The Design of an Open System with Distributed Real-Time Requirements

Roman Ginis and Victor Fay Wolfe
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
`lastname@cs.uri.edu`

J. J. Prichard
Department of Mathematics
East Carolina University
Greenville, NC 27858
`prichard@cs.ecu.edu`

Abstract

This paper describes some of the US Navy's New Attack Submarine C3I system - an open system with distributed real-time requirements. It also presents our research design and prototyping efforts that address these requirements. It concludes by discussing several areas where further research is needed to allow open systems to better support real-time applications.

1 Introduction

With the advent of U.S. government mandates for *open systems* in their computer system procurements, even large distributed real-time applications will need to meet the requirements of using commercial-off-the-shelf (COTS) technology with interfaces that adhere to widely accepted standards. As part of our work with the US Navy's UnderSea Warfare Center, the US Navy's Research and Development Labs (NRad), and Loral Federal Systems, we have assisted in specifying the requirements and a possible design of aspects of the US Navy's New Attack Submarine (NSSN) Command Control Communication and Intelligence (C3I) system - a recently-specified open system with distributed real-time requirements. This paper describes some of the NSSN C3I distributed real-time requirements, our research design and prototyping efforts that address these requirements, and several areas where further research is needed to allow open systems to better support real-time applications.

Section 2 presents the subset of NSSN requirements on which we focused, including its use of legacy subsystems and their open integration using the CORBA distributed computing environment and real-time databases to support data transfer. Section 3 describes our high-level architecture design of the use of CORBA and real-time databases to meet these requirements. Section 4 focuses on our lower-level design of techniques for the integration of real-time databases into a CORBA environment using open system techniques. Section 5 describes the results of our prototyping efforts. Section 6 summarizes and indicates several significant areas of further research that are needed to fully integrate real-time support into open systems.

2 Application Requirements

The specification of the C3I system for the NSSN was released in mid 1995. It calls for the system to be open, distributed, real-time, and for it to integrate legacy subsystems such as sonar, radar, combat control, and navigation. That is, the major design effort in the new C3I system will be a network and software to link existing legacy subsystems in such a way that as the legacy systems are upgraded, they can be easily replaced without affecting the other components of the C3I system. The development effort is mandated to be "open" by using COTS hardware and software whenever possible. Note that this is a significant change from past military procurements which often opted for proprietary solutions. The use of open systems is expected to reduce the cost of the system by leveraging commercial development and to facilitate upgrades of system components over time.

Databases. The NSSN specification requires that most data communication among subsystems be done through databases (to facilitate data sharing and data consistency). Sensor data from subsystems such as sonar and radar have real-time "latency" requirements that specify when the data must be used. Furthermore, many of the transactions that operate on the sensor data will themselves be time-constrained. The NSSN specification recommends the use of real-time databases to support this sensor data and its access.

Due to the open systems mandate, the interface to the database systems must adhere to a widely-accepted standard. The NSSN specification recommends SQL as that standard. SQL is widely-accepted as the primary relational database standard interface; most commercial vendors provide support for it in their products. The standard was first established in 1986 by the American National Standards Institute (ANSI). The latest revision, which constitutes the current standard, was approved in 1992 [1]. The SQL standard consists of three subparts: a data definition part, a data manipulation part and a data control part. The data definition language is used to define the structure and constraints of the database schema. It includes calls to create tables and specify constraints on values in the tables. The data manipulation language is used to manipulate the data defined by the data definition language. It includes calls to insert, delete, modify data, and to select data values based on predicates. The data control language is used for managing access to the database. ANSI has several subgroups working on extensions to SQL including : object-orientation, embedded SQL, security, condition handling, persistent stored modules, transactions, temporal data, multi-media data, and most recently real-time.

Real-time extensions to SQL (RTSQL) were proposed to ANSI and ISO by the Navy's Next Generation Computer Resources (NGCR) Database Standards Interface Working Group (DISWG) in 1995 [2]. This proposal has resulted in a working group in these standards organizations to address incorporation of the proposed real-time features into the official SQL standard. The extensions for real-time include syntax and semantics within SQL's framework to specify timing constraints on data manipulation statements (such as deadlines and earliest start times), timing constraints on data (temporal consistency validity intervals), and directives to facilitate predictable execution (such as directing main-memory storage and bounding table sizes). Although the NSSN specification does not mandate the use of RTSQL, because it not an approved standard yet, it does recommend SQL for data management and recommends real-time data management for certain data. We therefore, have opted to include the use of RTSQL in our design and prototyping efforts (see Section 4).

Distributed Computing Environment. To integrate the legacy subsystems, the NSSN specification calls for a distributed computing environment that adheres to the Object Management Group's (OMG) CORBA (Common Object Request Broker Architecture) standard [3]. The OMG is a consortium of over 500 software providers and users, including most of the major software vendors. They originally came together in the late 1980's to establish a standard way for heterogeneous client and server components to interoperate in a distributed system using the object-oriented paradigm. The first CORBA standard was established in 1991 and almost immediately there were many products adhering to it. Subsequent additions to the standard have been made and products soon followed. CORBA has been popular in document management and business applications, and now, as mentioned in Section 1, it has been specified as the distributed computing environment standard for the NSSN.

