avoid by resorting to COTS products.

To meet stringent real-time and fault-tolerance requirements normally means tailoring the internals of the operating system and CORBA middleware to use appropriate scheduling techniques and communication protocols. This is not feasible using current COTS if continuing support from the vendor is desired, high maintenance costs are to be avoided, and if the user desires continuing access to advancing commercial technology. With “open implementation” technology, using metaobjects and reflection, these problems may be solved.

4.2. Design

Here we outline our approach for RTDM services for command and control applications. Our design consists of RTDM services that form a framework for real-time systems. These services form a layer over a real-time operating system. It is relatively easy to migrate a legacy system to this infrastructure [Hughes97b]. As with ordinary ODBMS and CORBA systems, migrated applications can then evolve with minimal changes to RTDM services. Many technologies are important to this work but here we discuss only the critical subset.

We have chosen an object-oriented model for many of the same reasons given for non-real-time applications. We also have found objects to be convenient for the specification of real-time constraints (e.g., how often a report object should be updated) and of semantic concurrency-control metadata (e.g., which methods of an object can be executed concurrently). This approach in effect extends the object model to support real-time constraints, as is done in RTSORAC [Dexa94]. In most cases, this involves the creation of (new) object definitions for data in legacy systems, and these object definitions would be annotated with real-time metadata.

For an existing object-oriented system, we believe that definitions would be annotated with real-time metadata. For an existing system, this involves the creation of (new) object definitions and the enforcement of real-time constraints, as is done in RTSORAC [Dexa94]. In most cases, this involves the creation of (new) object definitions for data in legacy systems, and these object definitions would be annotated with real-time metadata.

In our design of an adaptable real-time ORB we chose to use the Meta Object Protocol approach. The preliminary design consists of the ORB baseobject and metaobject specification, their interaction and the interfaces they expose to the users and applications (see Figure 4.1.).

Our model consists of objects in the ORB, distributed objects (CORBA objects) and CORBA Services. We classify these objects into three categories: the distributed objects are the baseobjects (see Meta Object Protocol terminology). All the objects in the implementation of the ORB, services and facilities are metaobjects of metalevel 1. Objects in the ORB and distributed objects that control the execution of baselevel objects and metalevel 1 objects (through a meta interface) are metaobjects of metalevel 2. Consider, for example, our earlier example with Protocol substitution in a CORBA ORB. If Protocol is an object inside the ORB, it is a metalevel 1 object according to the definition above. A metalevel 2 object for the Protocol could be an object that controls which instance of Protocol is currently being active - e.g., Sun RPC or HTTP. Another example could be the following: an object inside the Concurrency Control service is a metalevel 1 object. An object that can control which CC algorithm is used in a particular situation (e.g., priority ceiling, two-phase locking, etc.) would be a metalevel 2 object.

The enumeration of metalevels is useful for establishing the control relationship between sets of objects. If the control/introspect relationship can be
established between two sets of objects, the controller set would be a metalevel object set with respect to the other. For example, we can say that there is a control relationship between distributed objects (CORBA objects) and objects in the ORB, since objects in the ORB have a degree of control over the operation of the distributed objects.

There are some theoretical implications stemming from our choice of base-meta level hierarchy:
- A metaobject can be a full-fledged CORBA object or an implementation object inside one of the services or the ORB (metalevel 2 and 1 respectively).
- Any communication between the services and the ORB can now be called inter-metaobject communication.
- The interfaces to the services and the ORB can now be considered a form of Meta Object Protocols.

There are several reasons for this choice. To achieve an open system, the openness must be a major design criterion from the early stages of the software development cycle. One, system design is hard. It requires the necessary expertise, takes longer to model and build and therefore costs more. In addition, there is no clear compensation benefit for the middleware vendor, since most of the benefits in terms of software extensibility are for the users of the implementation. The competitive edge that can be gained from these value-added benefits of the CORBA implementation have not yet offset the costs of redesigning and rebuilding the middleware implementation from scratch.

