INTEGRATION FOR REAL-TIME CORBA INTO AN EXISTING DISTRIBUTED PLANNING APPLICATION

BY
YURUO CHEN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1998

MASTER OF SCIENCE THESIS

OF

Yuruo Chen

APPROVED:

Thesis Committee

Major Professor ___________________________

____

____

____

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

1998

Abstract

This thesis presents the migration of a large military planning application to support the Object Management Group's IIOP protocol for middleware interoperability. It includes incorporation of the real-time CORBA system developed by the real-time lab at University of Rhode Island (URI) to the application.

This thesis also tested the military planning application running on top of Real-time operating system. The test indicated that the model developed by real-time lab at URI is easy to be incorporate into non-real-time system and it also correctly demonstrated the real-time features of the URI's RTCORBA system.

ACKNOWLEDGMENT

I would like to thank my major professor Victor Fay-Wolfe. His inspiration and wisdom are invaluable help to my work. I feel fortunate to have such a adviser, a truly pioneer in real-time distributed system research.

I would also like to thank gurus of real-time, Mike Squadrito and Dr. Lisa Cingiser Dipippo. I think I speak for Qingli, Levon and Greg when I say it would have been much longer, less enjoyable road to incorporate the real-time to JFLEX without their help.

I want to also thank all my friends who believe in and support me during my study here.

Finally, most of all I would like to thank my parents for their support not only during their stay here, but also for all my life. They are always been there for me, encouraging me and helping me. I dedicate this thesis to them.

Contents

1. Introduction 1

1.1 Problem Statement……………………………………………………1

1.2 Motivation …………………………………………………………2

1.3 Objective ……………………………………………………………...3

1.4 Thesis Outline ………………………………………………………3

2. Related Work and Background 5

2.1 Common Object Request Broker Architecture (CORBA) ……………...5

2.1.1 CORBA Standard and its Implementation ……………………5

2.1.2 Internet Inter-ORB Protocol (IIOP) …………………………...8

2.2 Real-time CORBA Model ……………………………………………...10

2.2.1 Requirement of Real-time System ……………………………11

2.2.2 Our Real-time CORBA Architecture …………………………12

2.3 Related Work …………………………………………………………15

3. Model Description of Real-time JFLEX 17

3.1 Structure of JFLEX …………………………………………………..17

3.2 Structure of Real-time JFLEX ………………………………………..19

4. Implementation **23**

4.1 Migration of JFLEX …………………………………………………23

4.2 Real-time JFLEX Implementation ……………………………………..24

4.2.1 Server Side Implementation …………………………………..24

4.2.2 Client Side Implementation …………………………………..26

5. Evaluation **31**

5.1 Testbed Construction ……………………………………………………31

5.2 Testing for IIOP for New JFLEX ………………………………………..32

5.3 Testing the Real-time JFLEX ……………………………………………...32

6. Conclusion **38**

6.1 Contribution ……………………………………………………………..38

6.2 Limitation and Future Work ……………………………………………38

7. List of Reference **40**

8. Bibliography ***42***

List of Figures

1. Object Request Broker Architecture ……………………………………………6

2. Real-time CORBA ……………………………………………………………13

3. JFLEX Structure ………………………………………………………………18

Chapter 1

Introduction

This thesis presents the migration of one ORB implementation to another for JFLEX planning software. It also modifies its implementation to support requirements of a real-time distributed system.

1.1 Problem Statement

Joint Force Level Execution Aid (JFLEX) is a software application developed by Navy Research Lab at San Diego. It provides a user-friendly method for monitoring the progress of a plan's execution. JFLEX depicts a plan being composed of responsible organizations and subplans (sequences of actions). Each responsible organization is assigned a subplan for execution. JFLEX distributes the status information for a plan to each workstation running an authorized JFLEX client over a wide area TCP/IP network. Each JFLEX station can change and update the status of plan activities. This allows managers to rapidly get a feeling of how well a plan is succeeding. Obviously JFLEX is a distributed application.

Distributed object computing is becoming a widely accepted programming paradigm for applications that require seamless interoperability among heterogeneous clients and servers. The Object Management Group (OMG), an organization of over 600

distributed software vendors and users, has developed the Common Object Request Broker Architecture (CORBA) as a standard software specification for such distributed environments. The CORBA specification includes an Object Request Broker (ORB), which is middleware that enables the seamless interaction between distributed client objects and server objects; Object Services, which facilitate standard client/server interaction with capabilities such as naming, event-based synchronization; and the Interface Definition Language (IDL), that defines the object interfaces within the CORBA environment.

There are many ORB implementations in the industrial world, such as NEO from SUN, ORB PLUS from IBM, Object Broker from DEC, ORBIX from IONA, CORBUS from BBN. In order to support distributed object computing independent of implementations of the CORBA standard, the OMG specified a protocol, via which a client of one ORB can invoke operations on an object in a different ORB. This protocol is called General Inter-ORB Protocol (GIOP). The OMG defines a specialization of GIOP that uses TCP/IP as transport layer. This specialization is called Internet Inter-ORB Protocol (IIOP).

1.2 Motivation

JFLEX is a distributed application that was originally implemented on BBN's CORBUS system. It originally did not allow other CORBA systems to run JFLEX, either as server or client. Since JFLEX stations will be distributed geographically and belong to different organizations, it is very likely that they have different CORBA compliant ORB

implementation on them. So we should make JFLEX support IIOP because of the popular TCP/IP in current network.

JFLEX is a military planning application with real-time characteristics: The situation, process, and condition of a plan or subplan could change at anytime. Any such kind of change may influence the overall success of a plan. The plan must be updated and monitored in real time in order to show the plan's very current status correctly. The original JFLEX project did not have a way to express and enforce time constraints.

1.3 Objective

The main goal of this research is to make JFLEX support IIOP and real-time features. The thesis goal involves the migration of an application from the CORBUS ORB to IONA's ORBIX ORB. Also it involves implementing URI's model of *Timed Distributed Method Invocations (TDMI)*. It will show how well the real-time CORBA developed at the real-time research lab at URI fits for a large real-time plan applications.

