CORBA-based Real-time Trader Service for
Adaptable Command and Control Systems

Steven Wohlever, Victor Fay-Wolfe, Bhavani Thuraisingham, Richard Freedman, John Maurer
The MITRE Corporation
Bedford, MA 01730, USA
{wohlever, wolfe, thura, rnf, johnm}@mitre.org

Abstract
This paper describes an approach to building adaptable real-time command and control (C2) systems. In
particular, it presents an overview of the Adaptable Real-Time Distributed Object Management (ARTDOM) project
in progress at the MITRE Corporation. This project is currently developing real-time extensions for the Common
Object Request Broker Architecture (CORBA) Trading Object Service. The goal of the project is to demonstrate
how current C2 systems can be more easily upgraded and made more adaptable by using emerging distributed
computing technology. This goal is being accomplished by investigating and developing real-time middleware that is
reflexive (i.e., capable of examining its current state and processing demands) and self-adapting (i.e., capable of
reconfiguring itself based on its reflective findings).

1. Introduction

Significant portions of today’s real-time command and control (C2) systems are becoming either functionally
inadequate or logistically insupportable. Furthermore, due to budget limitations, new developments of next generation
real-time C2 systems are not always possible. Therefore, current real-time C2 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 C2 systems into the extensible systems required for the future.

Between 1994 and 1996, MITRE carried out the Evolvable Real-Time Initiative project under the corporation’s
Mission-Oriented Investigation and Experimentation (MOIE) program sponsored by the United States Air Force.
This project developed an approach that would enable current real-time systems to evolve into the systems of the
future [Bensley94, Bensley95, Gates96]. The evolution approach that resulted from the three year effort is to use
near-term system upgrade activity to put a new infrastructure in place, with an emphasis on transitioning to
open architectures that are modular and free from proprietary or unnecessarily complex software designs. The
new infrastructure was designed to accommodate new upgrades more easily and allow the extensible system
architecture to ultimately replace the current hardware and software architecture.

As a demonstration of this, the project team designed and implemented an infrastructure and data manager for C2.
We used an implementation of the Object Management Group’s (OMG) Common Object Request Broker
Architecture (CORBA) [OMG98a] in conjunction with a real-time operating system to integrate the various
distributed components and provide some initial support for meeting real-time constraints. CORBA provides an object-
oriented standard for middleware in distributed systems. The standard specifies a basic framework for applications to
access objects in a distributed environment using a client/server paradigm. It is designed to allow a programmer to construct object-oriented programs without regard to traditional object boundaries such as address
spaces, programming language, or the location of the object in the distributed system. See [OMG98a] for a more
complete discussion of the CORBA specification. Also, in the context of our research, an application running on a
real-time system must meet its timing constraints (e.g., deadlines, periods of execution, etc.) in order for it to be
correct.

While the Evolvable Real-Time Initiative project did yield an approach for more easily upgrading C2 systems, it
did not produce an infrastructure that would allow these
systems to adapt to new runtime requirements. In order to address this issue, MITRE started a second MOIE project named Adaptable Real-Time Distributed Object Management (ARTDOM) in 1997. This is a three-year project that is expected to be completed at the end of 1999. The goal of the ARTDOM project is to design flexible, adaptable distributed object management systems for C2 systems. Such an adaptable system can switch scheduling algorithms, communication protocols, and other policies and system components depending on the need and the environment. This paper describes our approach to developing an ARTDOM infrastructure for C2 systems. In particular, our work carried out on this project during 1998 is presented. In section 2 we provide some background information on evolvable real-time command and control systems project which eventually led to the ARTDOM project.

2. Evolvable Real-time Systems

ARTDOM evolved from our initial project: Evolvable Real-time Command and Control Systems (ERTCCS). ERTCCS project was a three year project between 1994 and 1996 and addressed issues on evolving complex command and control systems such as AWACS. We identified requirements for complex real-time command and control systems, developed an infrastructure and data manager and hosted tracking application on the infrastructure. This work as been published in numerous papers and was one of the firsts to introduce real-time CORBA. ERTCCS then evolved into ARTDOM. Some of the specific details of the project follow.

