Client-server applications on distributed-parallel and distributed-object platforms

Richard Matthew Tito

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Recommended Citation
https://trace.tennessee.edu/utk_gradthes/10973

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.
To the Graduate Council:

I am submitting herewith a thesis written by Richard Matthew Tito entitled "Client-server applications on distributed-parallel and distributed-object platforms." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dinesh Mehta, Major Professor

We have read this thesis and recommend its acceptance:

Bruce Whitehead, Kenneth Kimble

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)
To the Graduate Council:

I am submitting herewith a thesis written by Richard Matthew Tito entitled "Client-Server Applications on Distributed-Parallel and Distributed-Object Platforms." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Dinesh Mehta, Major Professor

We have read this thesis and recommend its acceptance.

[Signatures]

Accepted for the Council:

[Signature]

Associate Vice Chancellor
and Dean of The Graduate School
Acknowledgements

I would like to begin by thanking the brilliant and persistent members of my committee: Dr. Dinesh Mehta, Dr. Bruce Whitehead, and Dr. Kenneth Kimble for the help they provided at all stages of this thesis. I am grateful to them for allowing me the flexibility in choosing my research topic. Secondly, I would like to thank the UTSI Computer Center: Richard Hopwood, Linda Hall, and especially Harry Ferber for his advice on this thesis and knowledge of networks, operating systems, and protocols. I believe working for the Computer Center was one of my greatest experiences at UTSI and has helped me gain more than enough experience with network problems, LANs, WANs, and databases; all of which contributed to this thesis. In that respect, I will always admire the faculty, staff, and students of UTSI.

For their endless supply of technical papers and resources, I would like to thank Mary Lo and the UTSI library staff whose unique personality and resources can never be equalled. I believe the library staff is the backbone of the academic and research community at this University.

Finally, for all their support, love, and understanding, I would like to dedicate this thesis to my friends, family, and parents Richard Peter Tito and Bernadette Matteo Tito, to whom I will never be able to thank enough.
Abstract

Client-server applications have been traditionally implemented on distributed-object platforms. Although very little research has been done on developing client-server systems on distributed-parallel platforms, these platforms offer advantages which stem from parallel computing. This thesis compares two specific platforms, PVM and CORBA, for implementing client-server applications and presents a client-server system which is used to compare both platforms. This system consists of networked, client and server tasks which search the entire chess game tree for a 'best move' based on a given gameboard and search depth. Both the client and server tasks were implemented using PVM [4] and ORBeline [5].

The results indicate that as communication increased between client and server, PVM is faster than ORBeline for the tests and situations presented. As computation increased, both platforms performed comparably. Other related areas which are useful to developers designing client-server systems in PVM include Distributed-Object PVM [25] and an RPC facility for PVM [26].
# Contents

1 Introduction .............................................. 1

2 Introduction to Distributed-Parallel Computing .......... 5

2.1 P4 .................................................. 6

2.2 Express .......................................... 7

2.3 Linda ............................................. 7

2.4 PVM (Parallel Virtual Machine) ......................... 8

2.5 Components of PVM .................................. 9

2.5.1 PVMD ........................................... 9

2.5.2 TID ............................................. 11

2.5.3 PVM Library Interface Routines ..................... 12

2.6 PVM Message Passing .................................. 12

2.7 TCP and UDP (Transmission Control Protocol and User Datagram Protocol) ....................................... 13

2.8 OSI (Open Systems Interconnect) Model ................. 15
3 Introduction to Distributed-Object Programming

3.1 The OMG (Object Management Group) and CORBA (Common Object Request Broker Architecture) ........................................... 20
3.2 ORB (Object Request Broker) ........................................... 22
3.3 IDL (Interface Definition Language) ........................................... 23
3.4 ORB Components ........................................... 24
  3.4.1 ORB Core ........................................... 25
  3.4.2 Stub and DII (Dynamic Invocation Interface) ........................................... 25
  3.4.3 BOA (Basic Object Adapter) ........................................... 25
  3.4.4 Skeleton ........................................... 26
3.5 ORB Message Passing ........................................... 26
3.6 Creating a CORBA Application ........................................... 26
3.7 ORBeline ........................................... 31
3.8 ORBeline Protocols ........................................... 32
3.9 CORBA and ORBeline ........................................... 33
3.10 PVM and CORBA ........................................... 33

4 Experimentation ........................................... 36

4.1 Factors Which Will be Measured ........................................... 37
4.2 Simple Data Retrieval Over a Network Using Clients and Servers ........................................... 37
List of Tables

4.1 Number of Packets Created by PVM and CORBA for Simple Client Requests ................................................. 38
4.2 User Time (in Seconds) When Client and Server are on the Same Machine ....................................................... 39
4.3 Response Time (in Seconds) When Client and Server are on the Same Machine ....................................................... 39
4.4 User Time (in Seconds) When Client and Server are on Different Machines ......................................................... 40
4.5 Response Time (in Seconds) When Client and Server are on Different Machines ......................................................... 40
4.6 Chess Server Response Time (in Seconds) When Client(s) and Server are on the Same Machine ........................................... 42
4.7 Chess Server Response Time (in Seconds) When Client(s) and Server are on Different Machines ........................................... 44
4.8 Chess Server Response Time (in Seconds) For 8 Clients and 2 Servers on Different Machines .......................... 45

4.9 Chess Server Response Time (in Seconds) For Client(s) and Server(s) on Different Machines .......................... 45

4.10 Chess Server Response Time (in Seconds) for Client(s) and Server(s) on Different Machines .......................... 47
List of Figures

2.1 Components of PVM: PVM Daemon (PVMD), Task Identification Number (TID), and PVM Library Routines .................. 10

2.2 The OSI Model .............................................. 14

3.1 The OMA Reference Model ................................. 21

3.2 The ORB functions as the link between client and server ... 23

3.3 The path from client to server via the ORB core uses DII client functions, the stub, BOA, and skeleton .................... 24

4.1 Board-state used for initial chess server tests ............... 43

4.2 Board-state used for last test ............................. 46
Chapter 1

Introduction

Client-server, distributed-parallel, and distributed-object platforms are currently the newest trends both in industry and in the academic community. On large networked systems, a client-server architecture provides the framework for portable, efficient, and scalable applications. Along with client-server applications, the demand for distributed-parallel and distributed-object platforms has also increased. On one hand, distributed-parallel computing provides speed by dividing up work among several networked, computational resources; on the other hand, distributed-object computing provides "all the benefits from object-oriented programming: modularity, encapsulation, and reuse" [1].

The recent popularity of client-server applications is due to the fact that both information and computation can be distributed over a network. From a business perspective, client-server computing provides the capacity to decentralize informa-
tion across a network for increased security, speed, and scalability. A client-server application is defined as any network application which separates or distributes work into client and server tasks. Conceptually, the clients in a client-server application may be computational tasks, data manipulation processes, or even user-interfaces which exist on network workstations. These clients make requests to a server task which provides services, such as computation or data retrieval, for the clients. To take advantage of client-server computing, developers designing network applications will isolate certain types of work which is best performed by clients and certain types of work which is best handled by a server or servers.