1.4 Thesis Outline

Chapter 2 introduces the JFLEX project and contains a review of the CORBA standard, the Internet inter-ORB protocol. It also provides background on the Real-time CORBA system developed by the real-time lab at University of Rhode Island. Chapter 3 describes the model of Real-time JFLEX. Chapter 4 describes the migration of JFLEX project, as

well as providing examples demonstrating how to incorporate the real-time control into the JFLEX software. Chapter 6 presents and analysis of the result of the performance tests using simulated workload, comparing the result with or without the real-time support. Chapter 7 explains the contributions and limitations of this thesis, and discusses future work.

Chapter 2

Related work and Background

This chapter provides background information on the CORBA standard, its components and its implementation. We also describe how the Internet Inter-ORB protocol is supported. The Real-time CORBA model developed at University of Rhode Island is introduced at the end.

2.1 Common Object Request Broker Architecture (CORBA)

This section first presents the overall structure of the CORBA standard and gives a simple introduction to each of its main components. It then provides the background on the Internet Inter-ORB protocol.

2.1.1 CORBA Standard and its Implementation

The basic notion behind CORBA is to provide a uniform way for any object to receive and respond to a request from any requester (client), either another object or even a traditional nonobject-oriented program. Once such a request is made, the ORB makes sure that the request is delivered to an appropriate receiving object, no matter where it is and how it is implemented. To provide all these capabilities, the CORBA specification [1] defines the architecture of interfaces that may be implemented in different ways by different vendors. The architecture was specifically designed to separate the concerns of

interfaces and implementations (Figure 1). The main components of the architecture may be divided into three specific groups: client side, implementation (server) side, and the ORB core. The client and server sides represent interfaces to the ORB via the IDL.

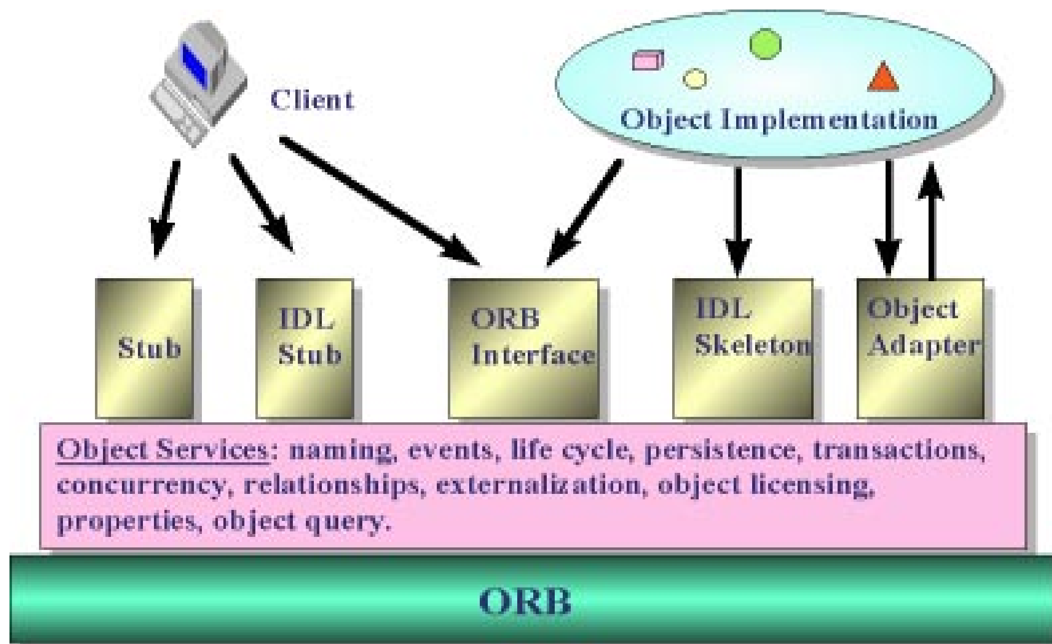

Figure1: Object Request Broker Technology

As mentioned previously, the purpose of the IDL is to allow the language independent expression of interfaces, including the complete signatures (name, parameters and their types, etc.) of methods or functions, and accessible attributes. An interesting aspect of the IDL is the exception. Exception declarations define a struct-like data structure with attributes that can be used to pass information about an exception condition to a server requester. An exception is declared with its name, which is accessible as a value when the exception is raised, allowing the client to determine which exception has been received.

The client-side architecture consists of three components:

- The Dynamic Invocation interface (Stub) - mechanism for specifying request at runtime;
- The IDL stub interface – small piece of machine-language code, which is generated according to IDL interface definitions.
- The ORB service interface – a number of functions that may be accessed directly by client code (e.g. retrieving a reference to an object)

One aspect of the client-side interface is shared by object implementations: the ORB services. The other two components on the implementation side are:

- The IDL skeleton interface – is an up-call interface through which the ORB calls the method skeleton of the implementation, on a request by a client;
- The Object Adapter – is the means by which server implementations access most of the services provided by the ORB (e.g. generation and interpretation of object reference).

The CORBA specification is not explicit about what services an adapter must support, but it is clear that the adapter is intended to isolate object implementations from the ORB core as much as possible.

Objects in CORBA are created and destroyed dynamically in response to the issuance of requests. Objects can also participate in any of the normal types of relationships, with perhaps the most important being subtype/supertype relationships or inheritance. Inheritance between object interfaces is specified syntactically by using the IDL. The object model in CORBA is strongly typed. As in C++, types are used to restrict and characterize operations. Unlike language such as Smalltalk, these types are not first-order objects, and cannot be manipulated as objects.

The major part of the CORBA standard is Object Services – some software designed to provide a particular set of operations applicable to broad classes of objects. For example, a given object service might store and retrieve objects, or it might manage relationships among objects, etc. the Object Services are the key to expanding the functionality and interoperability of objects beyond the simple request management capabilities of the ORB.

However, for the OMG's CORBA to be truly usable as an industry standard for wide range of commercial applications, much more work needs to be done, and the OMG formed different SIGs for that purpose. There are a number of implementations of CORBA available now on the market, one of which is ORBIX from IONA Technologies.

ORBIX represents the distillation of ten years in the area of distributed system. Because the software was built from scratch, it conforms faithfully to the OMG's CORBA specification. It does not contain vestiges of an old product trying to comply with the CORBA standard, and its architecture presents CORBA in a most natural way to C++ developers. The product has been tried, tested and deployed by corporations across the globe. It is used by leading software providers for the banking, telecommunications, engineering and government sectors. With ORBIX, programmers can develop distributed applications using object-oriented client-server technology, and use object technology to compose new applications from existing components and subsystems.