4.3. ORB adaptability

There are two types of meta-interfaces that might be used: internal and external. An internal meta-interface is a custom interface that exposes the expected features of the system. An external meta-interface uses delegation to allow an externally-supplied object to control the exposed features. We are interested in an external meta-interface, which would allow third party (e.g., COTS) meta-objects to be used in conjunction with commercial ORBs.

There are three kinds of objects which may be used in a meta-object implementation. A meta-object is an object added to an ORB implementation which provides the needed entry points for control. A control object is an external object which uses the interface provided by one or more meta-objects. A Meta Service (typically an object or set of objects) is used by the ORB (specifically, the meta-objects) to locate the correct control object as needed. We expect a Meta Service to be very similar to a Name Service, with similar restrictions.

We want our design to allow control object interfaces to be defined in standard IDL, but a lasting solution may require extensions to IDL, such as the identification of interfaces that can be affected by control objects. In this case, the goal is to develop a baseline design which would use extended IDL and to add workarounds (if possible) to allow the use of standard IDL. We recognize the difficulties involved in extending IDL.

We have strive to design a metaobject which can be used to control the ORB, services, and user-defined CORBA objects in a uniform manner. This implies that the ORB and service implementations must consist of objects (or at least present such a facade). There are then at least three possibilities for the interface used by a control object:
- Replace an object implementation
- Replace an operation (IDL’s word for method) implementation
- Replace an object attribute

We reject the third as too difficult to manage, because it represents the most egregious violation of encapsulation. We submit that either of the first two are sufficient in that they are capable of providing control over any feature desired, given an appropriate implementation. We find the replacement of operation implementations to be a natural fit with IDL-defined interfaces, so we will explore this option first.

There are three places in the flow of control of an ORB operation call in which we might insert the meta-object:
- Before the called object is selected.
- After selecting the called object but before selecting the called operation implementation.
- After selecting the called operation implementation but before calling it.

Any of these places can be used to replace (in effect) an operation implementation, but we suspect that the first is relatively difficult. We will use either the second or the third depending on which is easiest to implement. The scenario (for insertion in the second place) is:
- Control object selects an implementation for a particular operation O of an interface (from one of the implementations that is available for O).
- Some time later, a calling object calls operation O of some server object S.
- ORB locates an implementation of object S.
- ORB uses Meta Service to find appropriate control object C for O.
- ORB delegates the selection of operation implementation to C.
- ORB calls selected operation implementation.

The scope of a selection can be managed by the control object if enough information is supplied in the delegation call in the above scenario. For example, if the developer wants a selection to apply to a limited set of calling objects (clients), then the ORB has to tell the control
object who is calling the operation. More work is needed to understand what information is required and how best to implement this.

The Meta Service is like a switchboard for connecting to the appropriate control object. Like the Name Service, an ORB with meta-capability cannot really operate without a Meta Service. We are explicitly simplifying the ORB by not requiring it to know which control object it should use, it simply states the desire to call operation O of some instance of interface I and the Meta Service looks up the “I::O” control object. We suspect that only one control object should be used for a particular I::O, but the approach could be extended to select the desired control object if needed. In practice, IDL interfaces are grouped in modules, so the complete key for a control object might be of the form “M::I::O”. For performance reasons, it probably makes sense to tightly link the Meta Service with the ORB, and to cache the results of operation implementation selection (as is done in Smalltalk).

