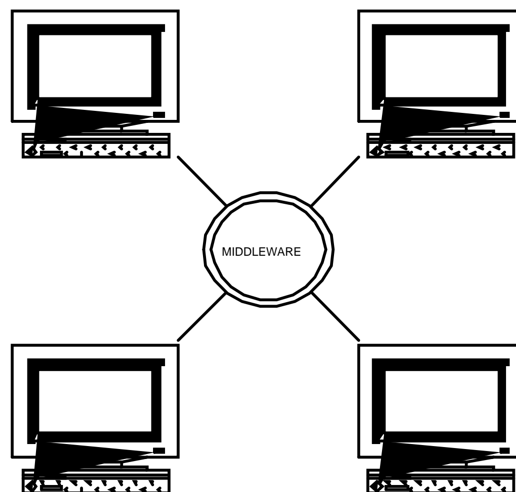
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Laboratoire de Systèmes d’Exploitation / Département d’Informatique

Atomic Broadcast on various middlewares



Niklaus Hirt

Diploma Thesis 1997/98

Professor: André Schiper
Supervisor: Rachid Guerraoui

Abstract

Nowadays, computers become more and more important. They control the flow of money, as well as air traffic and other major elements of our modern world.
This fact holds the danger of people becoming very dependent on the computer-provided services.
As computers are not free of breakdowns, there must be a possibility to avoid complete dysfunction in such a rare case. That is where this diploma thesis starts.
We will give an overview of the possible solutions, providing services that remain functional in case of a system failures. Then we present a possible implementation of such a system, using different algorithms and middlewares. And finally the solutions are tested, in order to find objective conclusions on their performance.

Abstract

Contents

Chapter 1 Introduction **8**

1.1Distributed systems **9**

Accounts Example **9**

Basic structure **11**

Formal representation **11**

1.2Total order and atomicity **13**

Accounts Example **13**

Chapter 2 Total Order **15**

Reliable Broadcast **16**

2.1Sequencer **16**

2.2Skeen **18**

2.3Chandra and Toueg **20**

Consensus **20**

2.4Theoretical performance **22**

Chapter 3 Middleware **23**

3.1Sockets **23**

3.2RMI **25**

3.3CORBA **27**

Chapter 4 Implementation **29**

4.1Basic concepts **30**

4.2First layer **31**

Sockets **31**
RMI **34**
CORBA **36**

4.3Second layer **39**

Application **41**

4.4Registration **43**

Sockets **44**
RMI **45**
Corba **47**

4.5Communication **49**

Sockets **49**
RMI **52**
CORBA **52**

Chapter 5 Performance Test **53**

5.1Test environment **54**

5.2Estimations **55**

Algorithms **55**
Middlewares **55**
Complete estimation **58**

5.3Closed System Model **60**

Message size **64**

5.4 Client Server Model **65**

Case A **65**
Case B **66**
Sequencer saturation **67**

Chapter 6 Conclusions **68**

Project task **68**
Solution **68**
Future Work **70**

Appendix A Performance Measures: Closed System Model

A.1 Invocation time performance 75

Sequencer 75
Skeen 76
Chandra Toueg 76
Socket 77
RMI 77
CORBA 78
Message size 78
Exact measures 79

A.2 Algorithm throughputs 80

Sequencer 80
Skeen 80
Chandra Toueg 81
Socket 81
RMI 82
CORBA 82

Appendix B Performance Measures: Client Server Model

B.1 Case A 83

Socket 83
RMI 84
Corba 84

B.2 Case B 85

Socket 85
RMI 86
CORBA 86

Appendix C Example and Usage

C.1 Interfaces 87

C.2 Server 88

C.3 Client 89

C.4 Compilation 91

Acknowledgements

I would like to thank the following people:

My parents - without your help this wouldn't have been possible.

My supervisor Dr. Rachid Guerraoui,

Mr. Romain Boichat for his help for checking the results,

and all the staff at the LSE for contributing their suggestions.

1 Introduction

The subject of this diploma project is to develop an application for atomic broadcast. We will make a comparison between several solutions, working on different middlewares, using different algorithms. The comparison will mainly be based on performance measures, varying different parameters, like the number of servers or the message length. This report will show the way we choose to provide objective conclusions.

This project has studied the implementation of Atomic Broadcast protocols on different middlewares. The chapters of this report show the theoretical and practical steps taken to achieve the desired results.

Chapter 1, first of all and for better understanding of the following chapters, the concept of distributed systems is introduced.

Chapter 2 presents the different atomic broadcast algorithms that have been used.

Chapter 3 describes the basic models of the used middlewares.

In *Chapter 4* the project implementation is presented.

Chapter 5 discusses the performance results measured with the implementation.

and *Chapter 6* finally makes a conclusion on the work of this project.

Additional information and complete source code can be found at:

http://in3www.epfl.ch/~nhirt

1.1. Distributed systems

The basic problem arises from the need of a highly available service (at disposal at every moment). If an error occurs, there has to be a mechanism hiding this from the user, keeping the service functional. So it is important to understand the meaning of service failures and how to avoid them.

A faulty service has a behaviour that is not wanted by the user. In order to avoid this, there are several ways preventing it from this undesired behaviour, by transforming it to be either predictable or masked.

This leads to a system that is **fault tolerant**.

There are various possibilities for implementing a fault tolerant service. This work considers only the case of service replication. It will be explained in the following paragraphs.

Some notions that could need further explanation will be presented like the following example. We choose this way in order to avoid interrupting the flow of the explanations with information that is possibly known yet.

Server : A server is a computer, which contains information on a certain subject. It can be seen as an information storage tank and is normally connected to a network.

Client : A client is also a computer, connected to a network in order to access the information stored on a server. The client does all the interaction with the user and thus is usually more complex than a server.

Accounts Example

Consider the example of a bank cash distributor that has to be operating 24 hours a day:
Typically, this distributor is connected to a server containing the users information, like the balance of his account or his withdrawal limit. The distributor has to be able to access the service at any moment to send his requests, so the transactions can be done.