During the first year our work focused on developing the infrastructure requirements. In particular, we first examined the features of current real-time C3 systems and then determined the systems of the future. Next, we defined various candidate architectures for future systems. Finally, we developed requirements for the infrastructure. The main focus was on operating systems, data management systems, and communication systems.

During the second year we have designed and implemented portions of the infrastructure, data manager, and the multi-sensor integration application. Our infrastructure has focused on the operating system and its associated extensions. Note that while the data manager is part of the infrastructure as far as the application is concerned, it can be regarded as an application by the operating system. Since an evolvable design was a major consideration, we were influenced by object-oriented design and implementation approaches as well as our initial investigation of real-time issues for distributed object management systems. Our infrastructure is essentially a collection of objects interacting with each other. This way existing applications as well as new applications could be encapsulated as objects.

During the third year of the project we ported the MSI application to the Lynx PC environment. The application made us of the data manager to store and retrieve tracks. Also, during the third year we integrated the different modules (e.g., the data manager, application, and infrastructure) using a distributed object management approach. In particular, we used Xerox’s ILU which is a CORBA-based system to integrate the modules.

3. Adaptable Middleware

3.1 Overview of the Project

Section 3 describes the details of ARTDOM. During the first year, the ARTDOM MOIE focused on investigating how to provide an open implementation for an ORB such that the ORB could be customized to provide the desired level of support for real-time, fault-tolerant performance. In this context, an ORB has an open implementation if it provides interfaces that a developer can use to modify aspects of the ORB’s underlying behavior. The ORB selected for this investigation was the SYLU ORB from Stanford University, a Python-based ORB implementation derived from Xerox PARC’s ILU ORB. The 1997 work was based on a research ORB (i.e., a non-COTS ORB) with the goal of investigating and developing techniques for providing a statically- and dynamically-configurable ORB. For example, the prototype ORB developed in 1997 provided interfaces for changing the messaging protocol used by the ORB.

The focus for work during 1998 shifted to investigating how existing COTS ORBs can be extended by non-intrusive techniques to provide better real-time performance. One reason for this shift is that fact that commercially-available ORBs do not have sufficiently open implementations. Also, as purchasing trends in the DoD move towards looking to the COTS market to provide hardware and software, it is beneficial to investigate how generic COTS ORBs can be extended to provide support for specialized needs such as real-time computing.

During the second year, the MOIE focused on developing extensions for commercial ORBs that improve the abilities of the ORBs to support real-time computing. The method we selected to do this was to develop an adaptable Trading Object Service, a real-time extension to the CORBA Common Object Service (COS). In particular, we developed a Real-Time Trading Object Service which, when coupled with a traditional Trading Object Service, provides a mechanism by which CORBA clients are bound.
to CORBA servers that can best meet their real-time requirements.

In section 3.2 we discuss our work during 1997. Then the details of our work in 1998 will be discussed in section 3.3, 3.4, and 3.5.

### 3.2 Adaptable ORBs

This section describes the initial work carried out on ARTDOM in 1997. 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.

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.

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].

MITRE also developed a demonstration. The demonstration was run on networked 386 machines. The operating system used was LynxOS 2.4, a real-time version of UNIX. The programs were written in Python, a dynamic, rapid-prototyping interactive language, with support for scripting, pattern matching, object-oriented programming, and many other “speciality” areas. The GUI was implemented using Tkinter, a Python adaptation of Tk, a widget set initially developed to run with Tcl. We used SYLU, a Python implementation of Xerox’s ILU developed by Scott Hassan at Stanford University, for our ORB. SYLU is the only CORBA ORB on the market, written in a dynamic language, such as Python, with complete source code available. We first ported Python to Lynx OS. Then SYLU was built on the Python virtual machine on top of Lynx OS. While Python is an object oriented language, SYLU did not use all of its capabilities. SYLU’s object structure was predominantly flat, using almost no inheritance. The only exception is the Transport/Protocol layer which contains a superclass for both the transport and protocol, making it possible to derive different protocols and transports for a particular implementation.

The project demonstrates dynamic server adjustment to match a particular client’s needs. As an example, a server running in RPC mode would get a signal to change its mode to HTTP so it may service a different, (higher priority) client. To the client, this change would be seamless. The project demonstrates this concept with the exception that the server did not dynamically change its mode. If a new client required a different mode, the server was shut down once it became idle. The server was then restarted in the appropriate mode. Although shutting the server down and restarting did not demonstrate the dynamic aspect of the concept, the change was transparent to the client.