2.1.2 Internet Inter-ORB Protocol (IIOP)

ORB interoperability specifies a comprehensive, flexible approach to supporting networks of objects that are distributed across, and managed by, multiple, heterogeneous CORBA-compliant ORBs. The approach to "interORBability" is universal, because its elements can be combined in many ways to satisfy a very broad range of needs. The elements of interoperability architecture are as follows: ORB interoperability architecture, Inter-ORB bridge support and General and Internet inter-ORB protocols (GIOPs and IIOPs).

In ORB interoperability architecture a *domain* is a distinct scope, within which certain common characteristic are exhibited and common rules are observed: over which a distribution transparency is preserved. This abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or bridge mechanism resides between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The GIOP and IIOP support protocol-level ORB interoperability in a general, low-cost manner. GIOP specification consists of the following elements [2]:

- The Common Data Representation (CDR) definition. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- GIOP Message Formats. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.

- GIOP Transport Assumptions. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.

The IIOP specification adds the following element to the GIOP specification:

- Internet IOP Message Transport. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.

For IIOP, agents that are capable of accepting object requests or providing locations for objects (i.e., servers) publish TCP/IP addresses in IORs (Interoperable Object Reference). A TCP/IP address consists of an IP host address, typically represented by a host name, and a TCP port number. Servers must listen for connection requests. A client needing a service from server must initiate a connection with the address specified in the IOR, with a connect request. The listening server may accept or reject the connection. In general, servers should accept connection requests if possible, but ORBs are free to establish any desired policy for connection acceptance (e.g., to enforce fairness or optimize resource usage).

Once a connection is accepted, the client may send **Request**, **LocateRequest**, or **CacelRequest** messages by writing to the TCP/IP socket it owns for the connection. The server may send **Reply**, **LocateReply** and **CloseConnection** messages by writing to its TCP/IP connection. After sending/receiving a **CloseConnection** message, both client and server must close the TCP/IP connection.

2.2 Real-time CORBA Model

2.2.1 Requirements of Real-time systems

In a real-time system, timing constraints must be met for the application to be correct. This requirement typically comes from the system interacting with the physical environment. The environment produces stimuli, which must be accepted by the real-time system within timing constraints. The environment further requires control output, which must be produced within timing constraints.

One of the main misconceptions about real-time computing is that it is equivalent to fast computing. Sometimes researchers challenge this myth by arguing that computing speed is often measured in average case performance, whereas to guarantee timing behavior, in many real-time systems worst case performance should be used. That is, in a delicate application, such as nuclear reactor or avionics control, where timing constraints must be met, worst case performance must be used when designing and analyzing the system. Thus, although speed is often a necessary component of a real-time system, it is often not sufficient. Instead, predictably meeting timing constraints is sufficient in real-time system design.

Real-time systems require that timing constraints be expressed, enforced and their violations handled. The unit of time-constrained execution is called a task. For example, in a real-time database, time-constrained transactions are considered tasks. Timing constraint expression can take the form of start times, deadlines, and periods for tasks. Timing constraint enforcement requires predictable bounds on task behavior. The handling of timing constraint violations depends on the task requirements: whether they

are hard, firm or soft real-time. A task with a hard real-time constraint has disastrous consequences if its constraint is violated. Many constraints in life-critical systems, such as nuclear reactor control and military vehicle control, are hard real-time constraints. A task with a firm real-time constraint has no value to the system if its constraint is violated. Many financial applications have firm constraints with no value if a deadline is missed. A task with soft real-time constraint has decreasing, but usually non-negative, value to the system if its constraint is violated. For most applications, most tasks have soft real-time constraints. Graphic display updates are one of many examples of tasks with soft real-time constraints.

Most real-time systems specify a subset of following constraints:

- An earliest start time constraint specifies an absolute time before which the task may not start. That is, the task must wait for the specified time before it may start.
- A latest start time constraint specifies an absolute time before which the task must start. That is, if the task has not started by the specified time, an error has occurred. Latest start times are useful to detect potential violations of planned schedules or eventual deadline violations before they actually occur.
- A deadline specifies an absolute time before which the task must complete. Frequently, timing constraints will appear as periodic execution constraints. A periodic constraint specifies earliest start times and deadlines at regular time intervals for repeated instance of a task.

2.2.2 Our Real-time CORBA Architecture

The real-time CORBA model developed by URI's real-time research group uses priority-driven scheduling [3] for the real-time tasks and enforces the soft real-time constraints. I will present theory background for the enforcement of real-time constraint in the following paragraphs.

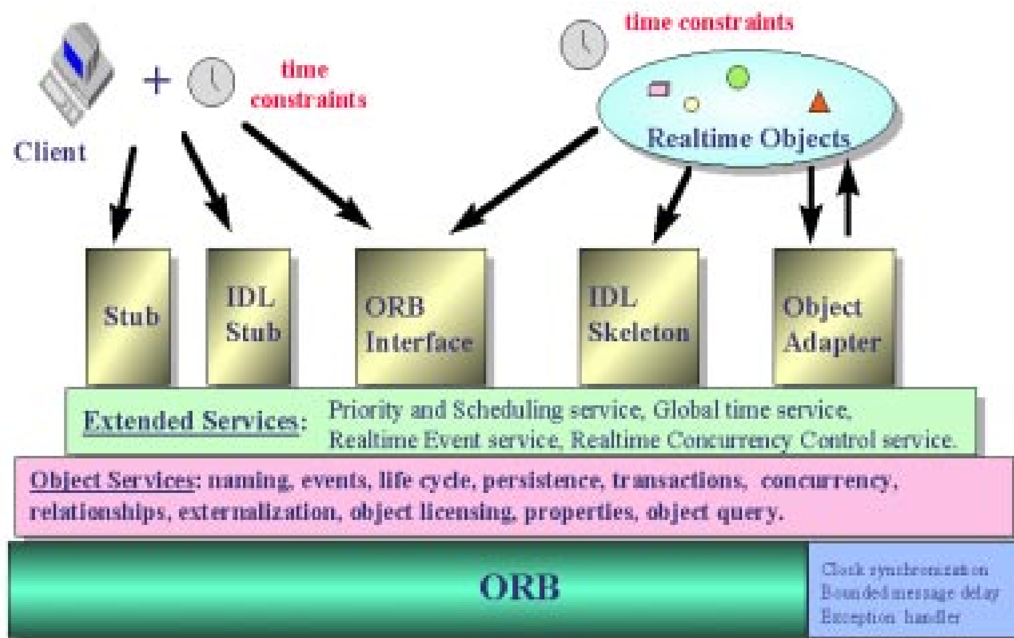

Fig2. Real-time CORBA

We use real-time constraints to calculate the priority of tasks in the CORBA system. Priority calculation is closely dependent on the scheduling policy chosen at the node. For example, in this implementation of a RTCORBA system we chose Earliest Deadline First (EDF) for the scheduling policy, so the activities with the shorter deadlines are given higher Transient Priority values.