CORBA is designed to allow a programmer to construct object-oriented programs without regard to traditional object boundaries such as address spaces or location of the object in a distributed system. That is, a client program should be able to invoke a method on a server object whether the object is in the client's address space or located on a remote node in a distributed system.

Two major components of CORBA are its Object Request Broker (ORB), and its Interface Definition Language (IDL). The CORBA ORB essentially enables communication between clients and remote server object implementations. The server object's implementation is the code and data that actually implements services of the object. The ORB provides all services that: locate a server object implementation for servicing a client's request; prepare the object implementation to receive the request; and communicate the data making up the request.

CORBA IDL is a declarative language that describes the *interfaces* to server object implementations, including the signatures of all server object methods that are callable by clients. The IDL grammar is a subset of ANSI C++ with additional constructs to support the method invocation mechanism, such as `in` and `out` specifiers for parameters. IDL also specifies C++-like exception raising and handling. IDL *does not* provide syntax for implementing methods: an IDL binding to the C language has been specified for that purpose. Other language bindings are being developed.

The process of implementing a client and server object is shown in Figure 1. The IDL specification is processed by an IDL compiler, which generates a header file for the CORBA object, stub code for linking into the client, and skeleton code for the server object. The client stub contains code that hides details of interaction with the server from the client code. Client stubs stand in for normal method calls by transparently directing normal-appearing C++ method requests into the ORB. Server skeleton code is used by the ORB in forwarding method invocation requests to the server, and in returning results to the client.

Currently, there is no support for real-time requirements in CORBA. Our research group, in collaboration

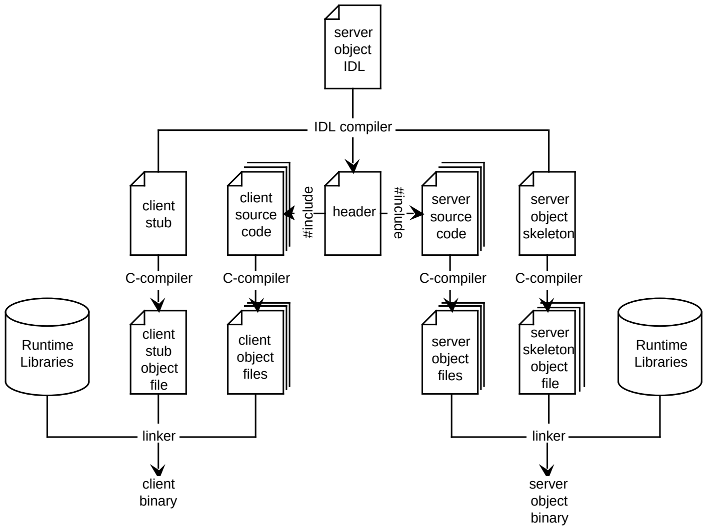

Figure 1: CORBA IDL Compilation Process

with researchers at MITRE Corporation, have presented white papers outlining real-time extensions to CORBA [4, 5]. These extensions include the ability for a client to time constrain its method calls, for the server to specify its worst case timing behavior to clients, and for the ORB to perform end-to-end real-time scheduling across the distributed environment. More elaborate real-time CORBA extensions call for the ORB to offer guarantees of timing behavior by employing a priori reservations of resources required by the client. The OMG is forming a Special Interest Group (SIG) in 1996 to examine real-time extensions to CORBA; our proposed extensions will be considered as part of that effort.

The NSSN requirements, which call for CORBA and for real-time behavior, indicate a need for real-time CORBA. However, since real-time is not part of the OMG's current standard nor had the SIG addressing real-time been formed when the specification was published, the NSSN requirements don't specifically recommend real-time CORBA be used. In our design, which we present in the remainder of the paper, we do not seek to include real-time CORBA features, but also are careful not to preclude their use once these features are solidified by the OMG.

The combination of the NSSN requirements we have outlined in this section indicate the need to incorporate a COTS real-time database, with a RTSQL interface, into a CORBA environment while still maintaining real-time functionality, such as predictability and timing constraint enforcement. We discuss the design for doing this in the remainder of this paper.