The demonstration consists of four components:

- The server process, which can start in RPC or HTTP mode.
- The RPC client process that accesses the server with remote method calls.
- The HTTP client (Netscape browser).
- The controller/GUI, which allows the system to change the mode of the server between RPC and HTTP. It also graphically displayed a network package being sent from the client to the server.

The four components will now be explained in more detail.

### 3.3 Adaptive Binding

The remaining three subsections 3.3, 3.4 and 3.5 describes the essential points of our current approach. Some of our preliminary ideas were also presented at the work in progress session at RTSS.

To demonstrate the utility of adaptivity in CORBA, the ARTDOM MOIE is developing techniques for providing an adaptive binding mechanism in CORBA. In this context, binding refers to the process by which a CORBA client obtains a reference to a CORBA server object and binds that reference to some locally accessible construct (e.g., a reference can be bound to a C++ pointer). There are a number of methods for doing this binding in CORBA:

1. `CORBA::ORB::resolve_initial_references()`
2. Vendor-specific mechanism
3. Read text version of object reference from text file and convert to object reference
4 CORBA Naming Service

5 CORBA Trading Object Service

Methods 1 and 4 are basically the same in that they do the bindings based on the name of the CORBA object the client wishes to bind to. These methods require the use of an additional object narrowing mechanism, which “casts” a generic CORBA object reference into a reference for a specific type of CORBA object. Method 2 is typically similar to methods 1 and 4 although some vendor-specific binding mechanisms can do the binding based on criteria other than the object name. For example, Iona Technologies’ Orbix ORB [1] can bind based solely on the type of CORBA object the client wants or based on the object type and one (or a combination) of the following characteristics:

1. The host machine on which the client wishes to look for the object
2. The server process in which the client wishes to look for the object

Method 3 makes use of the fact that the CORBA 2.0 specification includes the definition of a character string representation for CORBA object references and mechanisms for converting from a reference to a string and a string to a reference. These strings can be converted into generic CORBA object references that can then be narrowed to a specific type of CORBA object reference. Method 5 is the method of interest for the ARTDOM MOIE. The Trading Object Service and the Naming Service are CORBA Object Services, a set of services specified by the OMG that have been deemed useful for a wide variety of distributed applications. Whereas the Naming Service does bindings based on object names, the Trading Object Service does bindings based on the characteristics the client is looking for in its server.

3.4 CORBA Trading Object Service

In order to provide an adaptive binding mechanism for CORBA, the MOIE is developing real-time extensions for the CORBA Trading Object Service [1]. The OMG Trading Object Service facilitates the offering and the discovery of instances of services of particular types. A Trader is a CORBA object that supports the trading object service in a distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities. Advertising a capability or offering a service is called “export.” Matching against needs or discovering services is called “import.” Export and import facilitate dynamic discovery of, and late binding to, services. To export, an object gives the Trader a description of a service and the location of an interface where that service is available. To import, an object asks the Trader for a service having certain characteristics. The Trader checks against the service descriptions it holds and responds to the importer with the location of the selected service’s interface. The importer is then able to interact with the service.

The Trading Object Service acts as a negotiator of sorts. There are a sequence of events that allow a client, or service importer (I), to request a reference to a server, or service exporter (E), from the Trader (T). For example, the Trader is managing CORBA servers that represent network printers. It should be noted that a Trader is not limited to managing servers of only one type. Also, multiple Traders, each managing its own set of servers, can be linked together such that requests sent to one Trader can filter through to other Traders.

In step 1, a CORBA server registers itself with the Trader, indicating it is ready to provide a particular type of service (e.g., printing) to interested clients. Service types that are registered with the Trader typically include static properties (e.g., color or black and white printing, printer location) and possibly dynamic properties (e.g., current length of job queue, amount of paper).

In step 2, a client queries the Trader for a reference to a registered server that can provide a particular service. This query can also specify constraints (e.g., client may want a color printer that has enough paper to complete the print job). The Trader then compares the client’s constraints to the properties of the registered servers. This step may also require that the Trader query the registered servers for the current values of dynamic properties if appropriate (e.g., need to query printers to determine how much paper they have left). The Trader then returns one or more suitable CORBA object references to the client.