The main advantages of a client-server environment are portability, scalability, and efficiency. A client-server environment is portable since clients can exist anywhere on a network and can remotely access the server or servers from anywhere on the network. A client-server environment is scalable since the server is a separate entity from the clients and since it can be modified or replaced without any impact on the clients. Also, since the only network traffic consists of requests and responses between the client and server, a client-server environment can be made to be efficient as well as secure.

With increased technology in distributed-parallel and distributed object platforms, the question that needs to be addressed is which platform is best for implementing client-server applications. Research on implementing client-server applications on distributed-object platforms has become popular. One related
experiment included a client-server application implemented on C select (sockets) interface, C++ wrappers, and CORBA (Common Object Request Broker Architecture) platforms\footnote{The client side is described in [3].} [2]. The CORBA platform is explained in detail in Chapter 3. This experimental application allows investment brokers to query the price of a stock from a distributed quote database. Experiments were conducted to test the tradeoffs between extensibility, robustness, portability, and efficiency among the platforms. The results of these experiments showed that programming with C select produces an efficient application. However, since sockets and select are low-level interfaces, additional work to maintain the server must be performed. Also, the server must have application-specific service behavior hard-coded directly in the program. This will make it difficult to handle multiple requests from clients simultaneously. The C++ wrappers solution provided object-oriented interfaces which encapsulates lower-level network programming interfaces. This made programming simpler than the previous solution, improved portability, and increased reusability and extensibility. However, there were still drawbacks with the C++ wrappers solution caused by the extra work which is needed to ensure secure and fault-tolerant communication between the client and server. In addition, C++ wrappers does not have tools for object location, object activation, and object copying (portability). All of these issues were solved by using a CORBA implementation.
Very little research has been done on implementing client-server applications on distributed-parallel platforms such as PVM [4] (Parallel Virtual Machine). The PVM platform is explained in detail in Chapter 2. Since distributed-object platforms such as CORBA were intentionally designed for implementing client-server applications, they have a clear advantage over distributed-parallel platforms. Conceptually, however, client-server applications can be designed and implemented on distributed-parallel platforms to provide the same functionality and benefits of a distributed-object platform, with added benefits of distributed computing. This thesis will evaluate the implementation of client-server applications on distributed-parallel and distributed-object platforms and gather conclusions on the advantages and disadvantages of two specific platforms, PVM and ORBeline[^], as a result of research and experimentation.

Chapter 2 will introduce distributed-parallel platforms and will focus on PVM. Chapter 3 will introduce distributed-object platforms and will focus on CORBA. Chapter 3 will also discuss the similarities and differences between two specific platforms, PVM and ORBeline [5]. Chapter 4 will explain each experiment performed and the results that were obtained. Chapter 4 will also explain the meaning of the results obtained and will also provide a discussion on other issues pertaining to both platforms and the experiments performed. Finally, Chapter 5 will discuss conclusions and future work.

[^]ORBeline is a complete CORBA implementation developed by Post Modern Computing, Inc.
Chapter 2

Introduction to
distributed-Parallel Computing

Parallel computing is the process of splitting up work among more than one computational resource. Recent advances in parallel computing have been the result of developments in both MPP (Massively Parallel Processor) machine and distributed system technology. A MPP machine is a single parallel machine with identical processors. MPPs are the most expensive and most powerful individual computers in the world today and are used mainly for high-end computation by large organizations and the Government. Distributed systems, however, are made up of a network of (possibly different) machines whose processors may vary in speed, capability, and resources. An MPP machine is similar to a distributed system in that both allow developers to take advantage of SPMD (Sin-
gle Program, Multiple Data), data-parallel, message-passing, object-oriented, and shared-memory paradigms [6]. The advantage of distributed systems is that the computational power of a distributed system can reach or exceed the level of a MPP machine without costing even a fraction of what today’s MPP machines cost. Because of this, distributed systems and software applications for distributed systems have become more popular.

Distributed systems can improve “collaboration through connectivity and internetworking, performance through parallel processing, reliability and availability through replication, scalability and portability through modularity, extensibility through dynamic configuration and reconfiguration, and cost effectiveness through resource sharing and open systems” [7]. Some software packages which allow programmers to implement general-purpose, distributed applications on networked machines include P4 [8], Express [9], Linda [10], and PVM [4]. Each of these packages will be briefly discussed below.

2.1 P4

P4 is a library supporting both the shared and distributed memory models of parallel computation. P4 was developed at Argonne National Laboratory and uses a master/slave hierarchy for process management. Some of the constructs
which make up the P4 library include send, recv, broadcast, global maxima, global minima, and barrier synchronization.

2.2 Express

Express is a collection of tools which provide programmers communication, I/O, and parallel graphics routines for concurrent computational programming. The Express toolkit was developed by ParaSoft Corporation and contains various tools including a VTOOL, which graphically displays algorithms; an FTOOL, which analyzes programs for parallelization; and an NDB tool, which debugs parallel programs.

2.3 Linda

Linda is a programming model which introduced the concept of 'tuple space' where, unlike a strict shared-memory or message-passing architecture, processes which cooperate can communicate by manipulating records, or tuples. The tuple-space is an associative, shared memory area which processes can access by using embedded constructs. Linda was developed at Yale University and provides programmers with the advantages of the shared memory model of programming.
2.4 PVM (Parallel Virtual Machine)

PVM is a software package which allows a group of networked machines to be used as a parallel computational resource. Beginning in 1989 as a research project between Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University, PVM has gained widespread use and has been continuously improved and updated. PVM is based on the idea of simulating a message-passing MPP machine over a network of distributed resources. These resources may be single CPU workstations, shared-memory multiprocessors, distributed-memory machines, vector supercomputers, specialized graphics engines, or scalar workstations; and must run a UNIX-based operating system\(^1\). PVM is portable and runs on a variety of network architectures including ethernet, FDDI, and token ring. These are some of the reasons why PVM has become so common in the distributed-computing community.

From a programmer's perspective, PVM is flexible when creating applications. An application programmer using PVM can decide how many and which machines to use to distribute work in the virtual environment. These machines make up what is called a PVM session. In addition, a programmer may either view the virtual machine as a single computational source or may choose what tasks will run on what specific machine or machines.

\(^1\)Some independent groups have also developed PVM ports for DEC's VMS, IBM's OS/2, and some other operating systems.
PVM was originally designed with fault-tolerance, scalability, heterogeneity, and portability in mind. PVM is fault-tolerant since it has tools which, when implemented correctly, can produce applications with the ability to withstand host and network failures by providing notification if an application crashes. Also, there is no master control process in PVM; control is distributed in the interest of avoiding performance bottlenecks [11]. PVM is considered scalable since a single PVM session can be expanded to include hundreds of different hosts which can run thousands of tasks. Since PVM supports common languages such as C, C++, and Fortran, and since it does not use operating system dependent features, it is portable as well as flexible.

2.5 Components of PVM

As shown in Figure 2.1, the main components of PVM are the PVM daemon (pvmd), the task identifier (TID), and the PVM Library Routines.