One way to provide distributed real-time scheduling is through the enforcement of Global Priority. Global Priority can be represented as an ordinal quantity that is attached to every method invocation and is interpreted in a homogeneous fashion by the

scheduling and queuing for the devices, servers and services in the CORBA system. That is, if method invocation A has a higher Global Priority value than method invocation B, method invocation A should always be treated first.

Enforcement of Global Priority requires the use of real-time schedulers and priority-based queues throughout the distributed system. A real-time scheduler typically strives to execute the highest priority task first and a priority queue typically places the highest task at its head. If these conditions are violated anywhere in the path of a real-time method invocation, unbounded priority inversion [4] may occur and no guarantees can be made about the real-time behavior of any of the components involved.

In a distributed environment, there are two possible ways to manage scheduling: centralized or distributed. In the centralized paradigm, a single entity on the distributed system decides what is executed, where it is executed, and when it is executed. This approach can produce a vast amount of wasted CPU time on the nodes due to network lag of execution instructions. In the distributed model each node does its own scheduling. We have implemented a distributed model in which each node's operating system makes the best decision possible at every moment, based on its currently running tasks and their designated Transient Priorities-without any regard for the scheduling on any other node in the system.

In order to provide for full expression of timing constraints on method invocations, our group has designed a model of Timed Distributed Invocations (TDMIs) in which all timing information is packaged in a structure called a *real-time Environment*. In our model, a CORBA client is started at a base priority that is established from static timing and importance information available when the client is dispatched. The client runs at

this base priority whenever it is not executing a TDMI. To configure a TDMI, the client specifies its timing constraint parameters, its QoS parameter(s), and its scheduling parameters (importance) in the real-time Environment. This environment is attached to every method call and is used by the Global Priority Service, the ORB and the real-time Concurrency Control Service to enforce the specified timing constraints [5].

Our real-time system uses the soft real-time constraints in a CORBA compliant distributed computing environment. The real-time model developed by the Real-time Lab in University of Rhode Island lets the user specify constraints such as importance and deadline for some real-time activities. Then it computes the priority according to these constraints and assigns the priority to the activities. The system may spawn a thread for each real-time activity, where the threads may have different priority. The Solaris operating system can schedule the threads according to their own priority.

After the method invocation acquires a Transient Priority, it must set its thread to an appropriate priority in the server's local operating system, corresponding to its Transient Priority. This requires a mapping of the Transient Priority value into the range of the real-time priorities of the server's local operating system. The function that performs the mapping must be written for each operating system individually because of the variability in ranges of real-time priorities present on different system individually because of the variability in ranges of real-time priorities present on different systems (e.g. Solaris has 60 local priorities and LynxOS has 256). It is clear that mapping of a large range of Transient Priority values into a smaller range of operating system priorities can cause more than one Transient Priority to be assigned to a single local priority value, which could cause priority inversion during execution.

2.3 Related Work

The original JFLEX server was designed for interacting with MSQL (mini SQL) database server. Qingli Jiang in our group has added a CORBA server between the original JFLEX server and MSQL database server. This design allows JFLEX to reach tables in the database without knowledge of what kind of DBMS it is using. For example, Qingli will substitute part of MSQL database with RT Open OODB, a real-time Object Oriented Database. This design will extend the life cycle of JFLEX easily and we can easily incorporate the real-time enforcement into the CORBA server to extend the real-time feature to the database.

Chapter 3

Model Description of Real-time JFLEX

This chapter presents the model for real-time JFLEX. In order to develop an efficient model to incorporate real-time into the JFLEX, which makes the maximum use of the existing code, we first analyze the semantic content, usage patterns and the structure of the system. Then, we evaluate the old structure with new requirements. Finally we describe the model of real-time JFLEX. The first section of this chapter briefly introduces the structure of the original JFLEX. The second section outlines how to incorporate the real-time system into the JFLEX project and the design decisions made for the real-time JFLEX system.

3.1 Structure of JFLEX

JFLEX has three main operation modules: client, outlineServer and MSQL server. The client module allows a user to create a plan representation consisting of plan objectives, subplans, responsible organizations, conditions, and states. The outlineServer continuously receives plan-related information, processes that data, and relays the updated picture to all the JFLEX stations on the network. MSQL server is used to store all the plan related information and facilitate the extraction and update.

The original JFLEX server has two IDL files, outlineServer.idl and jflexOutline.idl. A client will make request according to these two IDL files. The client sends a request to

outlineServer through ORB layer. The outlineServer processes the request sent by client and produces corresponding SQL statements, which in turn are passed to MSQL (mini SQL) server through a C++ wrapper. MSQL server will do the corresponding operation to the JFLEX database according to the SQL statement and pass the result to outlineServer through the C++ wrapper. The outlineServer will then pass the result back to client through the ORB.