In step 3, the client chooses one of the object references provided by the Trader and begins its interaction with the specified server.

The Trading Object Service consists of a number of components that implement specific interfaces. These interfaces are as follows:

- **Lookup**: Allows the user to query the Trader for offers which satisfy particular conditions.
- **Register**: Allows offers to be added, modified, and withdrawn.
- **Admin**: Allows configuration settings to be modified.
- **Link**: Supports federation between Traders.
- **Proxy**: Essentially exports a lookup interface for delayed evaluation of offers, and may be used, for example, for encapsulating legacy applications, or as a kind of service offer factory.
**ServiceTypeRepository**  
Allows the user to register service types with the Trader.

There are a number of Trader configurations possible. These are based on the interfaces that are supported, i.e., the components that are implemented in the Trader, they build upon the capabilities of the simpler configurations. The simplest of these is the `query` Trader, which just supports the `Lookup` interface. This is useful, for example, where a Trader is pre-loaded and optimized for searching. The `simple` Trader adds support for adding, deleting, and querying offers. The `stand-alone` Trader adds support for the administration of the Trader’s configuration settings. The `linked` Trader adds support for a federation of Traders. The `proxy` Trader is a stand-alone Trader that supports the proxy functionality, which may be useful for legacy code, or where offers need to be dynamically created. The `full-service` Trader combines all the functionality of the above configurations. An implementation of the OMG Trading Object Service specification conforms to one of these configurations of Traders. On the MOIE project, we are building a Real-Time Trader that conforms to the `simple` configuration.

**Service Offer Types**

A service offer type is a runtime data type describing a particular service. A service offer type consists of a sequence of property definitions, plus information on the IDL type of target objects.

Service offer types would typically be agreed upon across a number of applications, or at a global level within an organization. These property definitions allow for type checking (for safety), while being flexible enough to deal with optional information. For example, a common set of printer properties could be defined world-wide. Properties may be `mandatory`, in which case any instance of the offer type must supply values for this mandatory property. They may also be `read_only`, in which case their initial value may not be modified later. Properties can also be both `mandatory` and `read_only`. For example, a printer may have a mandatory, read-only property “name” which is its identifying string, and a mandatory property “location,” which is also a string. A non-mandatory property could be “page_per_min.”

Service offer types may also be derived from supertypes. In this case, they must support all of the properties of their supertypes (including mandatory or `read_only` modes, though they may add to them). They may be used wherever their supertypes are expected. For example, a “multiple_page_size_printer” might be derived from “printer,” and define a property specifying the different page sizes possible, but it could still be returned in a general query for printers. Next, the programmer should invoke `add_type`, and pass it the relevant parameters.

**Service Offers**

Once the offer type is created, servers may export offers that correspond to this type. These contain values for the property data type, and an object reference for the relevant object. The server then invokes `export` method.

**Query**

After an offer has been entered, a client can then query the Trading Object Service, based on the properties previously defined. For example, the query might look something like the following: "looking for a printer where page_per_min > 50". The client is then returned zero or more offers. The client can then use the object reference in an offer to invoke on the server.

**Dynamic Properties**

There is a special category of properties called “dynamic” properties, which are evaluated by the Trading Object Service at runtime, typically via a callback object to the original server. For example, a printer could have a dynamic property “size_of_queue.” During a query involving “size_of_queue,” the Trading Object Service would query the printer server for the current value of this property. This is slower to evaluate, but offers significant flexibility to applications.

At a code level, a dynamic property is a normal property, whose type is a `CosTradingDynamic::DynamicProp` structure. This contains a pointer to an object which implements the interface type `CosTradingDynamic::DynamicPropEval`, which has one operation `evalDP`. The object is normally created in the relevant server. When evaluating a query, the Trading Object Service invokes `evalDP` on the object in the property to get the runtime value.

**Federation**

The Trading Object Service may also be federated, using one-way links, so that several Traders may pool information. For example, if a client is looking to print a book, its local Trading Object Service may not have a matching offer. If it is federated with a Trading Object Service that provides a “yellow pages” directory for the local district, it can forward the query, and return any results to the client.