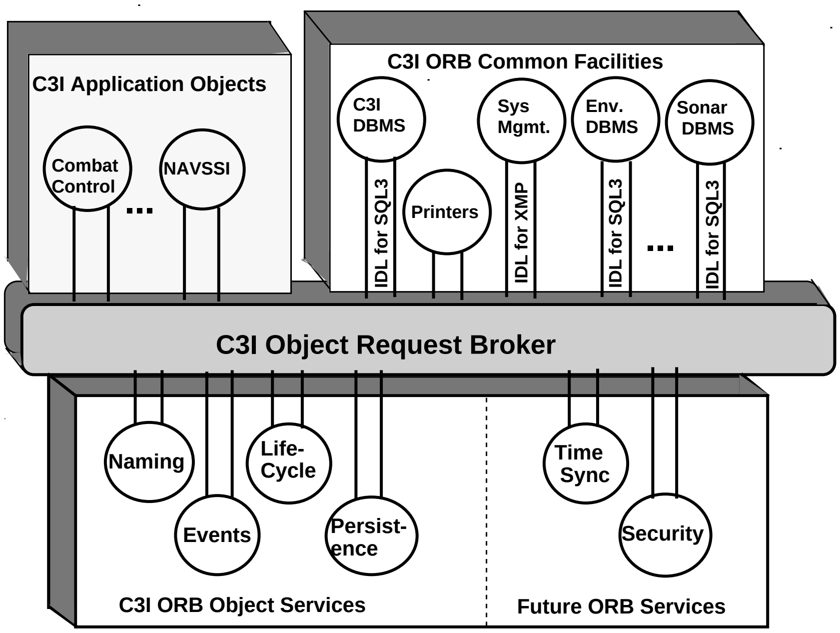

Figure 2: NSSN C3I CORBA Architecture

3 High-Level CORBA Architecture For the NSSN

Our NSSN C3I architecture design makes extensive use of CORBA to provide high-level, open, software interoperability. During the preliminary NSSN C3I architecture specification, there were unofficial estimates that 90% of the software used could be Commercial-Off-The-Shelf. In our design, ideally, the remaining 10% should be CORBA-consistent "wrapper" software. This way, even non-COTS software can still maintain some "openness" by adhering to a distributed computing environment standard.

Implementing a CORBA-based NSSN architecture requires that as many computing nodes in the NSSN C3I system as possible come with CORBA support. As outlined in Section 2, CORBA support means that the node comes with a IDL compiler, a CORBA library, and possibly CORBA daemon processes that are adequate to support CORBA application object servers. Since the NSSN specification calls for POSIX-compliant operating systems to be employed, and CORBA systems for POSIX operating systems are common, a COTS CORBA environment should be possible for most nodes. In those nodes where CORBA support is not cost effective, a proxy CORBA interface on another node (a node with CORBA support) may be possible. Each service provided by the node to the C3I system (e.g. database management, system management, or legacy functionality such as navigation information) would be implemented by a CORBA server object. A CORBA-style diagram of a possible CORBA-based C3I architecture is shown in Figure 2.

In Figure 2, users of data are CORBA clients represented by C3I application objects: Combat Control and NAVSSI (Navigation) are examples. Common Facilities are "servers" in the CORBA system. There are several things to note about the servers in the figure. First, note that database management components are represented as the servers (e.g. Sonar's DBMS is represented). This is done to adhere to the NSSN requirement that essentially states that all subsystems should communicate through the common data services provided by the subsystems. Thus, the DBMS components of subsystems should be servers to application components in the system. Other functionality may also need to be provided on a subsystem's interface besides DBMS functionality (e.g. System Management functionality). For instance, Sonar may provide other interface functionality besides just data generation for some part of the C3I system. If so, this other functionality could be represented either by a separate Sonar server object, or by a single Sonar server that specifies both DBMS and the other functionality in its IDL. Second, note that the interfaces to the servers are IDL that reflects a standard: some form of SQL (e.g. SQL3) for the database managers and XMP for the System Management. We discuss the use of standards in the IDL in Section 4.

The lower part of Figure 2 are the object services. The left four services (naming, events, lifecycle, and persistence) are standard CORBA 2.0 services that the NSSN will likely utilize. They will be COTS and provided with the CORBA 2.0 system. In particular, persistence can be valuable in terms of backing up objects for fault tolerance. The right two services are NSSN requirements, time synchronization and security, that CORBA has working groups for, but are not in the standard yet.

Figure 3 shows a more detailed view of the use of CORBA. In this figure, the JMCIS target tracking subsystem and the Combat Control subsystems have integrated CORBA along with the C3I system just described from Figure 3. Since part of the CORBA 2.0 specification is for ORB-interoperability, the notion of three separate systems, each with their own CORBA implementation interacting, is viable.

A CORBA implementation of the JMCIS system would look something like the top third of Figure 8, with its own ORB, application objects, etc. In this design, one of the JMCIS application objects would be an application object that represents the interface to the rest of the Combat Control subsystem (shown as the JMCIS/CC two-shaded object between the JMCIS ORB and the CC ORB). This application object would access JMCIS for Combat Control. The top part of Figure 3 essentially forms a wrapper for JMCIS. This is because requiring all access to JMCIS be done through application objects on the JMCIS ORB, mandates that other subsystems, like Combat Control, access JMCIS services through the IDL of the JMCIS components on the JMCIS ORB. If JMCIS were to be replaced with another subsystem that preserved the same IDL interfaces, conceptually, nothing in the rest of Combat Control system would have to changed. That is, the JMCIS/CC application object would simply call services on the new subsystem's ORB. If other subsystems need to access JMCIS, such as NAVSSI, they also would be represented as application objects on the JMCIS ORB.