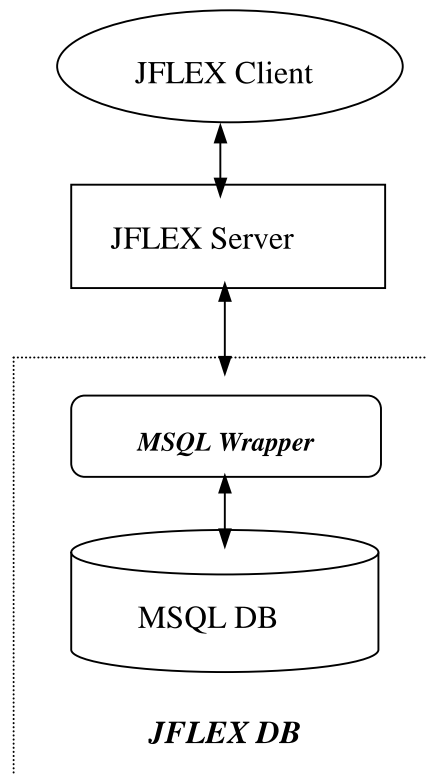

Fig3. JFLEX structure

The internal structure of JFLEX server is a little more complicated. The JFLEX database consists of many tables. For example it has State table, Plan Action table, Plan Table etc. Each table is a C++ object. Methods of the table objects call the functions in MSQL wrapper to operate on the JFLEX database. Those methods for each table object

may not limit their operation on that table, they may also operate on related tables. For example, methods in Condition Table object will operate on Condition Table and State Table. The constraint field in Condition Table is a State, which means that operations on Condition Table will lead to operations on State Table. That is, the methods keep the semantic constraint for the database, which will keep the data consistent among all tables during update etc. Basically, each component of a plan such as Plan action, Plan State, is a C++ object. These are the objects which client calls manipulate. When user wants to modify any components of a plan, he will need to call corresponding method for that object, which in turn calls the corresponding table object's method.

3.2 Structure of Real-time JFLEX

The real-time CORBA model developed at URI isolates the implementation of expression and enforcement of the real-time constraints from the implementation of the real-time work contents. The Pserver and RT_Manager act as real-time agents. To incorporate this model into JFLEX, we only need to modify the original system to interact with those real-time agents.

In order to use the Real-time CORBA model developed at URI, we will have a Pserver real-time daemon running at each node to schedule work at that node according its own load. Since the real-time daemon uses its local clock to check if the deadline is reached, A Global Time Service is required. This service ensures that all clocks in the distributed system are synchronized to within a known skew of each other to provide

consistent notion of time. Clients and servers must be able to call this service to get the current global time.

The approach we used for clock synchronization is the Network Time Protocol (NTP) [6]. The NTP specified in the Request For Comments (RFC)-1305 [7] can be used to synchronize computer clocks in the global Internet. It provides comprehensive mechanism to access national time and frequency dissemination services, organize the time-synchronization subnet, and adjust the local clock in each participating subnet peer. In most places of today's Internet, NTP provides accuracy of 1-50 ms, depending on the characteristics of the synchronization source and network paths. [8]

As mentioned in the last chapter, when we use the TDMI model for the real-time distributed system, we will spawn a thread for each method invocation. A RT_Manager object is created at both client and server side to pass the real-time information to the real-time daemon. It also acts as broker between the new created thread and real-time daemon. [9] The modifications to JFLEX client involve incorporation of the TDMI model from scratch. However, for the server side, we know that the request goes through a server object to table objects. We had to decide in which object we wanted the real-time constraint enforced. A single request usually affects only one server object, but might affect several table objects. Because the real-time constraint is for a single request, we want each request to spawn only one thread so we can enforce the real-time constraint just once. That is the reason we want to incorporate the TDMI model into the server object implementation. We will discuss the model in more detail at client side and server side separately.

At the client side, each time a real-time request is made, the client specifies the deadline and importance to object, RT_Manager_Client. The RT_Manager_Client object passes this information to the real-time daemon as real-time constraints. It also spawns a new thread, suspends the main thread, and also set the alarm clock for the new thread. The new thread makes corresponding request to the server with the real-time information stored in RT_Environment. If the request does not miss the deadline, the new thread will finish its work and join with the main thread. But if the request missed the deadline, the alarm sends a signal to the new thread. The new thread will detach from the main thread and the main thread will tell the user the deadline is missed now.

At the server side, each time a request is received, server retrieves the real-time information from the RT_Environment. Then the RT_Manager at server side sets the alarm according the real-time information passed from the client. It also has to recalculate the new priority according to the real-time information. If the deadline is missed, alarm will send a main thread signal, then main thread will throw real-time exception.

For the requests, that miss their deadline, we also have to decide how to handle the real-time exception both at client side and server side. At client side, it is the user who decides what to do when a request misses its deadline. A client can automatically send a update request or specify a new deadline etc. In our design, we will just report the miss of the deadline, and give the user the flexibility to decide how to recover from the real-time exception.

At server side, since real-time exception might be raised at any specific moment, the method that consists of series of function calls might finish only some of the function

calls. If interrupted in this series of database operations, corrupt data might be stored in the database. For instance, if interrupted in middle of plan object update, such as link list of Plan State object, it will leave plan objects inconsistent with the database.

It is decision of the system designer whether to keep the corrupt data in the database. Since in JFLEX project the database transaction is allowed to commit before all its changes are written to the database, the logical recovery algorithm will be UNDO/REDO. To undo the effect of certain write_item operations that have been applied to the database, a log for operations need to be kept to keep track of the history of database operations.

We choose to make the object methods call atomic. That means even when the timer sends a signal to the main thread to indicate miss of a deadline, the sever will continue the update and raise the real-time exception after it finishes this atomic operation. The advantage of making this operation atomic is that we do not have to record the log of the database operations. In other words, we do not have to keep track which part of database operations is completed before the real-time exception is raised, which minimizes the original code modification. Since we know during which method the real-time exception is raised, it will be easy for us undo all the work done by the method by reverse every operation in the method. For example, if the operation is insertion, we will do deletion, if the operation is deletion, we can do insertion. The more complicated situation is update. In order to undo update we have to record what the original state of the database.

Chapter 4

Implementation

This chapter presents the implementation of the real-time JFLEX. The first section describes the migration JFLEX to support IIOP. The second section presents how real-time CORBA system was incorporated into the JFLEX project.

4.1 Migration of JFLEX

The original JFLEX project used CORBUS as ORB implementation. Our goal was to migrate it to use ORBIX as the ORB implementation. The purpose of migration is to reuse the code as much as possible and retain all the original functionality of the JFLEX. In our case, we needed to do some modification to make it support IIOP. In the system design phase, we first analyze the semantic content, usage patterns and the structure of the system.

At server side, we kept the IDL files almost unchanged. However, JFLEX uses ORBIX's TIE method to link the IDL interface with C++ class for the interface implementation. The TIE approach gives a complete separation of the class hierarchies for the IDL C++ classes (as generated from the IDL interfaces by the IDL compiler) and the class hierarchies of the C++ classes which are used to implement the IDL interfaces.