A client may modify the Trading Object Service’s default link policy for an individual query, so it can limit the amount of searching being done. The value used is the minimum of the Trading Object Service’s own policy, and that of the query.
3.5 Real-time Extensions to Trading Object Service

The goal of this MOIE project during 1998 was to demonstrate how to add extensions to COTS ORBs such that the ORBs can better support real-time computing. To this end, MITRE, in collaboration with the University of Rhode Island (URI), has developed a Real-Time Trading Object Service for the MITRE ClassiX/COOL testbed system. The Real-Time Trading Object Service was developed as an extension to the CORBA Trading Object Service. That is, the Real-Time Trading Object Service relies on a standard Trader (e.g., MITRE ClassiX/COOL Trader, IONA OrbixTrader, etc.) to provide the capabilities described in Section 3.3.1. The Real-Time Trader builds on these underlying capabilities by adding real-time analysis that is used to determine which of several eligible servers a particular client should use in order to meet its real-time requirements.

The proposal for this MOIE states that the project will investigate and develop real-time middleware that is reflexive (i.e., capable of examining its current state and processing demands) and self-adapting (i.e., capable of reconfiguring itself based on its reflective findings). In particular, the proposal cites real-time criteria for making adaptation decisions. Reflection includes determining current system load and resource requirements, and self-adaptation includes allocating resources so that real-time constraints can be met. The reflection and self-adaptation should be handled by the middleware so that it adheres to sound real-time principals and is transparent to the clients and servers.

We developed algorithms for real-time reflective self-adapting load allocation in distributed real-time systems, and mechanisms to implement the algorithms in standard distributed system middleware (in this case, the CORBA Trading Object Service). The reflection algorithms provide real-time analysis of various parts of the distributed system to dictate whether they can meet the real-time requirements that the application imposes on them. The self-adapting algorithms provide load allocation to parts of the real-time system that can guarantee that timing constraints will be met.

Note that real-time adaptive binding is not load balancing. Load balancing techniques seek to balance load to optimize performance criteria such as minimal queue lengths, minimal average wait time of tasks, etc. However, a probabilistic load fitting approach can maximize the ability to find resources, such as CORBA servers, that can fit load to meet timing constraints. For instance, a load-balanced system will mean that it is less likely that CORBA servers can meet the deadline of a task with high processing demands. This is because all CORBA servers will be approximately evenly loaded, making it less likely that there will be servers with low enough utilization to meet the deadline of the resource-intensive task. However, a load-fitted system can use reflexive real-time information provided by the reflexive ORB to adapt load allocation to maximize the likelihood of real-time constraints being met.

To better understand the goal of the Real-Time Trading Object Service, we need to consider a task’s execution time, its deadline, and the slack time for the task in the current schedule. We are assuming EDF scheduling on each node. The question is: on which node do we place the new task to yield the best chance that future tasks will meet their timing constraints? Due to its deadline, it would be the third task in the schedule on both Node 1 and Node 2. However, its placement will reduce the slack times of the current third and fourth task on the node it gets placed upon. Choosing which node to place the new task so that future timing constraints have the highest probability of being met is non-trivial. It is this problem that our 1998 work addressed. We now provide details on our approach.

MITRE Chorus ClassiX/COOL Testbed and Trading Object Service

The testbed for the 1998 work is based on two Intel 80486 PCs running the Chorus ClassiX r3.1 operating system and the Chorus COOL ORB v4.2. ClassiX is a componentized, microkernel operating system which can be scaled (via features which are incorporated into the boot kernel) to meet the system user’s particular needs. Development for the testbed takes place in a cross-platform development environment: a Sun Sparcstation running Solaris 2.5 provides the host development environment (and serves as the boot server) while the PCs provide the target execution environment.

As part of the prototype to be discussed in the remainder of Section 3, the MOIE implemented a subset of the Trading Object Service on the ClassiX/COOL testbed. The MITRE Trading Object Service provides interfaces for adding service types, registering servers, and requesting references to all servers of a specific type. The Trader does not currently implement support for requesting servers based on constraints. This functionality is provided by the real-time extensions to the Trading Object Service.