The middle part of Figure 3 shows Combat Control being implemented as its own CORBA architecture. It may have been more accurate to represent the JMCIS/CC object as a Common Service ("server") on the CC ORB, but for simplicity we represent it as an application object. This application/server distinction is often vague in CORBA. Similar to the above description of the JMCIS subsystem treatment, the Combat Control subsystem is effectively "wrapped" by this architecture. Access to Combat Control from the C3I system is through the Combat

```
 This figure is not printing yet, it looks a little like
the previous figure.
It will be in the hard copy that I send.
```

Figure 3: NSSN C3I CORBA Architecture With Combat Control and JMCIS

Control object (shown as two-shaded between the CC ORB and the C3I ORB). Thus, when C3I components need to access Combat Control, they do it through this object, which makes IDL calls to the Combat Control server objects. If any of these server objects are replaced, as long as the IDL interface is preserved, the rest of Combat Control and C3I is not affected.

The lower third of Figure 3 is the C3I system of Figure 2. The diagram of Figure 3 is not complete. Sonar and other subsystems may also have their own CORBA environments and would be integrated similar to the way that we propose above for Combat Control.

We now examine the lower-level details of integrating a real-time database to act aos one of the Common Facilities shown in the CORBA architecture of this section.

4 Integration of a COTS RTDBMS into CORBA

To integrate a COTS RTDBMS into the CORBA environment, a CORBA "wrapper" has to be created. A wrapper is essentially an application that is connected to both the CORBA environment and to the RTDBMS. A client connects to the wrapper via CORBA and sends its request; the wrapper relates the request to the RTDBMS, receives a reply from the RTDBMS, and transmits the results back to the client. Among the open system design requirements are that the interface should be easy for the clients to use and it should be general enough to replace the RTDBMS without affecting the clients. This section focuses on the interface design and the wrapper components.

To illustrate the CORBA wrapper's interface and design we use a sample real-time database application from the NSSN's C3I system. In this simplified application, the NSSN Combat Control subsystem stores information remotely (in the RTDBMS) about the contacts that it is tracking: a contact identifier (trackid), the contact's speed (speed), the contact's bearing (bearing), and the timestamp that indicates when these values were recorded (tstamp). The Zip RTDBMS from DBx will be the RTDBMS used in this illustration. Zip RTDBMS is one of the two commercially available RTDBMS of which we are aware. It is based upon a relational data model and it is designed to be fast, predicatable, stand-alone (non-distributed) and adhere to open system requirements. We use this system for illustration and testing of our design.

4.1 Interfaces

There are two general forms of CORBA IDL interfaces for a COTS RTDBMS: a *Custom Interface* and a *Standard Interface*. A Custom Interface contains individual instructions that are specific to the underlying database, such as Insert, Delete, Select, and any special database setup commands that are necessary for the specific database. A Standard Interface is one that adheres to an established standard, such as SQL, and is independent of the underlying database. To support real-time databases functionality, our design calls for either form of CORBA interface to additionally include 1) Precompiled modules 2) Scheduling methods and attributes 3) Configuration methods and attributes. The *Precompiled modules* constitute preprogrammed actions built into

```
interface Database {
type DeclareRelation(params)
type BindBuffers(params)
type SetAttributeList(params);

type Insert(params);
type Delete(params);
type DelarePredicate(params);
type DeclareLogop(params);
type Select(params);
};
```

(A) CORBA IDL Interface

```
TupleSet TIDset;
DeclareRelation(tracks);
 BindBuffers(trackid);

 DeclarePredicate(trackid_name,
    trackid, trackid_dize, EQ, 2);
 DeclarePredicate(trackid_name, trackid,
    trackid_dize, EQ, 4);
 DeclareLogop(OR);
 TIDset = Select(database);
```

(B) Example Track Query

Figure 4: Custom Interface and Example Query (for Zip RTDBMS)

the server with corresponding methods defined in the CORBA IDL. The *Scheduling methods and attributes* are to facilitate analysis of real-time behavior. The *Configuration methods and attributes* enable/disable some functions of the RTDBMS or otherwise configure its features to achieve the desired real-time execution.

Custom Interface. A Custom Interface is typically more efficient and predictable at run-time. It is more efficient because the wrapper can map the CORBA clients' calls almost directly to calls of the underlying real-time database. The server's response is more predictable because there is no parsing of commands as there is in a Standard Interface (described next). Instead, the wrapper can translate all calls in a fixed amount of time.

However, a Custom Interface violates open system's requirements. In particular, it makes the process of replacing the underlying real-time database difficult. If a replacement occurs, one of two undesirable choices must be made. A new Custom Interface could be constructed that is specific to the new database, which would also require all of the clients to be rewritten. Alternatively, an awkward wrapper mapping the Custom Interface from the former real-time database to the capabilities of the new real-time database could be constructed.