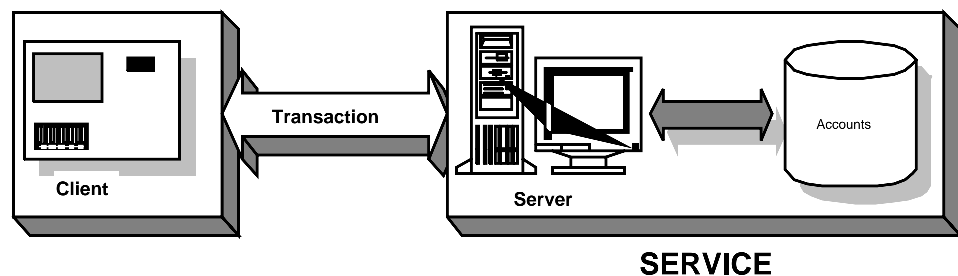


FIGURE 1. **Client/Server structure**

Now, in case the server fails and is not operable at the moment of the transaction, the user cannot make his transaction and the operability isn't guaranteed. Or even worse, if the server loses all his account information, the client will loose (at least temporarily) all his money, because there are no account records left.

As presented before, one possibility to prevent this annoying situation is to replicate the service. Resulting in what is called a ***fault tolerant distributed system***. This means, that all the information is replicated and distributed to several servers, able to provide exactly the same service. So if one server fails, there are others to accomplish the task.
The main goal is a complete transparency for the client. That is, that the client does see the duplicated servers as one system and that he hasn't to deal with the internal representation of the service.

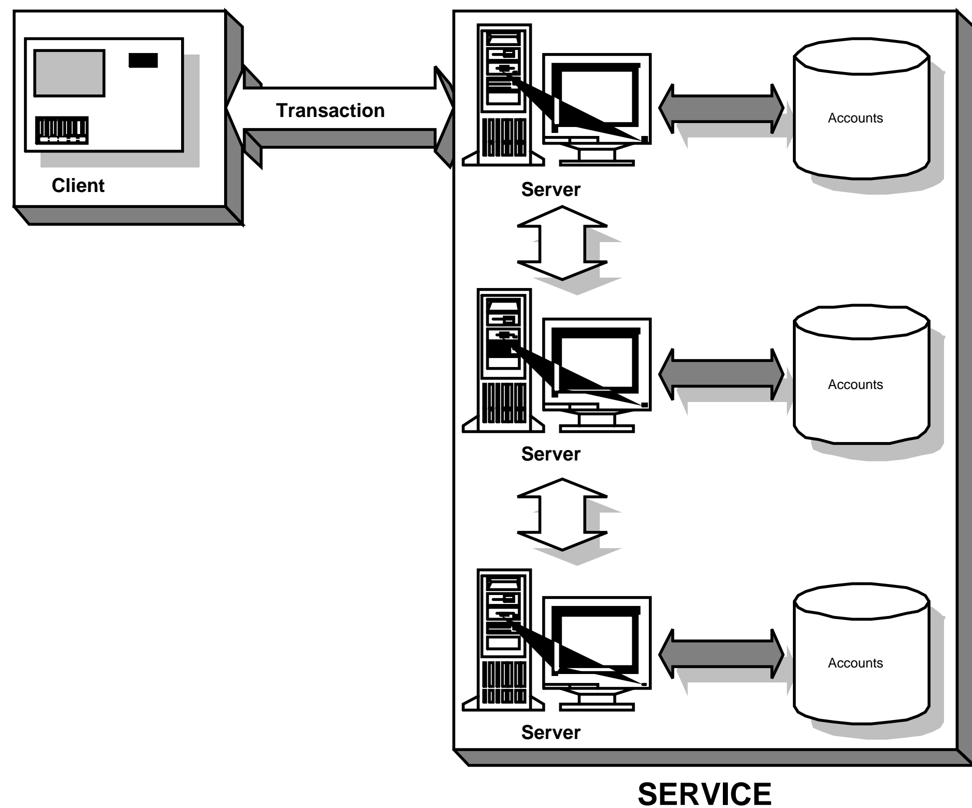


FIGURE 2. **Client/duplicated Server structure**

Basic structure

The basic structure of a distributed system is of a Client/Server type, where the Servers are duplicated to provide best fault tolerance.

- As mentioned before, the Server usually contains the records and thus is a storage centre for all the data and information. He equally treats the requests coming from the Clients and sends back the results.
- The Client usually doesn't contain any records and does just send requests for specific operations (transactions, withdrawal, etc...) to the Server.

It is very important, that the client side does not have to deal with the different servers; the service has to be perfectly transparent to the client. So the service must be represented as one entity and all administration tasks of the distributed system have to be done inside the service.

Formal representation

To study the behaviour of such systems, we have to introduce several notions.

- Every member of the system is considered as a *process*. These members form a *group*.
- Every process can *communicate* with every other process.
- A process can have an erroneous behaviour[1].
- The processes are not synchronized which leads to an asynchronous system.

Process : A process can be seen as a program running on a computer independently from its environment. Several of these processes can be executing simultaneously on the same computer.

Group : In our case, group means all the processes connected together with the network.

Asynchronous system: there are no time limits for communications between two different processes of the system.

1. a process can fail (e.g. stop executing or loose data).

We also have to introduce a graphical representation:

- The processes are represented by their identifier $p_1...p_n$ and the horizontal line indicates the lifetime of the process.
- The passing of a message is represented by an arrow from the source to the destination, and the identifier of the message $m_1...m_n$.

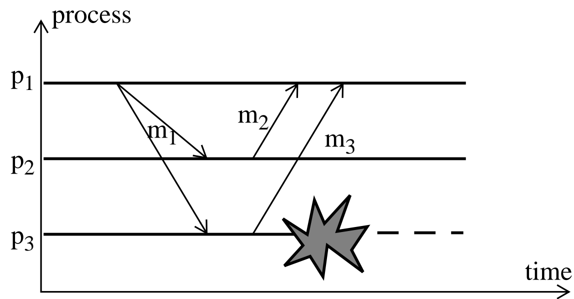
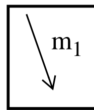
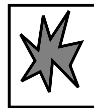


FIGURE 3. Example of inter-process communication



Process sending a message m1



Crash of a process

In Figure 3, the process p_1 sends a message m_1 to p_2 and p_3.
Then p_1 receives the messages m_2 (sent by p_2) and m_3 (sent by p_3).
Process p_3 crashes after having sent m_3.

1.2. Total order and atomicity

The goal here, is to guarantee that the information contained on the different servers is coherent (usually means identical).