2.5.1 PVMD

The PVM daemon, called pvmd, is the most important component of the PVM system. Pvmd runs in the background of all machines which make up the PVM session and is responsible for all communication between individual machines. Pvmd was designed so that any user with a valid login can install it on a machine. When a user wishes to run a PVM application, he or she starts pvmd by issuing
Figure 2.1: Components of PVM: PVM Daemon (PVMD), Task Identification Number (TID), and PVM Library Routines
'pvm' from the UNIX prompt. The user can manually add hosts or have pvmds on other specified machines start automatically, thus creating the PVM session. Then a PVM application may be started from the UNIX or PVM prompt on any of the allocated machines.

A PVM application usually consists of a series of programs, called tasks, on one machine which communicate with pvmd on that machine. Pvmd then communicates with a pvmd on a remote machine which, in turn, communicates with tasks on the remote machine. For this communication to be possible, a unique number must be assigned to each individual task. This number is called the TID (task identifier).

2.5.2 TID

The second component of PVM is the task identifier (TID) which PVM uses for task-to-task, task-to-pvmd, and pvmd-to-pvmd communication. The TID is a 32 bit integer which contains four fields: an S bit (address bit), a G bit (group bit), an H field (host number), and an L field (private pvmd field). The S bit is the first bit in the TID and is used to address pvmd. When this bit is set, the H field will contain the host number of pvmd. Next, the G bit is set when the TID refers to a groups of tasks, called a group identifier or GID. The H field is the next 12 bits which contains the host number relative to the virtual machine. Finally, the last 18 bits is the L field which is a private field that has different meanings.
depending on the situation. This field adds portability to PVM tasks since it is used differently depending on the architecture of the machine on which the task is being run.

2.5.3 PVM Library Interface Routines

Another component of PVM is its library, called libpvm, of interface routines which an application programmer can use in PVM applications. Included in this library are routines for message-passing, spawning tasks, coordinating those tasks, and modifying the virtual machine [12]. The routines are written in C, to support C and C++ applications, and Fortran, to support Fortran calling conventions.

2.6 PVM Message Passing

A PVM message can be of different lengths and can contain different types of packed or unpacked data. To send messages to other pvmds, pvmd uses the UDP\(^2\) (User Datagram Protocol) system call `sendmessage()`, which routes messages by destination address from one pvmd to another. To send a message to itself, pvmd uses the UDP system call `netentry()` to avoid the packet layer in order to quickly loopback the message. The number of tasks pvmd can handle is limited only by the number of processes allowed by the operating system and as the number of file descriptors available. When a message is sent from a task to a message buffer,

\(^2\)UDP is explained in the next section.
the task does not wait for an acknowledgment. However, the programmer can specify whether a receive will stop execution and wait for something to be placed in its message buffer or whether a receive will simply check its message buffer and continue. The reason for this flexibility is PVM’s protocols: TCP\(^3\) and UDP.

2.7 TCP and UDP (Transmission Control Protocol and User Datagram Protocol)

TCP is a widely used transport layer protocol which is very similar to the OSI transport protocol \([13]\), given in Figure 2.2. TCP is a connection-oriented, stream-oriented, and error-correcting protocol and was originally designed for the reliable transfer of data between remote machines on a network. TCP is connection-oriented since it requires that both machines set up a logical connection before the transfer of data occurs. TCP is stream-oriented because it requires that the data being sent is in the form of characters as opposed to blocks, frames, or datagrams. Finally, TCP is error-correcting since it stores the packets before transmission in case a problem occurs during a transfer.

On the other hand, UDP is a fast, connectionless, non error-correcting protocol which is sometimes used instead of TCP. Since UDP is connectionless, it provides no error correction and requires additional programming overhead if reliability is

\(^3\)PVM v3.3 also uses UNIX-domain sockets instead of TCP for local communication.
<table>
<thead>
<tr>
<th>LAYER:</th>
<th>PROTOCOL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application (File Server Concepts)</td>
</tr>
<tr>
<td>6</td>
<td>Presentation</td>
</tr>
<tr>
<td>5</td>
<td>Session (Courier, Netbios)</td>
</tr>
<tr>
<td>4</td>
<td>Transport TCP</td>
</tr>
<tr>
<td>3</td>
<td>Network IP, IDP Routers</td>
</tr>
<tr>
<td>2</td>
<td>Data Link Ethernet, Token Ring Bridges</td>
</tr>
<tr>
<td>1</td>
<td>Physical (Wiring System Specific)</td>
</tr>
</tbody>
</table>

Figure 2.2: The OSI Model

required. The benefits of UDP are speed and the assignment and management of port numbers which identify individual applications running on different machines.
2.8 OSI (Open Systems Interconnect) Model

The OSI is a 7-layer model developed in 1974 by the standards body known as the ISO (International Standards Organization) [14]. Each layer in the OSI model performs specific functions as part of the overall task of enabling communication between applications on different machines.

As shown in Figure 2.2, the 7 layers from the top are: application, presentation, session, transport, network, data-link, and physical layer. Each layer functions independently of the others. Together, they allow communication between remote applications to execute as if the applications were on the same machine. The application layer is responsible for producing the message to be sent. Some examples that are defined in this layer are terminal emulation and electronic mail. The presentation layer determines the data code set and is responsible for scrambling and descrambling the data as it is transmitted or received. The session layer opens the communication link between the two stations on the network and is responsible for name-to-station address translation. The transport layer is responsible for breaking down and backing up the message into segments. This layer also ensures a reliable transfer and ensures that the data is transmitted in the correct order. The network layer divides up the message segments into packets and is responsible for forwarding the message between stations. Routers are defined at this layer, allowing messages to be transmitted to different networks. The data-link layer
synchronizes transmission and adds a destination address to the packet header so that information can be transmitted over the physical layer. The physical layer is the lowest layer which defines the electronic link and voltage level of the wire or cable and is responsible for the transmission of bit streams. In addition, the wiring and signalling involved to transmit and receive data are defined in this layer.

2.9 PVM Protocols

PVM uses the UDP protocol for pvmd-to-pvmd communication across a network. Since UDP can lose or reorder packets, PVM implements an acknowledgment and retry mechanism. Also, since UDP limits packet length, PVM will fragment long messages. Some reasons TCP was not used for communicating between PVM daemons are scalability and overhead. Since TCP requires a connection between hosts, a file descriptor would be required in the pvmd. Also, since some operating systems limit the number of open files, using the TCP protocol would not have allowed PVM to be scalable across different operating systems. Finally, since TCP is connection-oriented, there would have been additional overhead resulting from multiple logical connections.

For task-to-task and pvmd-to-task communication, PVM uses TCP. TCP was chosen to be the protocol for local communication for a number of reasons. First,
with local communication, speed is not as important as reliability. Local tasks need to communicate with other local tasks and also with the local pvmd in a reliable manner even if increased system calls are required. UDP was not an option in this case since it can lose packets both over a network and within a host and since tasks cannot be interrupted while computing to perform I/O.