Consider the custom CORBA IDL interface for Zip shown in Figure 4A. Zip RTDBMS currently does not support an SQL interface, instead its interface contains methods such as `DeclareRelation(params)`, `BindBuffers(params)`, `SetAttributeList(params)`, and `DelarePredicate(params)`, which are all Zip-specific. In addition, it contains some more general database operations such as `Insert` and `Delete`. Even these general operations are low-level and require that each client be able to write in these low-level terms or parse down to them.

In the NSSN tracking example, suppose a client wishes to perform a selection of all the tuples with `trackid=2` or `trackid=4`. With the Custom Interface of Figure 4A, the query would be expressed as shown in Figure 4B.

Standard Interface. Unlike a Custom Interface, a Standard Interface adheres to open system requirements. The clients' requests can be written in a standard language, such as SQL, without knowledge of the underlying real-time database. In a large distributed system, with client's from many domains, this generality is important. In addition, if the underlying real-time database were to be replaced, as long as the new real-time database wrapper supported an SQL interface, the clients' code would not have to be changed. Furthermore, most COTS databases support an SQL interface already, which means that the CORBA wrapper can be programmed relatively easily and efficiently.

The Standard Interface we have designed allows a client to be written to use standard SQL to express database operations. At run-time, the SQL statements are passed as a string to the CORBA wrapper. Not only does this design allow implementation with CORBA's parameter types (it only uses a string type), it also relieves the clients from needing to have their own SQL parsers. Instead, a single SQL parser is implemented at the server to translate the string that was forwarded by the clients. The parser can be a part of the RTDBMS or an external module (ActiveParser as described later). The following example shows the CORBA IDL for the Standard Interface:

```
        /* CORBA IDL for Standard Interface */
    interface Database {
              string Command(in string transmission);

        /* Precompiled module declarations */
              string Module01(in withindeadline, ..attributes..);
              string Module02(in bydeadline, ..attributes..);
              ...
        /* Scheduling attributes */
              attribute float ModuleTime01;
              attribute float ModuleTime02;
              ...
/* Configuration attributes and methods */
              attribute unsigned SchedulingHeuristic;
              attribute long CacheBuffer;
              ...
    };
```

A client invokes the `Command()` method whenever it wishes to perform SQL operations on the remote real-time database. The method's only parameter is a string containing an SQL statement. The return `string` value for `Command()` returns results back to the client.

The client's query for the track identifiers that was expressed in the Custom Interface (as shown in Figure 4B), is expressed in the Standard Interface as:

```
string result;          /* declaration of empty string */
string query = "SELECT ALL * FROM tracks WHERE trackid = 2 OR trackid = 4;"
result = ZServer.Command(query);
```

After executing the call to the RTDBMS server (`ZServer`), the string `result` contains the set of tuples, if any, returned by the RTDBMS.

Observe that the client executing the query of Figure 4B on the Custom Interface needs not only to be compiled specifically for that interface, but also has to know the command structure of the underlying real-time database to execute the IDL methods in a specific sequence. Furthermore, the client's code is only usable for that particular database. On the other hand, the client program written for the Standard Interface only makes one call, passing a query in the widely-used SQL language. This client will work for any database that provides an SQL interface (even a local non-real-time database during testing) and the client program needs no information about how to program the particular database it is accessing.

Though the original system was designed to support SQL, it was relatively easy to move to RTSQL. The CORBA Standard Interface remains virtually the same, i.e. to send queries we simply pass RTSQL instead of SQL. A RTSQL interface to the Zip RTDBMS system was developed in [6]. This has made it relatively easy to start incorporating some of the basic features of RTSQL into the CORBA wrapper for the COTS RTDBMS.

4.2 Precompiled Modules

Although a dynamic RTSQL connection to the database is a powerful and desirable feature, hard real-time systems will be more likely to utilize a predefined set of actions that are known at system design. The CORBA interface could provide access to these actions using Precompiled Modules. One feature of these modules is that they have known execution metrics (i.e. maximum execution time) and are therefore predictable. Another feature of Precompiled Modules is that they are easily mapped to SQL Persistent Stored Modules (SQL/PSM). SQL/PSM expands SQL by adding procedural language extensions and stored procedures. SQL/PSM is expected to become a standard in 1996.

There are two ways to place Precompiled Modules on the CORBA interface. The first way is to create a method in the IDL for each module, using some consistent naming convention. However each module may need to have other modules (or methods) and attributes associated with it, such as read-only attribute for its execution time, an attribute for execution status, logging, etc. Thus, for each such module, a set of definitions must be added to the IDL. Clearly if the number of modules is large, the CORBA IDL will become unwieldy.

The second method of adding Precompiled Modules on the CORBA interface is to extend the **Command** method with the ability to accept some new functions that can invoke the modules on the database and return the necessary scheduling and administration information. These functions will have to be parsed by the on-server parser that previously handled only RTSQL statements.