To assure this we have to define two communication primitives:

TO-Cast(m,DST(m)) : sending the message m to the processes contained in DST(m) and
TO-Deliver(m): delivers the message m to the destination process

TO-Cast(m,DST(m)) is defined by three proprieties:

TO1. Uniform Agreement. Consider TO-Cast(m,DST(m)). If a process in DST(m) (correct or not) has TO-Delivered(m), then every correct process in DST(m) eventually TO-Delivers(m).

TO2. Termination. If a correct process TO-Casts(m,DST(m)), then every correct process in DST(m) eventually TO-Delivers(m).

TO3. Uniform Total Order. Let m_1 and m_2 be two messages that are TO-Cast. We note $m_1 < m_2$ if and only if a process (correct or not) TO-Delivers m_1 before m_2.

So if we use TO-Cast for sending our requests, the coherence of the copies will be respected.

Accounts Example

If again we take the example of a bank with a replicated bank account, with a current balance of 100$:

First we deposit 300$ then we withdraw 400$.
Finally we should arrive at a balance of 0$.
Figure 4 shows that the left server executed the transactions as desired. The right one did change the order of the requests which leads to an erroneous result and a fatal incoherence between the two servers. So the execution **order** of the requests must be the same for all receiving servers.

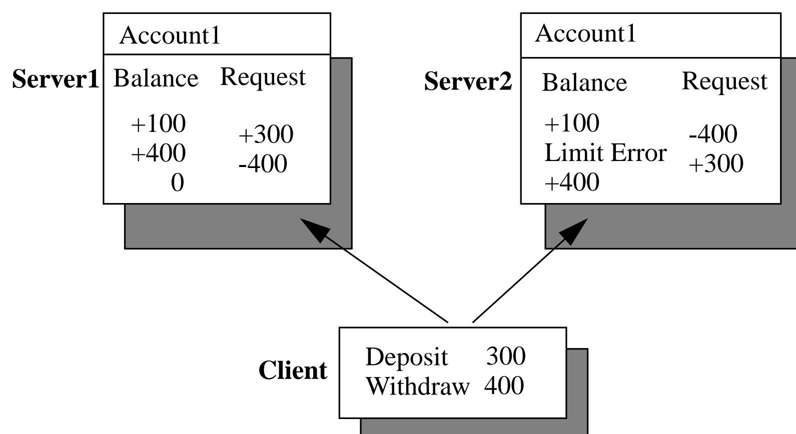


FIGURE 4. **Total order**

The other important part is the **atomicity** propriety. It says, that if one server executes the request then every other server has to do the same.

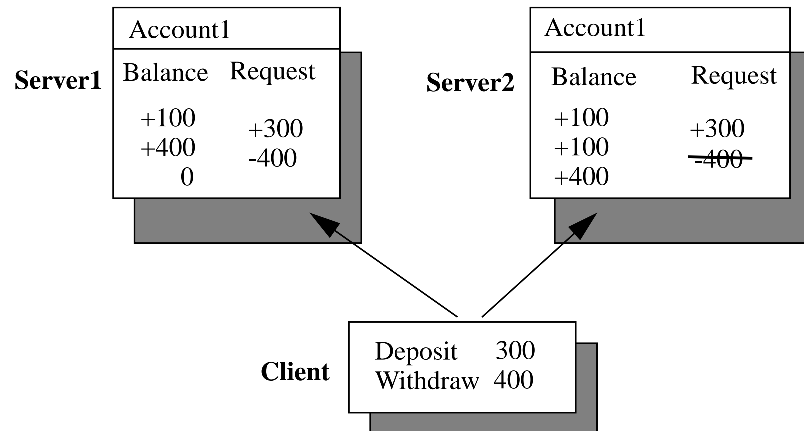


FIGURE 5. **Atomicity**

The atomic broadcast protocols will have to respect these proprieties (order and atomicity), in order to guarantee the coherence between the servers and thus a correct comportment.

2

Total Order

The task of all algorithms presented here, is to guarantee Total Order as shown in chapter 1. All of them need different conditions to present an error free execution. We will show the differences between them and give a first theoretical performance estimation.

Consider a **closed system** of processes, where each process can act as a sender or as a destination.
There is no difference between client and server processes. So no external client is considered and transmissions do not leave the system.

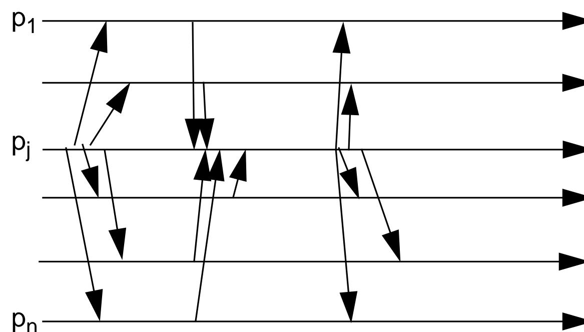


FIGURE 6. **Example of a closed system of processes**

Reliable Broadcast

To ensure that all the three proprieties (TO1-TO3) are respected, first a Reliable Broadcast method is needed, introducing two functions, ***R-Broadcast(m)*** and ***R-Deliver(m)***, which are defined by the following two proprieties:

Atomicity. If a correct process R-broadcasts(m), then every correct process eventually R-delivers(m).

Termination. If a correct process R-delivers(m), then every correct process eventually R-delivers(m).

So Atomicity and Termination are assured using the R-broadcast function. Combining this with the following total order algorithms, will provide all three proprieties for TO-cast.

This project is based on the assumption that we have **reliable channels**, so *reliable broadcast is replaced by a simple broadcast*.

2.1. Sequencer

The Sequencer algorithm presents the most simple solution to our problem.

It assumes a failure free system (processes aren't allowed to crash). To assure the proprieties of atomicity, one process is designed to be the sequencer process (p_q in this case). When it receives a message m, the sequencer sends it to all the other processes of the group including the message sequence number. The receiving processes then TO-Deliver the message according to their sequence number.