We needed to find the syntax difference between CORBUS and ORBIX in those C++ classes and change to the corresponding format. For example, they have different ways to report system errors. In order to support IIOP, a server Interoperable Object Reference (IOR) was retrieved when the server object was invoked. Then server changed the object reference into a string and wrote the string into file. The file was located on the partition mounted on client to facilitate client reading the file. At last, the server program has to link with the new skeleton.

At the client side, since the IDL files almost unchanged, the code did not have to be changed. The only thing to change is instead using ORBIX daemon to find the server, we use the IOR to find the server object. The client read from the file mounted from server and got the string. It then converted the string to IOR and used this object reference to refer server object. Also the client program has to link with the new stub.

4.2 Real-time JFLEX implementation

At both the client and the server sides, the Pserver needed to be started as a real-time daemon. RT_Manager objects are needed both at client side and server side to manipulate the real-time information and handle the interaction between Pserver and the client. I will elaborate on both sides of implementation in this section.

4.2.1 Server Side Implementation

At server side, the server needs the real-time information passed from client to schedule the TDMI. This required a modification of the IDL files to support passing real-time information for the real-time methods and support real-time exception. For example, the declaration of the method to create new state is:

```
JflexState      NewState(in string name, in string desc);
```

We added the RT_Environment parameter to make the method signature:

```
JflexState NewStateRT(in string name, in string desc, in RT_Environment rt_env) raises(RT_Exception);
```

The server side implementation for NewStateRT() as follows:

```
JflexState_ptr OutlineServer_i:: NewStateRT(const char * name, const char* desc,
 const RT_Environment& rt_env, CORBA::Environment &IT_env) throw(CORBA::SystemException,
RT_Exception)
{
    RT_Manager_Server rt_mgr(rt_env);

    try{
        rt_mgr.START_RT();
        rt_mgr.Start_Atomic_CORBA_Call();
     :
     //code for create new JflexState and update the database
     :
     rt_mgr.End_Atomic_CORBA_Call();
       rt_mgr.END_RT();
       return statePtr;          //return the new JflexState created
    }
    catch(const RT_Exception &rtex){
        cerr<<"at JflexState_i::NewStateRT "<<rtex.reason<<endl;
```

```
            cerr.flush();
            rt_mgr.STOP();
            return NULL;
        }
        :
}
```

A RT_Manager_server object is constructed using the RT_Environment from the client as an argument. The bulk of the work is done inside the START_RT() method and is transparent to the JFLEX server. This method determines the network delay, calls the functions to register with the real-time daemon, calculates the Transient Priority for the server thread, sets the thread to a new priority, and arms the timer according to the new deadline. Methods Start_Atomic_CORBA_Call() and End_Atomic_CORBA_Call() tell the server that if a real-time exception is raised between these two function calls, delay throwing of the exception until all the work between these two function calls is done. All the operations to the database are between these two function calls. This way, we make the transaction on the database atomic. If the server has not missed its deadline during the service, then END_RT() disarms the clock, performs some clean up and results are sent back to the client. If the timer expires (i.e., the deadline is missed), a CORBA exception of type RT_Exception is raised in the server. The server thread catches this exception, and performs any necessary cleanup and recovery operations.

4.2.2 Client Side Implementation

At the Client side, we need a global object RT_Manager_Client to handle the real time information. The RT_Environment data structure stores the timing constraint parameters, Quality of Service, importance and transient priority etc. We specify time constraints and importance through RT_Mangaer_Client.

The methods of the RT_Manager_Client are responsible for assembling the RT_Environment that will be attached to the TDMI for use by the ORB, the object services and server implementation. After RT_Manager_Client sets up the real-time environment, it will spawn a thread for each method call invocation. An example of client code for a thread is like following:

```
#include RT_Manager_client.h

RT_Manager   rt_mgr;

Struct newState_para
{
    OutlineServer_ptr osptr;
    Char* name;
    Char* description;
    JflexState_ptr jsptr;
}

void* newState_thread (void* arg)
{
    newState_para       nstate_para=*((newState_para*)arg);
    rt_mgr.START_RT();
    try{
```

```
        RT_Environment  rtenv=rt_mgr.Get_RT_Env();
        (nstate_para.osptr)->NewStateRT(nstqate_para.name, nstate_para.description, rtenv);
        rt_mgr.END_RT();
    }
    catch(const RT_Exception &rtex)
    {
        //handling the real-time exception
    }
    :
}
```

As we mentioned earlier in our real-time model, the client spawns a new thread each time a request to the server is made. The newState_para contains all of the information needed for the new thread. The function newState_thread defines the work that the new thread will do. Inside that function, the START_RT() will register the new thread with a real-time daemon on its local node to calculate and assign a global priority called Transient Priority to the TDMI. Also a timer with the proper signal handling will be armed according to the deadline. The END_RT() method disarms the alarm, and communicate with the real-time daemon to change the Transient Priority of TDMI to its base priority. If the client misses its deadline, a CORBA exception of type RT_Exception will be thrown from a signal handler to the calling thread. That thread can catch the exception and will execute the STOP() method to deregister with the real-time daemon and release the resources.

An example for client main program is like following:

```
int main()
{
//RT init call
```

```
RT_Manager_Init();
:
try{
//set the importance as 2
rt_mgr.Set_Importance(2);


//set constraints and scheduling parameters
//deadline=NOW+2 seconds
rt_mgr.Set_Time_Constraint_Now(By, REL,2,0);
rt_mgr.Start_RT_Invocation(newState_thread, (void*)&nstate);
//start TDMI: 1) calculate Transient Priority
//            2) call RT Daemon and register as an active client
//            3) map Transient Priority to this node's Priority
//                set and change this thread to the new prioriy
//           4) arm the timer
rt_mgr.End_RT_Invocation();
//finish TDMI       1) call RT Daemon and deregister as a client
              2) disarm the timer
              3) restore this thread to its original priority
}
catch (const RT_Exception &rtex)
{
:
}
:
}
```

In this example, we set a relative deadline of 2 seconds in the Set_Time_Constraint_Now() method. The bulk of the work is done inside the Start_RT_Invocation() function and is transient to the client. Start_RT_Invocation() calls the functions to register with the real-time daemon, calculate the Transient Priority for the client, set the client to a new priority and arms the timer according to the client's deadline.