4.3 Scheduling interface

In a real-time environment all RTSQL transactions must be scheduled either at the client side, the server side or both (depending on the implementation). Although scheduling techniques on a distributed system are outside the scope of this paper, it suffices to say that the CORBA wrapper for a real-time database should contain some Scheduling attributes and/or methods. For example, these could be read-only attributes containing predicted execution times for the precompiled modules. The Scheduling part of the IDL may also contain the status information about the state of the on-server scheduler, i.e. the transactions currently being considered, work-load, successful completion ratio, etc. All these can provide the client with the information it needs to make decisions about which real-time database to use, perform its own scheduling, identify database problems, etc.

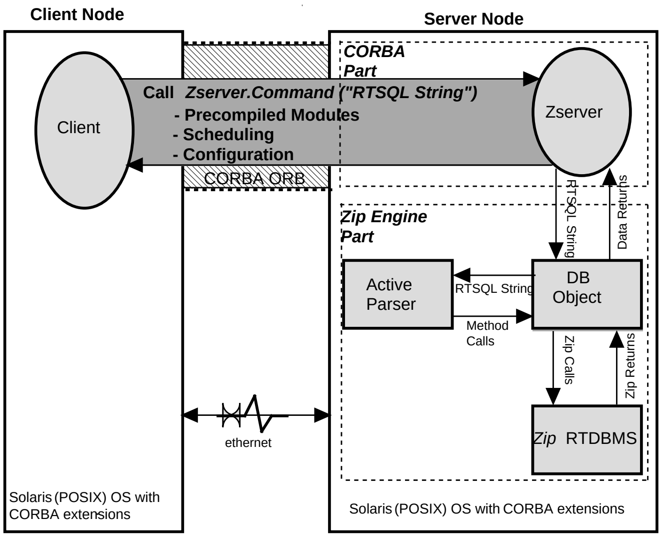

Figure 5: Architecture of CORBA Wrapper For COTS RTDBMS

4.4 Configuration interface

A configuration interface is a natural part of any RTDBMS, but for a real-time distributed system it may be necessary to place some of control methods on the CORBA IDL to enable client to make the decisions about the server configuration. For example, if the RTDBMS is simply a sensor monitor, its accuracy and period of update can be set by the client dynamically depending on the transaction load, strategic situation, hardware integrity, etc. The client may wish to turn some functions of the database off in order to increase predictability when it is most desired. Also, the on-server scheduler may be set to different modes of operation. Of course, the client must have the rights and priority to make these changes to the server, since it may impact other objects in the distributed system.

5 Wrapper Architecture

Figure 5 depicts the architecture of the CORBA wrapper for a COTS RTDBMS. The Client Node is an application running on any real-time POSIX-compliant operating system [7] with CORBA extensions. The Server Node is another application, on a (possibly remote) node that is also running a real-time POSIX-compliant operating system with CORBA extensions. The interaction between the client and the server is managed by the CORBA ORB (see Section 2), which is shown in the figure as the shaded region between the two nodes. As depicted, to execute a request on the DBMS Server, the client makes a CORBA method call. The wrapper provides the method `ZServer.Command()` of the Standard Interface to the RTDBMS for the RTSQL requests (see Section 4) and methods/attributes for scheduling and configuration.

Server Node Implementation. The Server Node consists of two major parts - the *CORBA part* and the *DBMS-Engine part*, outlined with dashed lines in Figure 5. Between the parts, two POSIX pipes are set up for exchange of information. The DBMS-Engine part contains three modules - the *Active Parser*, the *DB Object* and the actual *RTDBMS*. The CORBA part has only one module, *ZServer*, the CORBA implementation of the server object. In general terms, the CORBA part deals with maintaining the connections with clients and promptly transmitting their requests to the DBMS-Engine part. The DB Object and the ZServer are connected by two one-directional POSIX pipes. The pipe from ZServer to DB Object carries database requests (as RTSQL strings). The opposite pipe passes the return data from the RTDBMS. When something appears on the incoming pipe, the DB object retrieves it and sends it to the Active Parser. The parser operates on the string and executes direct method calls on the RTDBMS during the translation. Note that the parser is needed only for the databases that do not have a RTSQL interface or those that implement precompiled modules and scheduling directives that must be parsed. Each module of the wrapper is discussed in greater detail below.

ZServer. The CORBA part of the wrapper contains one module, the ZServer, that does everything necessary to set up the application as a CORBA server. It connects to the ORB through CORBA daemons (in our case ORBeline's CORBA "osagent"). The ZServer process then enters a loop, monitoring the CORBA connection for client requests and incoming SQL string data. When a client binds and sends a request, ZServer decodes the message and puts the information in form of a text string through the outgoing pipe to the DBMS-Engine part. The CORBA part of the wrapper is designed to be independent of the underlying database. It simply sends the SQL text strings to the pipe and expects a string of data in return.