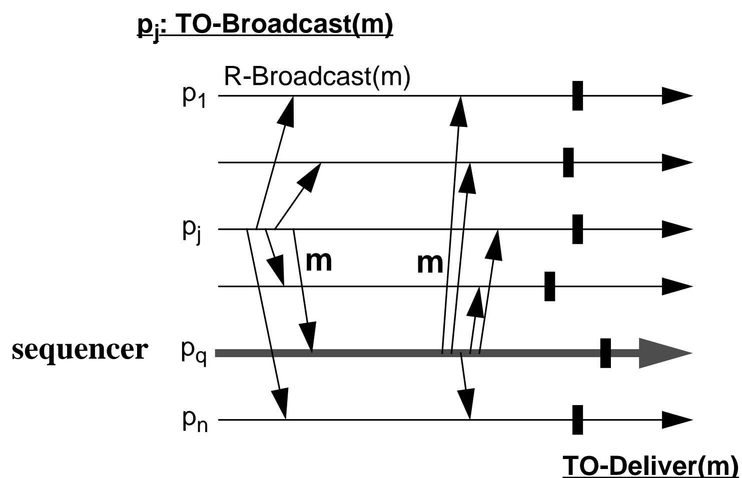


FIGURE 7. **Sequencer algorithm**

So it is the sequencing process p_q, which determines the order and thus assures the Total Order propriety. Atomicity is guaranteed by the conditions of having reliable channels and that no crashes are admitted.

The main disadvantage of this algorithm is the use of a fixed sequencer process. All requests have to be rediffused, by this designated process. Because of that, system contamination may occur. This means, that when meeting a high throughput situation, the sequencer process can become a serious bottleneck.

In case of a crash of the sequencer, another process has to be chosen to take this role, and the request has to be retransmitted.
If it's not the sequencer process which failed, then nothing has to be done.

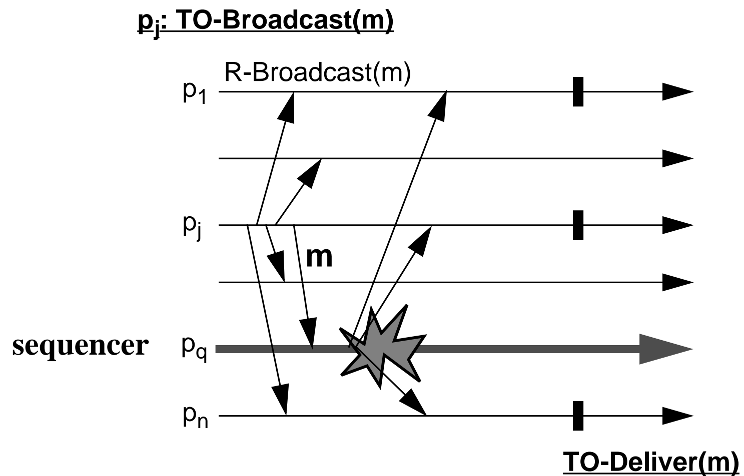


FIGURE 8. **Sequencer algorithm failure**

As we can see in this example the Total Order propriety is never violated (it is assured by the sequencer). But Atomicity can be violated, if the sequencer process crashes while re-broadcasting the message (Not all of the processes TO-Deliver m: Figure 8).

For practical implementation purposes, **only the message identification Id(m) will be broadcasted** by the sequencer, thus to improve the performance of this algorithm for long messages.

Fault-tolerance could be obtained, if the failure of the sequencer is reliably detected. This would mean to add an algorithm implementing a reliable failure detector, which can not be part of this project.

2.2. Skeen

The algorithm of Skeen is more powerful than the sequencer algorithm. Still it remains simple, because it is based on strong assumptions, like the need of a failure free environment.

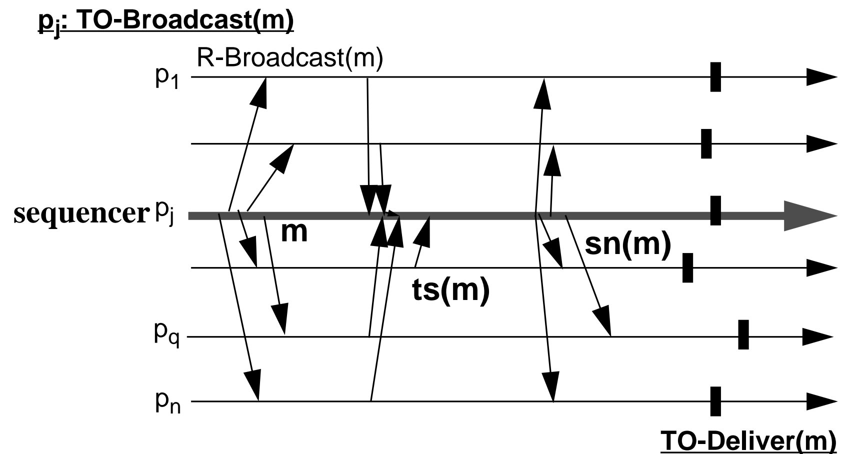


FIGURE 9. **Skeen's algorithm**

The sender process will be referred to as **sequencer**.
This algorithm uses Lamport's clocks to determine the logical time of the occurrence of an event[1].

The protocol assures that all processes decide on a unique timestamp assigned to a message. Thus to be sure to deliver the messages in the same order and so to respect Total Order. In Skeen's algorithm this timestamp (which is called sequence number sn) is determined by the sequencer process p_j.

- When a process receives a message it sends the timestamps of the reception of m (which we call ts(receive(m))) to the sequencer and stores the message with its timestamp in a pending buffer.
- The sequencer defines the sequence number sn(m) as the maximum of all the timestamps received and sends back sn(m) to all processes.
- When a process receives sn(m), the message m is moved to a delivery buffer (ordered according to the associated sequence number)
- A message m at the head of the delivery buffer can be delivered if and only if there is no message m' in the pending buffer such that ts(receive(m'))<sn(m). This assures that in the future the sequencer cannot decide on a sequence number that is smaller than the sequence number of the message we'd like to deliver.

Lamport's Clocks : Lamport's clocks give the logical time of an event in a chain of events. So their temporal order can be established.