In almost all cases, PVM communicates asynchronously. This means that there is no built-in timing or synchronization associated with sending and receiving messages. Tasks send messages to other tasks and continue processing. The time it takes for this communication may vary depending on network use and machine architecture. The application programmer can be confident that the messages are correctly being delivered because of PVM's built-in acknowledgment and retry mechanism. In addition, PVM provides asynchronous notification that can be used to implement fault recovery into applications. A task can request that the system send a message on the following events: *PvmTaskExit*, which is triggered when the task exits or crashes, *PvmHostDelete*, which is triggered when the host is deleted or crashes, and *PvmHostAdd*, which is triggered when new hosts are added to the virtual machine.

The main issue which will be addressed is whether PVM can provide comparable functionality to CORBA for implementing client-server applications. In order

\[4\] There is one exception to this. If the operating system in which PVM is running runs out of message handles, PVM will switch to synchronous-send primitives.
to design a client-server application using PVM, a programmer must create client
and server tasks that communicate by message passing tools provided by PVM,
as shown below:

/* PVM Client Algorithm */
...
/* Send Request(s) to Server */
pvm_initsend(PvmDataDefault);
pvm_pk<type>(&request,1,1);
pvm_send(TID_OF_SERVER, 1);

cc = pvm_recv(-1, -1);
pvm_upk<type>(&server_response,1,1);
pvm_exit();

/* PVM Server Algorithm */
while(TRUE){
    /* Wait on request */
    cc = pvm_recv(-1, -1);
pvm_upk<type>(&request,1,1);
    if (request==EXIT){
        pvm_exit();
        exit(0);
    }
    /* Receive any additional information */
    /* Perform service requested for */
    /* Send Result(s) to Client(s) */
pvm_initsend(PvmDataDefault);
pvm_pk<type>(&result,1,1);
pvm_send(TID_OF_CLIENT, 1);
}
}
Chapter 3

Introduction to

Distributed-Object Programming

Distributed-object programming is a style of programming which takes advantage of object-oriented concepts (such as classes, objects, inheritance, and encapsulation), and applies these concepts to a distributed environment. Object-oriented programming (OOP) closely models real world problem-solving by hiding specific implementation details from application designers\(^1\) and by allowing programmers to think in terms of a problem domain [15]. Some requirements which make this different from other programming methodologies are that with OOP, objects are the primary building blocks to solving the problem, objects are instances of some type or class, and classes are related to each other by inheritance relationships [16]. In general, OOP allows data and

\(^1\)This process is called encapsulation.
behavior to be combined and represented as classes whose instances are objects.

A programming language is said to be object-oriented if it supports objects and inheritance, and if it requires that objects belong to a class. The first object-oriented language to provide objects, classes, inheritance, and dynamic typing was Simula, developed in 1967. Later, Smalltalk added a graphical environment and a dynamic typing mechanism which made programming easier. Today, the most popular object-oriented language is C++ which was invented at Bell Labs by Bjarne Stroustrup in the mid 1980s. C++ adds a mechanism for user defined data types (classes) and other enhancements to the already popular language of C.

With the increasing popularity of distributed-object programming, different standards have been developed to ensure portability with object, client, and server communication over networks. References to implementing distributed-object systems can be found in [17] and [18]. One of the most popular standards of distributed-object programming is the CORBA standard developed by the OMG.

3.1 The OMG (Object Management Group) and CORBA (Common Object Request Broker Architecture)

The OMG is an international organization whose goal is to promote object-oriented technology in software development and is comprised of and supported by university professors, software developers, systems vendors, and users. In May 1989, the OMG was

---

2 Information obtained from Object Orientation FAQ, #1.22.
founded as a non-profit organization by the following companies: 3Com Corporation, American Airlines, Canon Inc., Data General, Hewlett-Packard, Philips Telecommunications M.V., Sun Microsystems, and Unisys Corporation. Today, the OMG has grown to include many more companies such as Microsoft, Novell, IBM, AT&T, Sun/SunSoft, DEC, Siemens, Objectivity, Ontos, and Versant and now has its headquarters in Framingham, Massachusetts. The OMG encourages modular, reusable software design which is easily integrated into other operating systems and hardware. The OMG believes software designed this way is more likely to represent real-world scenarios.

The OMG originally defined the OMA (Object Management Architecture) Reference Model, shown in Figure 3.1, which describes conceptual objects which provide service to clients and other application objects. Another standard developed by OMG which, like
OMA, defines object and client infrastructure is the CORBA (Common Object Request Broker Architecture) standard.

CORBA was created during the early 1990s to define the mechanisms by which clients make requests and receive responses from objects over a network. More formally, objects, or service providers, transparently provide services to client requests on a distributed environment.

CORBA can be compared in some ways to RPC (Remote Procedure Call). With both architectures, a remote service can be invoked as a local function call which is transparent to the programmer [19]. Also, in both cases, the function which hides the details about message passing is called the stub function. With RPC, the stub function which is linked to the client is called the client’s stub and the stub function which is linked to the server is called the server’s stub. With CORBA, however, these are called the stub and skeleton respectively. With RPC, a client stub will block and wait for a response from the server after sending a message. CORBA, however, defines a DII (Dynamic Invocation Interface) which enables client applications to send messages asynchronously to servers. The object and client communication defined by CORBA is provided by an ORB (Object Request Broker), as seen in Figure 3.2.

### 3.2 ORB (Object Request Broker)

The ORB is the most important part of the CORBA standard. Put simply, an ORB is a memory-resident program (TSR or daemon) which takes object requests from client
Figure 3.2: The ORB functions as the link between client and server

programs and directs those requests to servers (or server objects) which provide a service to the clients. The server fulfills the request and returns any values back to the client as necessary via the ORB. Therefore, a client application does not need to know the location of the object. The ORB keeps track of all objects in a specific domain which may span the entire physical network on which the ORB exists. Hence, the ORB enables objects to appear transparent to client applications. Interfaces to objects are defined in IDL (Interface Definition Language) which provides “detailed information about the operations permitted on every object that implements that interface, the arguments each such operation expects, what it returns, and what happens when exceptions occur” [5].

3.3 IDL (Interface Definition Language)

IDL is the CORBA-defined language which describes the interfaces that clients invoke and object implementations provide. The interface in IDL needs to be converted into a compilable language for the specific machine where the client and server will reside.
Most CORBA implementations provide an IDL to C++ mapping.

3.4 ORB Components

The main components of the ORB: the ORB Core, the stub, the DII (Dynamic Invocation Interface), the BOA (Basic Object Adapter), and the skeleton are shown in Figure 3.3.
3.4.1 ORB Core

The ORB core is the main infrastructure of the ORB which is responsible for locating server objects, transmitting requests from the client, and transmitting data or responses back to the client. During the connection-establishment phase of message passing, the ORB core is first located by the stub. The ORB core then locates the server using the BOA (Basic Object Adapter).

3.4.2 Stub and DII (Dynamic Invocation Interface)

The stub and DII are the functions on the client side which provide the capability to route messages. Both are executed during a server request by the client, and will locate the ORB and establish a connection to ensure a reliable transfer to the server. The difference between the two is that the application programmer has the choice of issuing requests to the object by either making a method invocation, in which the stub is used, or by dynamically composing a request, in which the DII is used.