5. Project impact and future directions

5.1. Impact

The major lessons learned during this year’s efforts are listed below:

- Exposing the implementation of the ORB allows for the support of multiple QoS approaches. Since QoS (real-time response, fault-tolerance, tracking quality, etc.) is important for C3, but not well understood, an open implementation will be necessary. Bottlenecks in system performance cannot also be remedied by using faster hardware. Often times, the problem is with a particular protocol that is used in the system. For example, unless some form of priority inheritance is used in a locked-based concurrency control mechanism in a real-time system, unbounded blocking times will occur due to unbounded priority inversion.
- By applying principles of group communication, dynamic changes to system features can take place reliably.
- A well-founded open implementation approach defining a real-time CORBA metaobject protocol provides an approach that can be adopted as an industry standard through the OMG Real-Time SIG. In addition, it would permit greater utilization of COTS real-time distributed object management technology. This benefits C3 systems which must utilize COTS technology to lower costs and meet DoD requirements. Adaptable real-time distributed object management will increase the feasibility of using COTS technology and meet C3 requirements, legacy system interoperability requirements, and lower engineering costs through the effective reuse of COTS technology.

4.2. Directions

The aim in year two of this project will be to concentrate on prototyping and experimentation using metaobject protocols implemented in a reflective version of ILU. This will involve developing a prototype of a representative subset of the metaobject protocols as a new reflective version of ILU. This will touch on the following issues:

- Test reflective ILU with simple applications of metaobjects
- System Monitoring
- Real-time Scheduling Analysis and Support
- Browsing and Inspection of Application and System Objects
- Modification of system state (to support debugging and system control, normally all difficult to do because of need to access system internals).

We will also prepare for a demonstration of some simple “open implementation” capabilities by porting an appropriate example of an object-oriented C3 application. The steps involved are:

- Port the object-oriented, blackboard-based sensor fusion application to the selected real-time operating system.
- Benchmark. Determine its real-time characteristics. Identify real-time requirements that fusion application should meet.
- Port the sensor fusion application to reflective ILU.

We hope to demonstrate a simple level of “open implementation” following these steps:

- Modify application to use “real-time” facilities of reflective ILU (CORBA).
- Benchmark again. Determine if all real-time constraints are being met.
- Demonstrate a simple level of “open implementation” by running the application with two scheduling techniques, defined using the metaobject protocols for scheduling.

The aim in year three is to demonstrate a proof-of-concept for a C3 object-oriented Design Sensor Fusion C3 OO Framework using reflective ILU and the sensor fusion application from year two. This will include designing and implementing a simple CORBA-based object-oriented framework based on our experience from year two, interaction with the DOMIS project, our work with AWACS Extend Sentry program, and the OMG C3I Working Group. The framework will of necessity be
simple since object-oriented framework development is a complex task. The steps that will be involved include:
- Design a C3I Sensor Fusion framework model based on sources identified above.
- Implement the framework.
- Port the sensor fusion application to the framework.

Work in year three will also demonstrate the openness of the sensor fusion application by making changes that normally would be impossible or impractical in conventional approaches. This will involve devising experiments to determine openness of application and reflective ILU as it relates to the following areas:
- Multiple scheduling techniques.
- Multiple networking protocols.
- Multiple OSes.
- Interoperability of application ILU objects on heterogeneous platforms.
- Benchmark to demonstrate real-time capabilities.
- Add in monitoring and scheduling analysis capabilities to ILU using metaobjects.

This future work will allow distributed objects (CORBA) to be used in many new problem domains. By opening up the implementation of CORBA ORBs and services, this gets the ORB vendor out of the loop and makes it possible for a third party market to develop in CORBA metaobjects. Users could then tailor CORBA to their needs instead of waiting for the commercial ORB market to catch up to their requirements.

We will learn how to design systems using metaobject (open implementation techniques) and object-oriented CORBA-based frameworks for C3. Currently, a number of programs are using CORBA, attempting to use it in real-time, C3 systems. The use of metaobjects to achieve an open implementation may also lead to a convergence of non-real-time and real-time CORBA implementations since the difference between the two could be specified using different metaobjects. This also may have impact on other types of “closed” system software. With greater access to the internals (an open implementation) of compilers, language runtimes, operating systems, the potential of COTS in DoD systems acquisition may be finally realized.

References