1. In our case, an event would be a send or a receive.

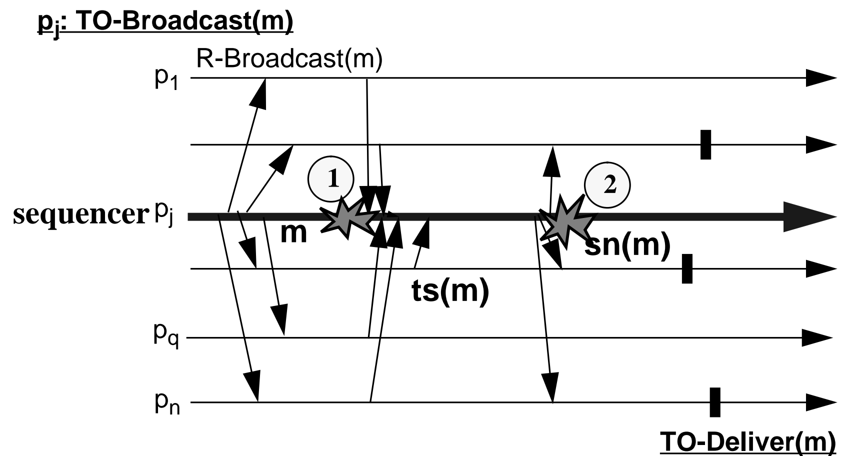


FIGURE 10. **Skeen's algorithm failure**

If the sequencer crashes at position 1, there will be no problem, because no message will be delivered (Total Order and Atomicity are not violated). The message will be delivered after the next TO-Broadcast

If the sequencer crashes at position 2, there will be a violation of Atomicity and Total Order, because not all of the processes deliver the message.
From where originates the need for a failure free environment.

As for the Sequencer algorithm, a fault-tolerant implementation of this algorithm needs a perfect failure detector in order to determine if the sequencer is correct.

2.3. Chandra and Toueg

The algorithm of Chandra and Toueg [11] is a failure tolerant extension of the Sequencer algorithm. The difference lies in the method the sequence number is determined. For this purpose we use a consensus algorithm [7], presented in the next paragraph.

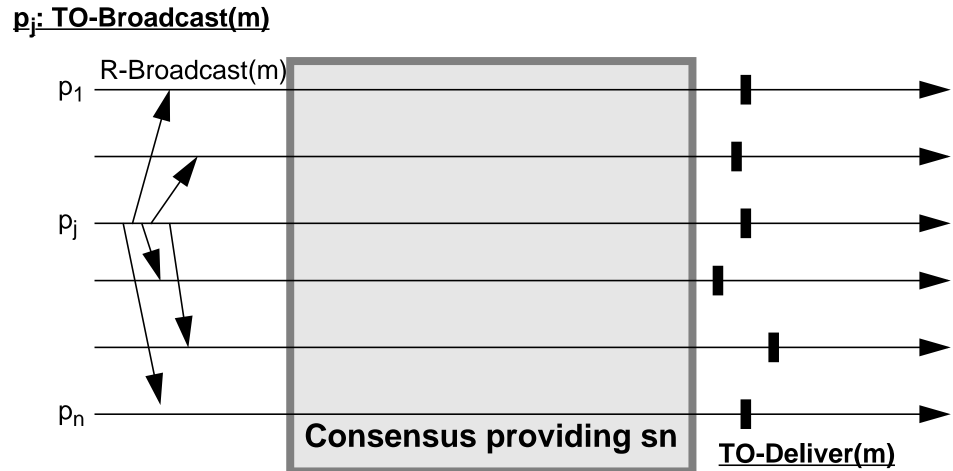


FIGURE 11. **Chandra and Touegs Algorithm**

Every process having R-Delivered(m) starts a consensus with the timestamp of R-Delivery(m) as initial value. The consensus decision provides the sequence number sn(m). Finally the messages are delivered according to their sequence number.

Consensus

Despite the FLP Impossibility result[1], Chandra and Toueg presented a possibility to solve the consensus problem. They proposed an algorithm that finds a solution to the problem in a non deterministic number of rounds:

Consider n processes each of them proposing a value v_i. The processes have to decide on one and the same value.

Specification:

- ***agreement***: no two processes decide differently
- ***termination***: every correct process eventually decides
- ***non-triviality***: the value decided is one of the values proposed

The consensus algorithm guarantees that all processes decide on one and the same value.

1. See [8]+ for details

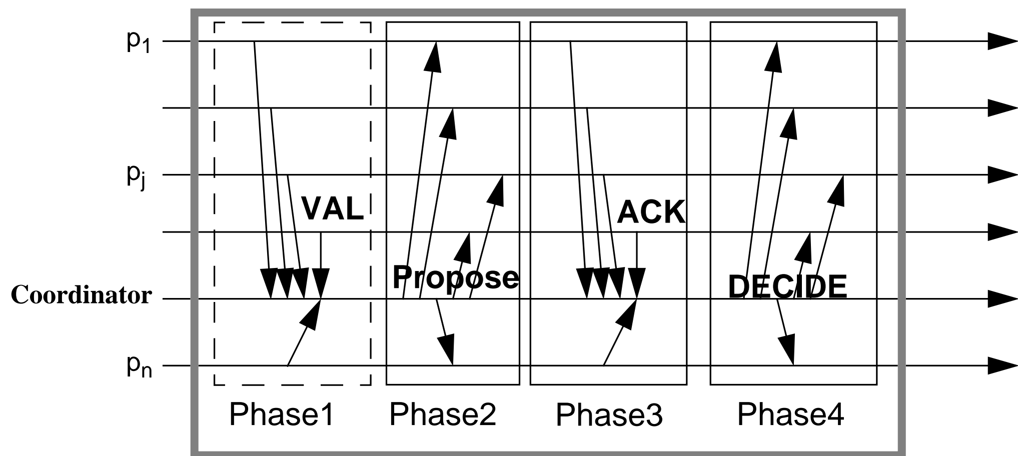


FIGURE 12. **Consensus**

The different phases are:

- *Phase 1*: All processes propose a value to the coordinator.
- *Phase 2*: The current coordinator gathers n/2+1 values and proposes a new estimate.
- *Phase 3*: All processes wait for the new value proposed by the current coordinator. If there is no problem with the new value, they send an acknowledge message (ACK). If the value is not delivered or the coordinator is supposed to have crashed, then a NACK response is sent.
- *Phase 4*: The current coordinator waits for n/2+1 replies. If these replies indicate, that n-f processes adopted its estimate (ACK), the coordinator sends a request to decide. Otherwise, a new round is started with a different coordinator.

In practical use we can omit the first step and let the coordinator directly propose a value.

The consensus is strongly based on the use of failure detectors[5],[7]. Failure detectors make it possible to suspect a process of having failed, and thus let a process behave differently when failures are suspected. To be operational in an environment which admits process failures, the consensus algorithm needs failure detectors.
As this project is considering only failure free runs, ***failure detectors will not be used.***

2.4. Theoretical performance

For theoretical performance estimation, the three algorithms are compared by analysing their costs in terms of communication steps needed, and the number of messages exchanged.