3.4.3 BOA (Basic Object Adapter)

The BOA is the part of the ORB which locates the server and establishes a connection. Once a connection is established, the BOA delivers the client’s request to the server in the form of a method invocation by the skeleton.
3.4.4 Skeleton

The skeleton is the function on the server side which provides the capability to route messages. The skeleton is located by the BOA and delivers any client requests to the server object.

3.5 ORB Message Passing

Figure 3.3 also illustrates a communication path from the client to the server via the ORB. When the client requests a service from the server, the ORB locates the server, connects to the object if necessary, and transfers any arguments to the server from the client. The BOA (Basic Object Adapter), which includes the skeleton, will locate the implementation of the object and make a method invocation which will fulfill the request made by the client. The path of a response from the server back to the client is similar.

3.6 Creating a CORBA Application

There are several steps to creating a CORBA application:

- Step 1 - Create IDL descriptions for the application objects. An example is shown below:

  ```c
  struct object {
    short value;
  };
  ```
interface interf {
    exception CapacityExceeded {};
    boolean add(in object obj_info) raises(CapacityExceeded);
    short retrieve() raises(CapacityExceeded);
    void terminate();
};

In this example, the client and server can share a short integer, called value, by using interface functions add and retrieve. Both add and retrieve must be defined in the code for the server and can be called by clients.

• Step 2 - Invoke the IDL compiler which generates stub and skeleton C++ code.

• Step 3 - Generate the client and server code, as shown below:

/*@ Sample CLIENT function written using ORBeline constructs */
#include <iostream.h>
#include <test1_c.hh>
main()
{
    CORBA::Boolean ret;
    CORBA::Environment env;
    CORBA::Exception *exp;
    CORBA::SystemException *system_exp;
    CORBA::String strout;
    object obj_entry;

    obj_entry.value(INITIALIZATION_VALUE);
    interf *interf.object = interf::_bind(env);

    if (env.check_exception() ≠ 0) {
        exp = env.exception.value();
        system_exp = CORBA::SystemException::cast(exp);
        if (system_exp ≠ NULL) {
            cout ≪ exp->id() ≪ " occurred: " ≪ endl;
            cout ≪ " with minor code " ≪
            system_exp->minorFlag() ≪ endl
            ≪ " with completion status" ≪
system.exp->completed() << endl;
}
return(0);
}
else
    cout << "*** Bound to server ***" << endl;

// Set value of server
ret = interf.object->add(obj.entry, env);

if (env.check_exception() != 0) {
    interf::CapacityExceeded *cap.excep;
    exp = env.exception_value();
    cap.excep = interf::CapacityExceeded::_cast(exp);
    if (cap.excep == NULL)
        cout << "Exception raised was not CapacityExceeded" << endl;
    else {
        cout << "CapacityExceeded exception" << endl;
        cout << "Server already initialized" << endl;
        interf.object->release();
        return 0;
    }
}

if (ret == CORBA::TRUE)
    cout << "Put successful." << endl;
else
    cout << "Put failed." << endl;
interf.object->release();
return(1);

/* Server Header File */
#include <test1.s.hh>
#include <iostream.h>
class obj.list
{
    private:
#define MAX 1
short _obj_count;
object *_.obj_array[MAX];

public:
    obj_list() {
        _obj_count = 0;
    }

~obj_list() {
    for (int i=0; i<_obj_count; i++)
        if (_obj_array[i]) delete _obj_array[i];
}

CORBA::Boolean add_to_list(const object &b) {
    if (_obj_count < MAX)
        return CORBA::FALSE;
    else {
        _obj_array[_obj_count++] = new object(b);
        return CORBA::TRUE;
    }
}

short getvalue(void) {
    if (_obj_count==0) return 0;
    else return (*_obj_array[_obj_count-1]).value();
}

};

class Interf: public interf_impl {
private:
    obj_list object_values;
public:
    Interf(const char *object_name=NULL);
    CORBA::Boolean add(const object &obj_info, CORBA::Environment &_env);
    short retrieve(CORBA::Environment &_env);
    void terminate(void);
};

/* Sample SERVER function written using ORBeline constructs */
#include <signal.h>
```
#include <iostream.h>
#include "server.h"
void signal_handler(int, ...) {
    cout << "SIGNAL**" << endl;
    exit(0);
}

int main() {
    signal(SIGINT, (void (*) (int)) signal_handler);
    class Interf * interf_server = new Interf("Harvard");
    CORBA::BOA::impl Js_ready();
    return(1);
}

Interf::Interf(const char *object_name) : interf_impl(object_name) {}

CORBA::Boolean Interf::add(const object& obj_info, CORBA::Environment& env) {
    CORBA::Boolean ret;
    if ((ret = object_values.add_to_list(obj_info)) == CORBA::FALSE)
        env.exception_value(new interf::CapacityExceeded());
    return ret;
}

short Interf::retrieve(CORBA::Environment& env) {
    short val;
    if ((val = object_values.getvalue()) == 0)
        env.exception_value(new interf::CapacityExceeded());
    else return val;
}

void Interf::terminate(void) {
    exit(0);
}
```
When accessing objects, the programmer has the choice of issuing requests on the object by making a method invocation using a stub or by dynamically creating a request using DII\(^3\). Using a method invocation only involves calling a method function contained in the object. DII allows applications to invoke methods without having access to the object's stubs and without needing an IDL compiler. This makes DII better for developing applications using scripting languages [5].

Unlike PVM, which is itself a collection of tools and software for programmers, CORBA is a standard which provides detailed communication definitions. The specific implementation of the CORBA standard which was used for the experimental phase of this thesis is the ORBeline software package developed by Post Modern Computing Technologies, Inc.

### 3.7 ORBeline

ORBeline [5] is a CORBA implementation which includes an IDL compiler, ORB daemon, and library functions. The reason this system was chosen for experimentation is because it is a complete CORBA implementation which is easily modified to run on different computer architectures and also because it is supported by Post Modern Computing for the academic community.

ORBeline is currently supported on CRAY supercomputer operating systems, SunOS,

---

\(^3\)The client example above uses the method invocation `add` to initialize the server object `.obj.array/`. 
Solaris, OSF/1, UnixWare, HP-UX, IBM/AIX, Windows 3.1, Windows NT, and Vx-Works. Also, ports of ORBeline to Lynx-OS, pSOS, and some other operating systems are being developed.

ORBeline supports persistent and transient objects which developers can create in CORBA applications. Persistent objects are entities which exist after the process which created them is terminated. Persistent objects can be started manually or through activation and can be called by any client on the network. Fault tolerant systems usually take advantage of global, persistent objects. In contrast, transient objects exist only as long as the creating process. Transient objects are usually designed to accomplish simple tasks since they have a limited life-span and can only be used by clients that have received a reference to a transient object. Also, transient objects are suitable for applications requiring the use of a very large number of objects [5].