Real-Time Trading Object Service

The CORBA Trading Object Service is designed to provide bindings to distributed objects based on application-specific criteria. Since the goal of this MOIE project was to design real-time adaptive capabilities, our objective was to determine the best real-time criteria for use in the Trading Object Service. More specifically, we chose to implement...
a trading criterion that yields the highest probability that timing constraints will be met. Consider a distributed system consisting of $N$ nodes, each of which has CORBA servers. Let us consider some moment of time when some clients have been already bound to some servers but have not yet completed their work (the trivial case when there is no such client could be also analyzed by the algorithm). These servers are currently executing some service requests while other service requests are queued. The Trader contains the list of all service requests currently bound to each node. These lists contain the following parameters for each service request:

- **Execution Time** (specific value for each service),
- **Absolute Deadline** (which specifies timing constraint and the priority under the EDF priority assignment mechanism),
- **Worst Case Completion Time** (calculated based on execution of higher priority tasks and maximum possible blocking),
- **Slack Time** (defined as Absolute Deadline minus Worst Case Completion Time minus Current Time).

When a Trader receives a new service request from a client, it must determine to which server the client’s request should be bound. The decision is made based on the following steps:

1. Identify all the nodes (and servers) where the requested service is provided.
2. Among these nodes, choose those where the requested service will meet its timing constraints and will not violate any timing constraints of the previously bound requests. At this stage, the Trader should also identify all possible deadlock situations and eliminate such possibilities by eliminating particular nodes from the list of possible “service providers.”
3. Among the remaining (after the above selection) candidate nodes, choose the one which provides the highest probability that the future requests will be schedulable and also will not violate the schedulability of the previously bound requests. This step requires the calculation of the described probability for each candidate-node. That is, we assume the binding of the new request to each candidate node in turn and calculate the probability of future tasks meeting their timing constraints for every possible binding. The RT Trader selects the binding that yields the highest probability that future tasks meet their timing constraints.

The first step is a trivial selection process based on which nodes host candidate servers. The second step is a calculation of the worst case completion time for each affected service request (as a sum of all execution times of all higher priority requests and the worst case blocking time). The prevention of deadlock has been discussed in the literature before, so it is not presented here.

In this MOIE project, we concentrated on developing the algorithm used in the third step for calculating the probability that the future requests will meet their timing constraints such that they will not violate the timing constraints of the previously bound requests. We base our analysis on consideration of some virtual requests that may arrive in the future. We assume that the probability distributions of arrival times and deadlines of such requests are known. These probability distributions could be obtained by accumulating the statistics of requests during run-time, or they could be specified by the Trader Server architect. Theoretically, the number of the virtual future requests is unlimited, but to limit the Trading Object Service overhead, one needs to limit the number of the virtual future tasks to be taken into account in the analysis. Without loss of generality, we restrict the algorithm we used in the RT Trader to one looking ahead to one future request. Our approach could be simply extended to as many future requests as necessary.

We now present the formal problem statement and description of the algorithm. A more detailed, real world example illustrating the algorithm at work will be presented in a future MITRE technical report.

**Formal Problem Statement:** The RT Trader seeks to calculate the probability that a future service request will meet its timing constraints and will not violate the timing constraints of previously bound service requests. Detailed algorithm is presented in a MITRE Technical Report [Maurer98]. This algorithm was coded and built into the Real-Time Trading Object Service so that with inputs consisting of the frequency distribution of method calls, the probability of deadlines, and the probability of arrival times, the RT Trader can bind the client to the node in the distributed system such that timing constraints are met and there is the highest probability that future tasks will meet their timing constraints.

**4. Conclusion**

The goal of the ARTDOM project is to investigate how existing middleware products such as CORBA can be extended using reflexive and self-adapting mechanisms to provide support for real-time computing. This research will permit greater utilization of commercial real-time distributed object management technology in C2 systems, which must utilize commercial technology to lower costs and meet DoD requirements. Adaptable real-time distributed object management will increase the feasibility of using commercial technology to meet C2 requirements, legacy system interoperability requirements, and will lower engineering costs through the effective use of commercial technology.
The technical approach for future work on ARTDOM will be to modify at least two applications to use the RT Trader. The first application will be a set of benchmarks that will allow us to accurately understand the behavior and performance of the RT Trader as compared to a baseline system. Following experimentation with this system, we plan to port an Air Force C2 application to use the RT Trader and demonstrate its operational utility.

References