All algorithms start with a R-Broadcast(m)/R-Deliver(m), and thus this step can be found in all of them. As we replace the reliable broadcast by a normal broadcast, the cost of the number of messages transmitted will be much lower. In fact a reliable broadcast transmits $(n-1)^2$ messages (where n is the number of processes), whereas a normal broadcast transmits only (n-1). So there is a great gain of performance when using many processes.

• Sequencer

The sequencer algorithm first reliably broadcasts the message, followed by the broadcast of the message containing the sequence number. So the algorithm requires 2 communication steps. There are (n-1) messages for the reliable broadcast (replaced by the normal broadcast), and again (n-1) messages for broadcasting the sequence numbers.

• Skeen

Skeen's algorithm takes one step for gathering the timestamps and one for broadcasting the sequence number. So it takes a total of 3 steps. In terms of messages exchanged, each step transfers (n-1) messages, leading to a total of 3(n-1) messages exchanged.

• Chandra Toueg

Finally Chandra and Touegs algorithm takes 4 communication steps: one for reliable broadcast and 3 for the consensus (as the first step has been omitted), resulting in 4 communication steps. Finally there are 4(n-1) messages transmitted.

The following table shows a summary of the theoretical results.

Algorithm	Nb. of Steps	Messages transmitted
Sequencer	2	2 (n-1)
Skeen	3	3 (n-1)
Chandra Toueg	4	4 (n-1)

TABLE 1. Theoretical algorithm performance

In the following chapters will be presented other basic elements for theoretical performance prediction.

3

Middleware

In this chapter we present the three different communication techniques that have been used. Their basic proprieties are showed as well as their special features.

3.1. Sockets

In Internet transmission, the data is stored in packets of fixed size, called datagrams. They consist of a header, containing mainly the address and the port number of the sender and the destination. In addition to this it contains a checksum and packet sequence number in order to offer reliable transmission. The data itself can be found in the second part of the datagram. As datagrams are limited in size, data has often to be cut in several datagram packets, the destination can join in the order given by the packet sequence number. If a packet gets lost a demand for retransmission of the lost packet has to be made.

Fortunately there is the concept of Sockets which is a innovation of the Berkeley UNIX, implementing all the above protocols and methods. The goal was to use the network as a standard UNIX IO Stream. The user can read and write as if he would access a file on a harddisk.

The possibilities for Sockets are the following:

- connect to a distant machine.
- send data
- receive data
- close a connection

To handle the connection phase, the concept of ServerSocket has been introduced. This is a Socket attached to a port, waiting for a connection request. When a distant machine sends its request, it can be rejected or accepted, in which case the connection is established. So there are another two possibilities:

- wait for a connection request (listening) on a specified port
- accept a connection request and create a Socket instance for permanent connection with the remote Socket

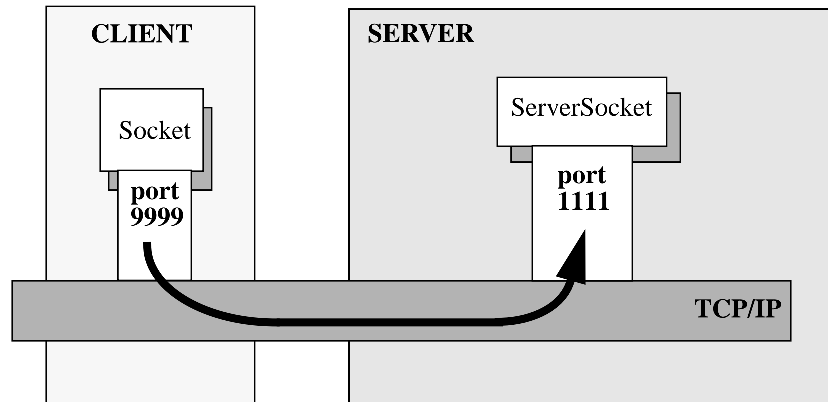


FIGURE 13. **ServerSocket listening and remote Socket trying to connect.**

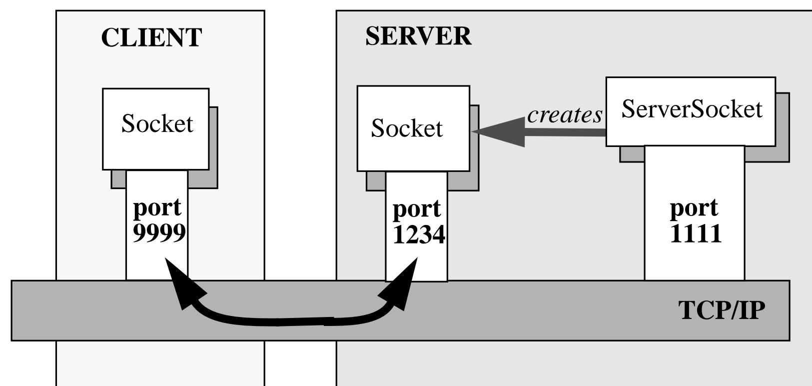


FIGURE 14. **ServerSocket creating local Socket for connection with client.**

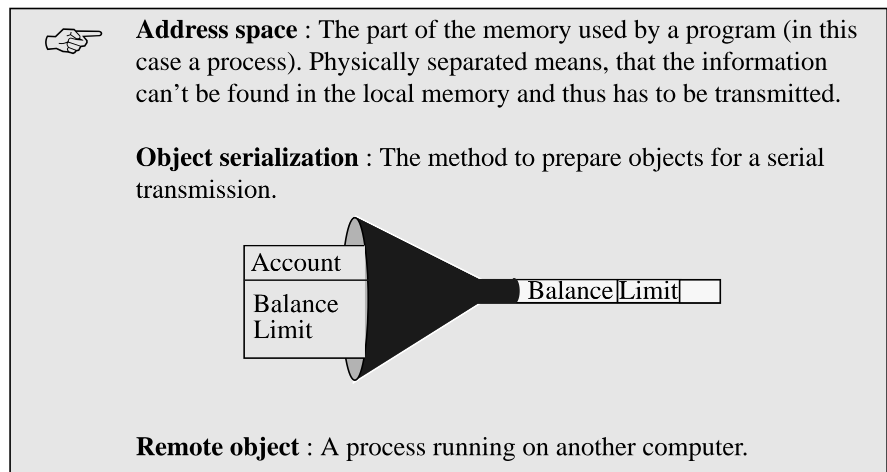
Sockets are a low level solution for network programmers as they don’t implement any protocol functions. This has to be done by the programmer. Still when used properly, Sockets offer very fast transmission, because the protocol used (and so the transmitted data) can be optimized for the respective application. Though a more sophisticated solution is presented in the next chapter.