3.8 ORBeline Protocols

ORBeline uses different protocols, depending on the situation, which are determined at the connection-establishment phase of message passing. Initially, if the client and server are on the same machine, they will be bound together using C++ language mechanisms, thus bypassing the network and ORB altogether. ORBeline uses its own on-the-wire protocol (UDP) on top of TCP/IP for client-to-ORB and ORB-to-server
communication if the client and server are on different machines. During ORB-to-ORB communication, ORBeline uses UDP protocol. Finally, if another vendor's object and ORB need to be called by ORBeline's ORB, the other vendor's on-the-wire protocol is used, thus avoiding the need for a gateway or translator.

3.9 CORBA and ORBeline

ORBeline was designed to be a complete CORBA representation with added features. These features were not mentioned or used previously because they are not traditionally part of CORBA. One such feature is built-in fault tolerance.

Another feature which is not part of the CORBA specification is ORBeline's Dynamic Directory Service. This feature will allow an ORB to dynamically start an object if a client request is made and the object is not active.

3.10 PVM and CORBA

The similarities between PVM and CORBA are that they both provide tools for message passing between application tasks. The details of message passing, in both cases, is transparent to the application programmer. Also, both PVM and ORBeline use the UDP protocol for network communication which is made reliable by implementing an acknowledgement and retry mechanism.

Both PVM and CORBA have the capability to allow developers to create fault-

---

4 A complete discussion of these features can be found in [5].
tolerant systems. If a host machine crashes which contains a CORBA server object, an ORB (if one exists) on another host will try to redirect all future client requests to another copy of the object (if one exists) on another machine. However, with PVM, extra work is required to ensure fault tolerance. `PvmTaskExit` and `PvmHostDelete` can be used by an application programmer to explicitly check if a server task has crashed or if the host has crashed.

There are many differences between PVM and CORBA for developing client-server applications. First, PVM is simply a tool for distributing work among different machines with the only communication being between PVM daemons. CORBA, however, is designed specifically for client-server programming where communication occurs between client, server, and ORB tasks. With PVM, individual clients must know the TID of the server (and vice versa) or be part of a group in order to send requests and receive responses to and from the server. CORBA, however, eliminates this requirement by using an ORB. The CORBA client requests a object service by issuing a member function on that object. The programmer does not need to worry about the location of the object or how to contact that object as these concerns are taken care of by the ORB and IDL respectively. To the programmer, the CORBA call to a remote object looks like standard C++ member function invocation.

CORBA allows the client, server, and ORB to be physically located on completely different machines on the same network. If a PVM session includes all machines on the network, then the functionality is the same as CORBA since the ORB domain spans
the entire physical network. For explicit message passing, PVM contains specific library functions. CORBA, however, provides similar functionality through the use of server objects and method invocations on the server objects\(^5\).

\(^5\)ORBeline does include library functions but are used mainly for exception handling and obtaining information on the ORB.
Chapter 4

Experimentation

In all of the tests which will be explained, there were some intentional similarities in the design and implementation of both platforms. First, the compiler used, in both cases, was GNU C++ for Solaris/SUNOS. Second, for response time considerations, both implementations were compiled and run on the same machines with no outside interference such as additional users and/or jobs running. Finally, the machines involved in the testing consisted of Sparc10 workstations, each containing one 36 Mhz RISC processor and 32 megabytes of RAM, a Sparc10 workstation with one 36 Mhz RISC processor and 64 megabytes of RAM, and a Sparc2000 workstation, with two sequential 40 Mhz RISC processors, 128 megabytes of RAM, and 1 megabyte of cache.
4.1 Factors Which Will be Measured

To compare PVM (v3.3) and ORBeline platforms on client-server applications, the following criteria will be addressed: communication overhead, speed, portability, robustness, and programmer convenience. To measure communication overhead, a test will be performed to count the number of packets sent between two machines where the client, server, and ORB will be resident. The number of packets required for the client to make requests to the server and the server to send back its response will be recorded. To compare speed, different variations of client and server locations as well at the time required to complete a client-server request/response loop will be evaluated. To compare portability, robustness, and programmer convenience, a client-server application will be designed, implemented, and run on both platforms while experiences with each platform will be recorded.

4.2 Simple Data Retrieval Over a Network Using Clients and Servers

4.2.1 Load on the Network

Before testing the speed of each platform, a test was performed to count the number of network packets each platform creates for the client to make a request to the server and then for the server to send back its response. The simplest way to do this, in order to

---

1 This process is called a complete client-server request/response loop
Table 4.1: Number of Packets Created by PVM and CORBA for Simple Client Requests

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of requests</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>PVM</td>
<td>8</td>
<td>44</td>
</tr>
<tr>
<td>ORBeline</td>
<td>27</td>
<td>99</td>
</tr>
<tr>
<td>Difference</td>
<td>19</td>
<td>55</td>
</tr>
</tbody>
</table>

ensure the only variable being tested is client-server communication, is to have the client request a single integer value from the server. This will minimize server computational time in order to focus on communication time only. Recall that just as each pvmd must locate the other pvmds in the PVM session, CORBA's ORB must locate both the client and server objects which can reside anywhere on the network. This overhead created by the ORB should result in additional network traffic for CORBA's implementation of the test. Table 4.1 shows that ORBeline consistently created additional UDP packets during a complete client-server request/response loop.

4.2.2 Speed of Communication

The next series of tests were performed to measure the speed at which each platform establishes communication, transfers a request to the server, and transfers a response back to the client. These tests repeated the previous one exactly except that, in this case, the client and server were timed. Both the user time and the response time were computed. Table 4.2 shows the times obtained for a single server to fulfill multiple

---

2 Pvmbs continuously check with each other every few seconds which results in 2 additional packets.
3 User time is the time the CPU spends on the program itself and not on operating system calls and/or other jobs.
4 Response time is the total time the client must wait for a response, including user and system time.

38
requests from a client. In this test, the user time was recorded and the client and server were located on the same machine. Table 4.2 shows that PVM was approximately 68% faster than the ORBeline implementation when the number of requests exceeded 10.

In the next test, the response time was recorded while the client and server were located on the same machine. Table 4.3 shows that PVM was approximately 86% faster than the ORBeline implementation when the number of requests exceeded 10.

In the next test, the user time was recorded while the client and server were located on different machines. Table 4.4 shows that PVM was approximately 81% faster when the number of requests exceeded 10.

In the next test, response time was recorded and the client and server were located on different machines. Table 4.5 shows that PVM was approximately 83% faster when the
Table 4.4: User Time (in Seconds) When Client and Server are on Different Machines

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of requests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PVM</td>
<td>0.10</td>
</tr>
<tr>
<td>ORBeline</td>
<td>0.10</td>
</tr>
<tr>
<td>Difference</td>
<td>0.00</td>
</tr>
<tr>
<td>% Difference</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.5: Response Time (in Seconds) When Client and Server are on Different Machines

<table>
<thead>
<tr>
<th>Platform</th>
<th>Number of requests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PVM</td>
<td>0.41</td>
</tr>
<tr>
<td>ORBeline</td>
<td>1.64</td>
</tr>
<tr>
<td>Difference</td>
<td>1.23</td>
</tr>
<tr>
<td>% Difference</td>
<td>75%</td>
</tr>
</tbody>
</table>

number of requests exceeded 10. Also, the times in Tables 4.4 and 4.5 are generally larger than the corresponding times in Tables 4.2 and 4.3 because of network communication time.