DB Object. The DB Object sets up the connection with the actual DBMS and maintains it throughout the life of the wrapper application. The DB Object serves a dual purpose in the DBMS Engine part design. One of its jobs is to poll the pipe for incoming strings from the CORBA part. When they are received, it calls the Active Parser, passing the string to it. When the parser returns, the DB part packages information into a string format and sends it back to the ZServer via the return pipe.

The second job of the DB Object is to serve as a layer between the Active Parser and the actual RTDBMS. The DB Object contains methods for accessing the RTDBMS directly. The DB Object also configures all of RTDBMS's configuration variables, data-exchange buffers, etc. By hiding these RTDBMS-specific interactions, the DB Object isolates the Active Parser from dependence on the underlying database and makes DBMS appear as an object to other parts of the wrapper.

Even if the RTDBMS has a RTSQL interface, the DB Object will still be needed to provide methods for configuration and scheduling to implement requests from the client.

Zip. The Zip RTDBMS does not have a standard SQL interface and is accessible to users only through a set of function calls. The Active Parser and the DB Object essentially translate RTSQL into a series of Zip function calls. The functions that Zip provides account for inserting, deleting and selecting. Selecting in Zip is done through direct declaration of predicates in Postfix notation, one at a time, followed by a call to the `selectDB` function. The result is a linked list of tuples, if any, that can be traversed one at time for individual content.

Active Parser. In the initial implementation, the Active Parser was designed to receive an SQL string from the DB Object, parse it down to basic database instructions, and call the DB Object to execute RTDBMS functions as it parses. The Active Parser was written using the YACC parser generator program. YACC accepts a context-free grammar representation of (in this case) the SQL grammar along with annotations of C functions to be called if a grammar rule is matched. We wrote one for Zip RTDBMS and hence will demonstrate it using Zip's grammar and calls. For instance consider the YACC input fragment from the Active Parser:

```
where_clause:
                WHERE search_condition
            ;
search_condition:
            |    search_condition AND search_condition { DeclareLogopAND(); }
                 ...
            |    comparison_predicate
            ;
comparison_predicate:
            |       TRACKID COMPARISON INTNUM  {  SetTRACKID($3);
                      switch($2) {
                        case 0: DeclarePredicateTRACKID(EQUAL); break;
                        case 1: DecalrePredicateTRACKID(LESSTHAN); break;
                         ....
                      }
                    }
```

This YACC input specifies the grammar of an SQL `where_clause`, which is part of the `SELECT FROM WHERE` statement in SQL. The rules state that if the *where* clause contains an "AND", then Zip's `DeclareLogopAND();` operation must be called to specify the AND logical operator. Thus, if the Active Parser received the SQL string from our tracking example:

```
SELECT FROM tracks ALL * WHERE trackid = 2 AND speed = 200.
```

it would parse the *where* clause by first translating the two search conditions (`trackid = 2`, `speed = 200`) and then executing a Zip logical AND (`DeclareLogopAND();`).

Note that the Active Parser is specific to the relation *tracks*. It contains tokens and constructs such as `TRACKID` and `SPEED` which are part of the grammar. Furthermore, the DB Object methods also effectively correspond to the *tracks* column entries, including `DeclarePredicateTRACKID`, etc. Due to their database relations dependence, parts of the Active Parser and the DB Object have to be generated for each set of relations. This implies that there is an Active Parser and a DB Object for each database schema. In our generation of the Active Parser and the DB Object we used the Zip Data Definition Language file. The file defines all the relations for the database and is sufficient to automatically generate the Active Parser and the DB Object. The resulting Active Parser and DB Object then have to be compiled and linked into the server application.

Real-Time SQL. Currently, we are extending the Active Parser to support RTSQL. In particular, timing constraints on data manipulation statements have been added. For example:

```
SELECT ALL * FROM tracks WHERE trackid=2 AND SPEED=200
  COMPLETE BEFORE CURRENT_TIMESTAMP + INTERVAL '5' SECOND;
```

contains a deadline specification equivalent to "NOW + 5 seconds". This implies that the RTDBMS must complete the query within five seconds of its arrival at the RTDBMS. Note that this timing constraint is specific to the RTDBMS, not any of the layers above it. The client must be provided with other mechanisms for time constraining the entire process.

Since timing constraints on actions are not provided by the Zip RTDBMS, the Active Parser will cause DB Object to set a POSIX operating system timer to expire at an absolute time (such as NOW +5 seconds). Since timing constraints appear at the end of the data manipulation statement, the parsing is slightly different for the statements with the time constraints and without, by calling additional methods on the DB Object. If it does, then values such as `CURRENT_TIMESTAMP` will be evaluated and the timer set immediately before any calls to the Zip RTDBMS are executed.