3.2. RMI

RMI (Remote Method Invocation) is a powerful extension offered by the Java 1.1 standard. It provides the possibility to call methods on physically separated computers with different address spaces through a network. With RMI there is no need to develop special protocols to encode and decode the messages to pass. Especially if the messages are Java objects, RMI is very well adapted because it does not have to be language-neutral (as with CORBA).

The RMI System consists of three layers:

- The stub/skeleton layer - client-side stubs and server-side skeletons
- The remote reference layer - describes the remote reference behaviour
- The transport layer - connection set-up and management and remote object tracking



Address space : The part of the memory used by a program (in this case a process). Physically separated means, that the information can’t be found in the local memory and thus has to be transmitted.

Object serialization : The method to prepare objects for a serial transmission.

Remote object : A process running on another computer.

As we are only interested in the part that does the interfacing with the application, we will not describe the remote interface nor the transport layer.

The Stub/Skeleton layer transmits data (objects) by using marshal streams. Marshal streams employ object serialization to pass the objects between different address spaces. Normally the objects are passed by copy to the remote host.

- The client-side stub is mainly in charge of:
  Marshalling the invocation and to unmarshalling the return value or the exception.
- The server-side skeleton is in charge of:
  Unmarshalling the invocation, making the call to the remote object and marshalling the return value or the exception.

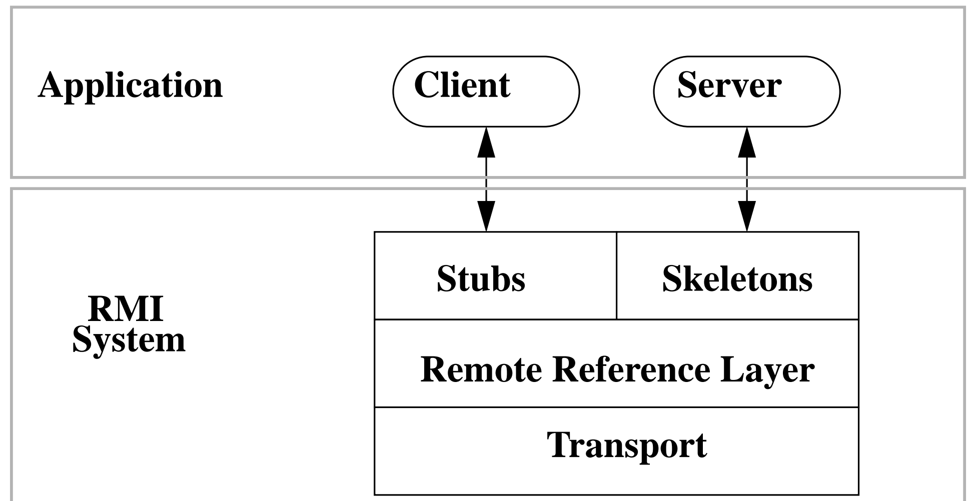


FIGURE 15. **Structure of the RMI System**

To allow the remote objects to know each other, there is a registry service, which maps names to the remote objects. A remote object can be bound and unbound in a registry server. The reference to a remote object is obtained by calling a lookup method passing its name as argument.

The normal way to proceed is the following:

- Step 1: The Remote Object (Server) registers at the registry service with its ID ("MY_Server").
- Step 2: A client wants to access the server (invoke a method) and does a lookup for the server's name at the registry service.
- Step 3: The registry service returns a reference to a local stub, with which the client can accomplish its invocation.

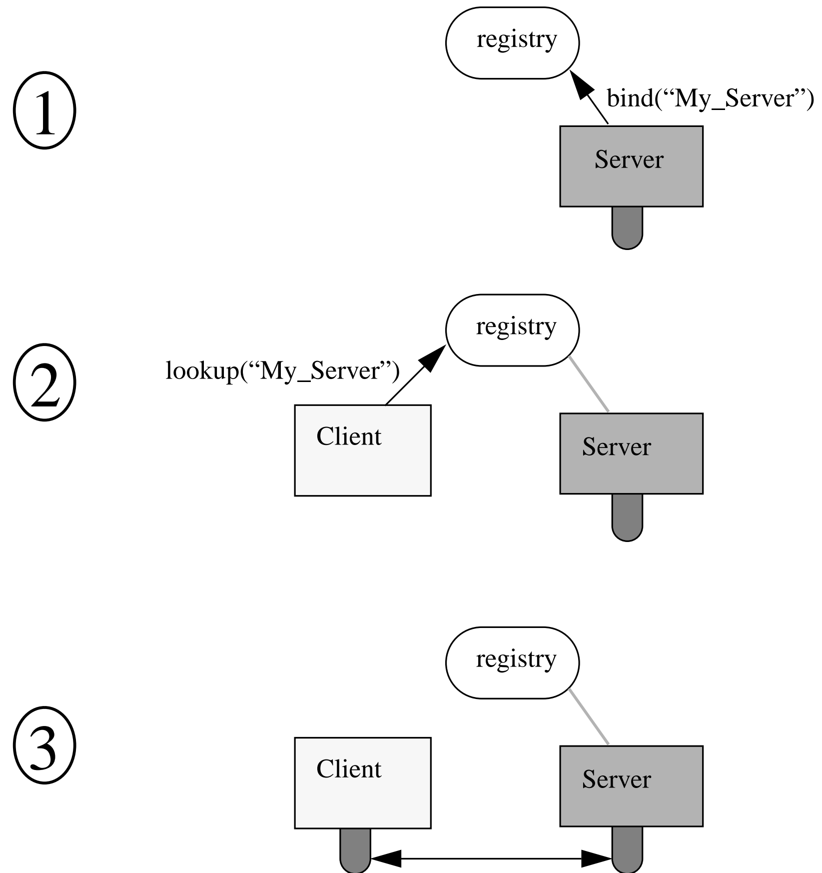


FIGURE 16. **Method invoking process**

3.3. CORBA

The Common Object Request Broker Architecture (CORBA) was specified by the Object Management Group (OMG) to reduce the complexity and the costs of developing distributed system architectures. CORBA is a purely object-oriented solution, treating objects as encapsulated entities and providing an interface, giving access in a user-defined way. Once an object is working properly it can be reused over and over again.