After the above sequence is complete, the client makes the CORBA call to the table. The RT_Environment that is sent with the call contains the timing information computed by the RT_Manager_Client. At this point the request is scheduled on the server as described before. If the client has not missed its deadline during the CORBA call, then End_RT_Invocation() disarms the clock and performs some clean up. If the timer expires (i.e., the deadline is missed), a CORBA exception of type RT_Exception is raised in the client. The client catches this exception and performs any necessary recovery operations and cleanup.

Chapter 5

Evaluation

After the implementation was completed, several tests were done to show that the new JFLEX support IIOP protocol and perform at real time with the expected results. Then a suite of random requests with different deadlines was executed to evaluate the performance the new JFLEX project. We also measured the overhead introduced by the real-time function calls, and performed a series of tests to determine how well the real-time JFLEX met deadline.

5.1 Testbed Construction

A set of tests was done to retrieve plan information to prove the correctness of JFLEX supporting IIOP. The tests were generated from the following parameters: importance, deadline of the TDMI and different request such as creating/deleting the JFLEX state. For showing the correctness of the new JFLEX project, each request was verified by using client to retrieve information after the request and by retrieving information directly from the MSQL database through database command. Each test was performed on our RTCORBA on Solaris, with expression and enforcement of timing constraints.

The analysis of the real-time performance was based on the comparing the time needed to finish more important requests under different loads between original JFLEX and real-time JFLEX. In this thesis, the results of each test were averaged and analyzed over 10 trials producing in error of at most 1% in most cases.

All testing was performed on two Sun Sparc workstations (IPX and Sparc Station 5) on our department LAN with a fixed number of CORBA clients and a server on each computer. Global Time service NTP server is running on Sun Sparc Station 5, NTP client is running on Sun Sparc IPX. Network delay was measured under different loads using the test C program, and was found to be approximately 0.30 second (s) except the first request usually is 0.72s because of the need of finding routing information for the first time.

5.2 Testing IIOP for new JFLEX

The support of IIOP for the new JFLEX was tested by starting a server at one node, and having the server write the IOR to a file that is on the partition mounted to the other node. The client at the other node used that file to find the server and send its request for JFLEX plan information. After the client received the plan information, it retrieved the information from the database through database operation at server side. These tests indicated that the client was able to bind to, and access, the server via IIOP.

5.3 Testing the Real-time JFLEX

The real-time feature of the new JFLEX was tested by periodically starting up a client on one node (on a Sparc IPX station) that sent a request to a server on the other node (on a Sparc Station 5). The purpose of this testing was to determine correctness and the overhead produced by the real-time features. Recall that in incorporating the TDMI support into JFLEX, an extra parameter (struct RT_Environment) was added to all method invocations to be executed in real-time. Thus, an extra data copying, moving, dereferencing and transmission was done by the Stubs/Skeletons/ORB. And signal handling, threads create and Priority scheduling also adds to system overhead.

Correctness. The correctness of the implementation was tested by running a set of clients with short (0-5 second), medium (5-10 seconds) and long (10-14 seconds) deadline on various requests (add, delete, change JFLEX state). The clients and server threads were forced to miss their deadlines at different phases of their execution as at the very beginning, in the middle, and at the end of database operations. As mentioned earlier, CORBA exceptions of type RT_Exception were raised and processed successfully by both client and server thread, showing that the deadlines were actually missed in all cases. And we checked the database contents each time the deadlines were missed. There was no corrupt data. And all the requested operations were fulfilled successfully.

Performance. We start several clients on a Sun Sparc IPX station with different deadline and importance. Two sets of tests were done for the same importance with

different deadlines and the same deadline with different importances. The results always show that the more important or shorter deadline, the earlier the tasks were finished.

For non-real-time JFLEX, we started five clients running in the background with a UNIX script. Each request was to delete certain state in the database. Five clients finished the request in random order, we could not make certain requests finished earlier. For real-time JFLEX, we did two set tests. The first set is starting four clients with same deadline but different importance. The deadline is given 6 seconds, so each request could be finished without miss deadline and we can get the time for processing the request. Table 5-1 shows that the more important the request, the earlier the request will be finished. The second set of test is starting four clients with same importance but different deadlines. The table 5-2 shows that the shorter the deadline, the earlier the request was finished.

Overhead and Latency. The Server implementation was tested on a Sun Sparc Station 5. A client implementation was on a Sun Sparc IPX station. Clients with real-time features and without real-time features were tested with different requests. Time difference was estimated as time delay caused by processing additional real-time function calls.

For non-real-time JFLEX, we recorded the time for processing the requests (add, delete state and change GeographicArea of a state). For real-time JFLEX, we found it gave different request processing time for different importance, which might have happened because system processes and daemons competed with request for CPU time etc. Since we do not know what priority operating system assigned to the user process, I

did the requests using importance 0,1 and 2 and used their average as processing time for real-time JFLEX. We tested 7 times for each requests and averaged the time as the processing time. The processing time might vary as high as 30 percent for the same request, which we believe resulted from the workload of the server at the specific moment. Average overhead was computed and compared between RT and Non-RT JFLEX in Table 5-3. Note that we always discarded the data from first request, since it is usually quite larger than other data probably because the object was first loaded into cache or memory.

Although the overhead seems quite large compared to the original processing time, notice for high importance such as 2, the overhead was compensated by request putting its work ahead of some system processes through the priority scheduling.

Meeting Deadline. To measure the effectiveness of real-time CORBA at meeting JFLEX deadline, we started the four clients with a UNIX script. The clients have requests with different deadlines or different importance. We checked which clients missed their deadline. And we also tested these clients with same request for non-real-time JFLEX. From Table 5-4, we can see for non-real-time JFLEX, the two clients with higher importance missed their deadlines. But for real-time JFLEX, only one client with a lesser important request missed its deadline. This proves the advantage that real-time JFLEX has over non-real-time JFLEX. That is, although more deadlines might be missed due to the overhead of the real-time mechanisms, the most important deadlines are more likely to be met with real-time JFLEX.