A more complete RTSQL interface to Zip RTDBMS was developed in [6]. The interface was implemented by creating an object wrapper to the Zip RTDBMS (similar to DB object), and an RTSQL parser which produced calls to a specific database object. The RTSQL parser included support for a variety of timing constraint clauses on statements including **START AFTER**, **START BEFORE**, and **COMPLETE BEFORE**. It also allowed for these constraints to appear in a nested statement environment. Data timing constraints were supported by the addition of timestamp values on data, along with extensions to the SQL constraint specification to allow access to this timestamp and time valued functions. An important aspect of the work was a condition handling mechanism that allowed the user to specify alternative actions when timing constraints on statements or data were violated. At this time, the RTSQL Active Parser utilizes default handlers provided in the DB object.

Implementation for Performance Test with Zip RTDBMS. We implemented a prototype after the design using Zip RTDBMS as the database for the DBMS-Engine part and were able to test it and collect performance statistics. We tested the prototype using various examples from the tracking application. In a *parse through* test, an SQL string comes in from the client and goes through the process of parsing and executing on Zip. An *embedded statement* test consisted of a sequence of pre-made (embedded in DB Object code) queries that operate on Zip directly without going through the Active Parser module. We measured performance as the amount of time needed to execute one complete statement, i.e. from the time of `Command` method call on the client side to the time of the result arrival back to the client (with precision of +/- 500 ns). For example, a client call of

```
INSERT INTO tracks (trackid, speed, bearing, tstamp) VALUES (v1, v2, v3, v4);
```

required 136ms on a parse through and 123ms as an embedded statement.. A client call of

```
SELECT ALL * FROM tracks WHERE [clause_list];
```

required 138ms on a parse through and 128ms as an embedded statement. Our tests indicate that the active parsing of the wrapper adds approximately 10% overhead.

6 Conclusion

This paper has discussed the US Navy's NSSN, an application with distributed, real-time, and open system requirements. We presented a high-level CORBA architecture to support these requirements. We also presented a lower level implementation of a key aspect to that architecture: the integration of a COTS real-time database into a CORBA environment. The real-time aspects included using an underlying COTS real-time database with predictable execution times for basic operations, a Real-Time SQL CORBA IDL interface that allows time-constrained SQL statements, CORBA interface additions for pre-compiled modules with known execution times, CORBA interface scheduling parameters to provide information for real-time scheduling decisions, and CORBA interface configuration parameters to allow clients to configure the real-time database for suitable real-time execution.

Our design and prototyping effort identify several areas where further research and development is required to achieve suitable real-time behavior in such open systems. First, the real-time databases that are commercially available are still immature. The Zip COTS RTDBMS that we used has a non-standard interface, limited database capabilities, and no real-time scheduling of low-level database operations. Basically, it provides predictable execution times for data storage and retrieval, but is far from possessing full database functionality. Second, the semantics of RTSQL are currently defined assuming a single processor system. In a distributed client/server system it is not clear what the real-time semantics of RTSQL are. For instance, is the deadline specified by the client a client-side or server-side deadline? In our implementation we interpreted it as server-side, semantics such as this need to be defined for RTSQL's use in a distributed environment.

Finally, the real-time semantics of CORBA are also undefined and this is a major weakness of the ability our prototype to provide distributed real-time behavior. As we mentioned, a SIG of the OMG is approaching defining real-time CORBA, but the NSSN-based application that we presented here indicated the relative complexity of the problem. Assuming that clients want at least time-constrained service (soft real-time) and maybe even predictable service (hard real-time), there are many parts to our system that need modification: the scheduling on the client, the protocols on the network, the scheduling of services in CORBA, the scheduling on the server - to mention a few. The problem of end-to-end scheduling in a distributed system is a complicated one, but one that must be solved and supported in an open standard like CORBA if a distributed open system is to achieve full real-time performance. Although, much more work has to be done by researchers, commercial vendors, and standards developers in applications that integrate existing research and standards, this paper presents design ideas and prototyping experiences toward creating distributed, real-time, open systems.

References

[1] J. Melton, ed., *ANSI X3.135-1992, American national Standard, Database Language SQL.* American National Standards Institute, 1992.

[2] P. Fortier, J. Prichard, and V. F. Wolfe, "SQL/RT: Real-time database extensions to the SQL standard." To appear in *Standards and Interface Journal*, 1994.

[3] O. M. Group, *The Common Request Broker Architecture.* X/Open, 1992.

[4] P. Krupp, A. Schafer, B. Thuraisingham, and V. F. Wolfe, "On real-time extensions to the common object request broker architecure," in *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with the Common Object Request Broker Architecture (CORBA)*, Sept. 1994.

[5] V. F. Wolfe, J. Black, B. Thuraisingham, and P. Krupp, "Towards distributed real-time method invocations," in *Proceedings of the International High Performance Computing Conference*, Dec. 1995.

[6] J. Prichard, "Real-time extensions to the SQL query language." PhD Dissertation, Department of Computer Science, the University of Rhode Island, 1995.

[7] IEEE, *Portable Operating System Interface (POSIX); Part 1: System API; Ammendment 1: Real-time Extension.* IEEE, 1994.