4.3 The Chess Server

The next test involves the design of the chess server. The chess server is a system where clients request the computation of a ‘best move’ in the game of chess based on a given gameboard and search depth. This application was chosen because it is easily modified for testing and because it illustrates a practical example of a client-server application. By changing the ply depth and the number of clients and servers, the communication/computation ratio can be varied. This makes it possible to study
speed, portability, robustness, and programmer convenience under different scenarios.

The server in this application is the task which is responsible for choosing the best move in a given chess board by searching the entire chess game tree down to a given ply depth. It was implemented using ORBeline as well as PVM in order to make comments on portability, robustness, and programmer convenience. The main component of the server is a recursive search function which evaluates the entire chess game tree while searching for the best possible move, based on a given board configuration. The algorithm is shown below:

Move_Value BestMove(board, turn, depth){
    if (depth=1){
        for each possible move{
            if (checkmate) value = CHECKMATE;
            else if (check) value = CHECK;
            else value=best_move_so_far();
        }
        return value;
    }
    else{
        for each possible move{
            if (checkmate) value = CHECKMATE;
            else if (check) value = CHECK;
            else value=BestMove(newboard, !turn, depth−1);
        }
        return value;
    }
}

The implementation of this algorithm was identical for PVM and ORBeline and was written in standard ANSI C. The only difference between the PVM and ORBeline chess servers was the communication components of the client and server. This made the results of the tests focus primarily on the differences between each platform’s message
passing. The criteria used to determine the best move was a typical, non-heuristic assignment of integer values to pieces and board situations in the game. A detailed discussion of numeric piece assignments to chess pieces can be found in [20].

Throughout the following tests, only the response times were recorded. The variables of the following tests were the location the ORB, client(s), and server(s) and also the number of clients and servers. For the first group of tests, the chess board state used is shown in Figure 4.1. For simplicity, we will assume it is white's turn to move. This game-state is a simple chess setup which results in checkmate for white after 2 moves. Since the algorithm used to evaluate the board is a brute-force algorithm, approximately 7,000 board positions are evaluated.

4.3.1 Multiple Clients and One Server on One Machine

The results of the first test are shown in Table 4.6. In this test there existed one server, while the number of clients that were started simultaneously varied from 1 to 9. Also in this test, the client(s) and server were both located on the same machine. The times reveal that on the average, PVM performed slightly slower.

Some good references to computer chess design and artificial intelligence can be found in [21], [22], and [23].
Table 4.6: Chess Server Response Time (in Seconds) When Client(s) and Server are on the Same Machine

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Platform</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PVM</td>
<td>8.4</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.1</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>3</td>
<td>PVM</td>
<td>8.1</td>
<td>15.1</td>
<td>22.8</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.2</td>
<td>14.2</td>
<td>21.9</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>6</td>
<td>PVM</td>
<td>8.7</td>
<td>15.3</td>
<td>21.5</td>
<td>29.4</td>
<td>36.0</td>
<td>42.7</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.7</td>
<td>15.2</td>
<td>21.3</td>
<td>28.1</td>
<td>36.6</td>
<td>42.2</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>9</td>
<td>PVM</td>
<td>9.0</td>
<td>15.7</td>
<td>22.6</td>
<td>29.0</td>
<td>37.3</td>
<td>43.6</td>
<td>50.3</td>
<td>57.1</td>
<td>63.0</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>9.0</td>
<td>16.6</td>
<td>22.5</td>
<td>28.0</td>
<td>36.0</td>
<td>42.4</td>
<td>48.6</td>
<td>54.3</td>
<td>60.9</td>
</tr>
</tbody>
</table>

Figure 4.1: Board-state used for initial chess server tests
Table 4.7: Chess Server Response Time (in Seconds) When Client(s) and Server are on Different Machines

<table>
<thead>
<tr>
<th>Number of Clients</th>
<th>Platform</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PVM</td>
<td>8.3</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.2</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>3</td>
<td>PVM</td>
<td>8.6</td>
<td>15.9</td>
<td>23.6</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.1</td>
<td>15.3</td>
<td>22.3</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>6</td>
<td>PVM</td>
<td>8.8</td>
<td>15.8</td>
<td>23.4</td>
<td>29.9</td>
<td>37.6</td>
<td>44.2</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.7</td>
<td>15.3</td>
<td>22.4</td>
<td>28.6</td>
<td>36.3</td>
<td>42.0</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>9</td>
<td>PVM</td>
<td>9.0</td>
<td>15.9</td>
<td>23.7</td>
<td>29.9</td>
<td>37.8</td>
<td>44.8</td>
<td>51.0</td>
<td>57.3</td>
<td>64.5</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.9</td>
<td>15.8</td>
<td>22.5</td>
<td>28.0</td>
<td>36.0</td>
<td>43.3</td>
<td>49.3</td>
<td>55.6</td>
<td>61.3</td>
</tr>
</tbody>
</table>

4.3.2 Multiple Clients and One Server on Different Machines

Table 4.7 shows the results of the next test which was identical to the previous one except that the client and server were located on different machines.

The times revealed by these tests indicate that as the number of clients increase on a single-server system, both platforms perform similarly. Also, in these tests it appears that communication time is not a factor since the times obtained when the client(s) and server were on the same machine is similar to the times obtained when the client(s) and server were on different machines.

4.3.3 Multiple Clients and Multiple Servers

The next tests examine the concept of multiple clients and servers. Throughout these tests there existed two or more servers and three or more clients which were started simultaneously. One reason for this setup was to show that both implementations will perform relatively similar in terms of speed. Another reason for this setup was to show
Table 4.8: Chess Server Response Time (in Seconds) For 8 Clients and 2 Servers on Different Machines

<table>
<thead>
<tr>
<th>Platform</th>
<th>Client Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PVM</td>
<td>8.9</td>
</tr>
<tr>
<td>ORBeline</td>
<td>9.9</td>
</tr>
</tbody>
</table>

Table 4.9: Chess Server Response Time (in Seconds) For Client(s) and Server(s) on Different Machines

<table>
<thead>
<tr>
<th>Number of Servers</th>
<th>Platform</th>
<th>Client number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>PVM</td>
<td>9.3</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.4</td>
</tr>
<tr>
<td>2</td>
<td>PVM</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.3</td>
</tr>
<tr>
<td>3</td>
<td>PVM</td>
<td>9.1</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>8.2</td>
</tr>
</tbody>
</table>

that the ORB will consistently locate a server which is not busy for the CORBA clients.

Recall that the PVM clients must communicate with a particular server which, in some respects, limits its flexibility.

Table 4.8 shows the results when 8 clients were started simultaneously from 1 machine while 2 servers existed on separate machines.

Table 4.9 shows the results of another variation. In this test, there were 3 clients while the number of servers varied from 1 to 3. Also in this test, the servers each existed on different machines.