Importance	Deadline (s)	Start Time (s)	Start Time (ms)	End Time (s)	End Time (ms)
0	6	897768975	910591	897768979	556065
1	6	897768975	899804	897768979	525502
2	6	897768975	889287	897768979	492916
3	6	897768975	921964	897768977	909373

Table 5-1 Start and End time for a set of requests with same deadline

Importance	Deadline (s)	Start Time (s)	Start Time (ms)	End Time (s)	End Time (ms)
1	2	897762033	984914	897762035	938783
1	3	897762033	952631	897762036	83432
1	4	897762033	973953	897762037	677277
1	5	897762033	962079	897762037	710380

Table 5-2 Start and End time for a set of requests with same importance

	Importance	Addition Time (s)	Deletion Time (s)	Change Time (s)
NonRT	N/A	1.55	1.59	1.65
RT	0	3.27	5.42	4.30
RT	1	2.19	3.74	3.69
RT	2	1.78	1.64	1.64
RT Average		2.41	3.6	3.21
Overhead Average		0.86	2.01	1.56

Table 5-3 Average Overhead for different requests

	Request Number	Importance	Deadline (S)	Starting Time (S)	Starting Time (ms)	Ending Time (s)	Ending Time (ms)	Process Time(s)	Miss Deadline
Non-real-Time request	1	1	6	898650460	123635	898650463	313711	3.190	No
	2	1	4	898650460	122284	898650461	741606	1.619	No
	3	2	3	898650460	223238	898650466	468509	6.245	Yes
	4	3	2	898650460	173931	898650464	884655	4.711	Yes
Real-time request	1	1	6	898651493	864337	898651498	313711	4.897	No
	2	1	4	898651501	432512	N/A	N/A	N/A	Yes
	3	2	3	898651500	167601	898651500	712006	0.644	No
	4	3	2	898651493	808511	898651495	555715	1.748	No

Table 5-4. Performance compare for RT JFLEX versus non-RT JFLEX

Chapter 6

Conclusion

6.1 Contribution

This thesis has applied real-time and IIOP technology to a large legacy software application. The basis for the work was the real-time CORBA model developed by real-time research group at URI (University of Rhode Island). It focuses on the migration of ORB implementation, expression and enforcement of real-time constraints for JFLEX.

The result of thesis shows migration of the application from one ORB to another can reuse large amount of original code, due to CORBA compliance and object-oriented programming style. The result of thesis also shows the real-time model developed by the real-time group at URI is suitable and reliable for real-time application. At the same time, this thesis makes the JFLEX application more suitable in the practical world. Further it provides a methodology as to how to add real-time features into a distributed application without changing too much code, which is very important since most of development of software is built one feature after another.

6.2 Limitation and Future work

One drawback to the current RT JFLEX is that the JFLEX server is not multi-threaded, which prevents concurrent execution on server, thus increases waiting time for requests.

Another drawback is that we finish the request even if the real-time exception is raised which can leave outdated information in the database.

There is still significant work to be done to meet the practical use standards in the Real-time JFLEX. This includes switching the server main program to multithread program. For this we need to modify the current program to put an in-request pre marshal filter pointer [10] at beginning of the server main program to let server spawn a thread to handle each request. Once the server program becomes multithreaded, we need to implement concurrency control. And we can also substitute the MSQL with other more advanced DBMS such as ORACLE [11], which support transactions, to make data recovery easier for real-time exception.

The success of the migration the ORB implementation and the incorporation of real-time support for JFLEX, proves the good isolation capability of CORBA, and provides a practical methodology to add real-time feature to a distributed application.

List of References

[1] The Object Management Group, "CORBA service: Common Object Service Specification", Revised Edition March 31, 1995.

[2] The Object Management Group, "The Common Object Request Broker: Architecture and Specification", Revision 2.0 July 1995.

[3] Liu, J., "Real-time System ", Prince Hall Press, May 1998.

[4] Rajkumar, Ragunathan, "Synchronization in Real-time Systems: A Priority Inheritance Approach", Kluwer Academic Publishers, Boston, MA 1991.

[5] Victor Fay-Wolfe, Lisa Cingiser DiPippo, Roman Ginis, "Expressing and Enforcing timing Constraints in Orbix", ORBIX Award submission, April 1997.

[6] Mills, D.L., "Internet time synchronization: the Network Time Protocol", IEEE Trans. Communications COM-39, pp1482-1493, October 10, 1991.

[7] Mills, D.L., "Network Time Protocol (version 3) specification, implementation and analysis", Report Request For Comments (RFC)-1305, University of Delaware, March 1992. Currently available at HTTP site: http://www.eecis.udel.edu/~mills/bib.html.

[8] Mills, D.L., "On the accuracy and stability of clocks synchronized by the Network Time Protocol in the Internet system", ACM Computer Communication Review 20, pp.65-70, January 1990.

[9] Igor N. Zykh, "Real-time Event Service",Master thesis May 1997

[10] IONA Technologies PLC, ORBIX Program Guide, March 1997.

[11] Rachel Becker, Matthew Bennett, Winnie In-Kuan Cheang, “Oracle Unleashed”, 1996 by Sams Publishing

Bibliography

Coplien, J., "Advanced C++ Programming Style and Idioms", Addison-Wesley Publishing Company, Reading, MA, 1992,

Coulouris, G., Dollimore, J., "Distributed Systems. Concepts and Design", Second Edition. Addison-Wesley Publishing Company, Reading, MA. 1994

Donohoe, P., Shapiro, R., Weiderman, N., "Hartstone Benchmark User's Guide. Version 1.0", Carnegie Mellon University, Software Engineering Institute, March 1990.

Gallmeister, B., "POSIX .4. Programming for the real world", O'Reilly & Associates. Inc. , Sebastopol, CA. 1995.

Guttman, M. & Matthews, J., "The Object Technology Revolution", John Wiley & Sons, Inc. New York, NY, 1995.

IEEE Standard for Information Technology, "Portable Operating System Interface (POSIX). Part1: System Application Program Interface. Amendment 1: Realtime Extension", Institute of Electrical and Electronics Engineers (IEEE), Inc. New York, NY, 1994.

Levine, J., Mason, T., Brown, D., "Lex&Yacc", O'Reilly & Associates. Inc., Sebastopol, CA. 1992

Lippman, S., "C++ Primer", Addison Wesley, Reading, MA, 1993

Mullender, S., "Distributed Systems", Second Edition, reprinted in 1994, Addison-Wesley Publishing Company. Reading, MA.

Silberschartz, A., Peterson, J., Galvin, P., "Operating System Concepts", Addison-Wesley Publishing Company, Reading, MA. 1992

Stroustrup, B., "The Design and Evaluation of C++". Addison-Wesley Publishing Company, Reading, MA, 1994

Tanenbaum, A., "Distributed Operating Systems", Prentice Hall, Englewood Cliffs, NJ. 1995.