The last test used a different board configuration, shown in Figure 4.26. Like with the board used in the previous tests, white can mate in 2 moves. With this board,

---

*6 Taken partially from [24].*
Figure 4.2: Board-state used for last test
Table 4.10: Chess Server Response Time (in Seconds) for Client(s) and Server(s) on Different Machines

<table>
<thead>
<tr>
<th>Number of Servers</th>
<th>Platform</th>
<th>Client number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>PVM</td>
<td>22.3</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>22.2</td>
</tr>
<tr>
<td>2</td>
<td>PVM</td>
<td>22.2</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>21.0</td>
</tr>
<tr>
<td>3</td>
<td>PVM</td>
<td>22.7</td>
</tr>
<tr>
<td></td>
<td>ORBeline</td>
<td>22.0</td>
</tr>
</tbody>
</table>

however, the number of states that must be evaluated increased (from approximately 7,000) to approximately 26,000. The results of this test are shown in Table 4.10. These results indicate that it takes approximately 22 seconds to find checkmate using the given board configuration. The times reveal that the ORB must have distributed client requests to idle servers in order to perform the same work as PVM, which explicitly divided up work among the available servers.

4.4 Meaning of Results

The first issue that needs to be addressed is the tradeoff between communication and computation. Traditionally, this is a parallel computing issue that applies to these experiments because of the coarse-grained parallelization that occurs when more than one server is allocated on more than one machine. The results in Tables 4.2 through 4.5 showed that, as communication was large, PVM ran faster than ORBeline because of ORB overhead. Since the TID of the PVM server consistently was known to the clients, PVM had an advantage which was apparent during the communication-intensive
tests. The last experiment\(^7\) was designed to test situations where computation is large. The results showed that both platforms perform almost identically as far as speed is concerned. As computation increased, the time differences between both platforms became less apparent.

The next issues to address are portability, robustness, and programmer convenience. Portability can be taken to have two different meanings in the context of comparing client-server platforms on networks. First, portability can refer to the ability of the platform to be transferred across different networks running different operating systems. Portability can also refer to the ability to transfer application tasks between different machines. In both respects, PVM and ORBeline allow tasks to be on completely different machines running different operating systems. If PVM or ORBeline are set up on two different networks, client and server tasks can be run on both configurations. The only difference is PVM requires that all machines which make up a PVM session be pre-allocated. In this respect, CORBA is more portable than PVM since CORBA clients and servers can be started on any machine (that they have been compiled on) without an ORB being present (as long as there is at least one ORB resident on the network).

Another issue that was noticed relates to each platform’s ability to handle inconsistent or incorrect data. In another experiment\(^8\), the number of requests the client made to the server increased to 100,000. This did not present a problem when the PVM client

\(^7\)The results are shown in Table 4.10.
\(^8\)The results of this experiment were not recorded in a table.
and server existed on the same machine. This did, however, present a problem when
the PVM client and server existed on different machines. In this situation, when the
number of client requests increased above 50,000, PVM would abnormally terminate.
Also, before PVM would crash, the system would sit idle for a large amount of time
(20 or so minutes). A severe decrease in PVM's performance was also present when the
number of requests exceeded 5000. These problems were assumed to be due to PVM's
message buffering.

There are certain advantages of both platforms which contributed to programmer
convenience when designing client-server applications. First, it is easier to modify ex-
sting objects in CORBA since it only requires changing of IDL classes. Second, C++
features such as templates, public inheritance, and object reuse provided advantages for
CORBA. Also, programming in CORBA is easier since clients make requests to services
on the network. To simulate this in PVM, the explicit TIDs of servers must be known
to client tasks. Also, since PVM is a completely distributed system, client and server
functionality must be explicitly built into the client-server application.

PVM, on the other hand, is more flexible than CORBA. First, since PVM has no
built in constructs for implementing clients and servers, a client can dynamically assume
the role as a server (and vice-versa) if necessary. In this respect, PVM allows for the
development of client and server tasks which can be customized for a specific application.
Also, when the client-server application is responsible for high-end numeric computation
(modeling, simulation, matrix operations, etc), PVM will provide advantages of parallel
servers. PVM servers can spawn other servers on remote machines (provided they are included in the PVM session) to distribute work.
Chapter 5

Conclusions and Future Work

The purpose of this research has been to evaluate the implementation of client-server applications on distributed-parallel and distributed-object platforms and to gather conclusions of the advantages and disadvantages of two specific platforms, PVM and CORBA (using the ORBeline implementation). The theoretical implications of this research and experiments performed relate to what platform a developer will choose when designing a client-server application. After researching and experimentation, it can be said that there is no existing platform which is best suited for all types of client-server problems. In general, PVM appears better for designing parallel-server applications and client-server systems which require high-end computation or customization. CORBA appears better suited for traditional database applications and structured, client-server systems.

There are many related areas of investigation which could prove to be beneficial to the academic community. First, there has been related work in a DoPVM (Distributed
Object Parallel Virtual Machine) [25] project conducted to provide an object-oriented, distributed environment to PVM via library functions to support shared objects. This is an interesting addition to PVM which could provide the benefits of both distributed-parallel and distributed-object platforms. DoPVM could prove to allow developers to include tasks on hardware that is best suited for particular stages of a client-server application.

Another area of investigation is setting up PVM groups with global broadcasting. Realistically, in a parallel client-server system we want servers to talk with other servers to distribute work. This can be accomplished using global broadcasting in PVM. This feature allows for the design of systems where servers will distinguish messages that are intended for servers from messages that are intended for clients, without using the specific TID of clients and servers. A system could then be designed to provide benefits of distributed-parallel systems to dynamically created servers involved in the application. This area would provide advantages to client-server applications in PVM which are extremely difficult to implement using CORBA.

Other relevant work [26] describes an alternative interface to PVM which supports client-server computing. This is based on a RPC (Remote Procedure Call) type of facility that “permits the specification and export of services which may be invoked by clients using the well-established RPC paradigm and mechanics” [26]. Some of the benefits of this research to PVM include user-transparent load balancing, failure resilience, and adaptive parallelism.
In conclusion, a PVM implementation with comparable functionality to CORBA was presented, tested, and compared to CORBA. The advantages and disadvantages in terms of network load, speed, portability, robustness, and programmer convenience of both platforms were recorded based on experimentation and experiences with each platform. The intention of this research was not to prove that one platform was better than the other, but to present situations where, depending on the problem, using the correct platform can offer additional benefits to the client-server development, implementation, and testing processes.

The author hopes that this research along with the various related areas mentioned will contribute to the advancement of knowledge in similar research topics and also contribute to further developments with distributed-parallel and distributed-object, client-server systems.
Bibliography


Vita

Richard Matthew Tito was born in Laureldale, Pennsylvania on August 29, 1970. He graduated High School in Hazleton, Pennsylvania in June, 1988, and later received a Bachelor of Science degree in Computer Science and Engineering from The Pennsylvania State University.

After working in industry for various companies including IBM, Fujitsu Business Communication Systems, Electronic Specialties, Richey Enterprises, Inc., and Sverdrup Technology, Inc., he obtained a Masters of Science degree in Computer Science from The University of Tennessee.