For this project we decided to use VisiBroker Object Request Broker (ORB) for Java, as it is the most complete Java implementation of CORBA at the moment.
The ORB provides the possibility of connecting a client without knowing the servers location. The only things the client needs to know, are the name of the server and the use of its interface. The ORB then takes care of all the underneath operations.

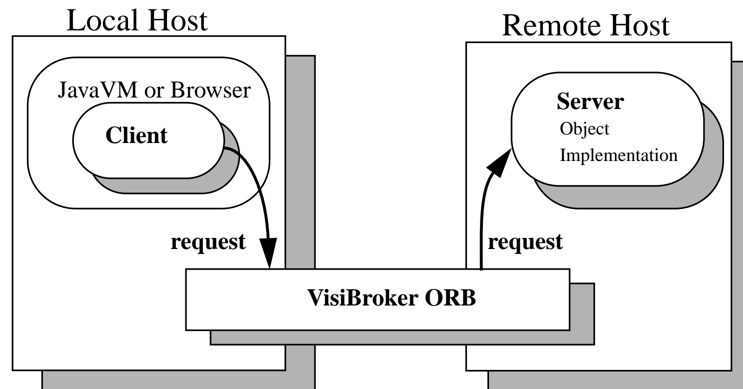


FIGURE 17. **Request invocation through the ORB**

The Visibroker ORB offers several features, which we will not all cover in detail:

Interface Repository

The IR (Interface Repository) contains information about the ORB objects types, mainly the interfaces and the exceptions.

Dynamic Invocation Interface

A client can obtain information about an object´s interface from the IR and from that dynamically construct requests for the object.

Smart Binding

VisiBroker automatically chooses the optimum transport mechanism:

- If the object is local, it performs a local method call
- If it is a remote object IIOP is used.

Smart Agents

Smart Agents are an extension to the CORBA specifications. They can reconnect a client to a server if the latter has been unavailable. Furthermore they allow a client to launch a server process on demand.

Location Service

Combined with the Smart Agent, the Location Service provides an easy to use solution to locate available instances of an object in a network.

But the main feature of the VisiBroker CORBA implementation is the independency on programming languages. There is no problem using a Java client invoking requests on a C++ server. This makes it easy to use the appropriate language for different implementation purposes.

For more information refer to [9].

4 Implementation

As presented before, there are several possible algorithms and middlewares that can be used for the implementation. This chapter will give an overview of the choices made and the general structure of the implementation.

To assure the conditions of Total Order, consider the use of the following three algorithms, already presented in chapter 2:

- Sequencer
- Skeen
- Chandra and Touegs Algorithm

For communication purposes there are:

- Sockets
- RMI[1]
- VisiBroker CORBA[2] implementation

Those possibilities are implemented to be able to compare performance and behaviour of the different solutions.

	Sequencer	**Skeen**	**Chandra Toueg**
Sockets	✓	✓	✓
RMI	✓	✓	✓
CORBA	✓	✓	✓

1. RMI is the Remote Method Invocation package of Java 1.1
2. CORBA is Common Object Request Broker Architecture

In this chapter is shown the implementation of the different communication methods and algorithms and also the test environment is defined.

But first some basic ideas for structuring the approach.

4.1. Basic concepts

The structure is based on a three layer model (as seen in Figure 18) to provide an independency between the different layer implementations. So the goal is to provide a uniform interface between the layers, completely concealing the underneath layer structure.

All this will make it much easier to accomplish the planned performance measures, because they won't be biased by differing program parts.

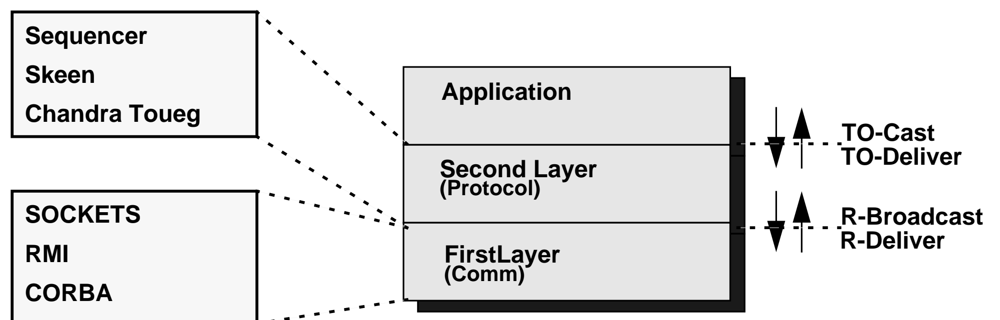


FIGURE 18. Three layer structure

Communication from one Application to another passes all the layers as shown in Figure 19.
To be able to refer to a certain destination, each process will be identified by a proper ID number. As we will see later it is assigned by the registration server.

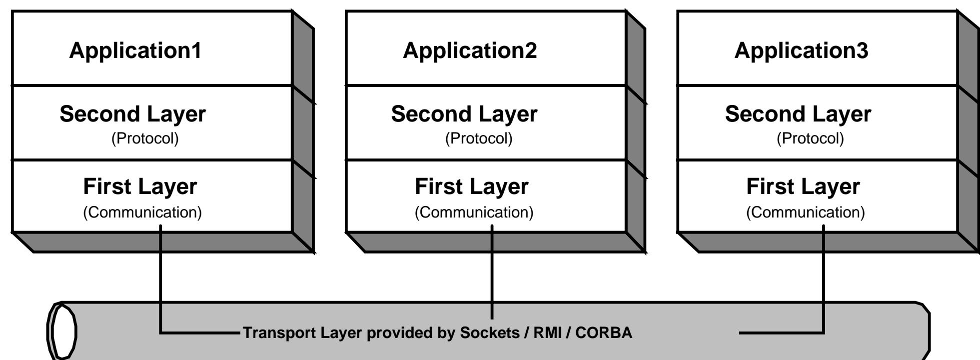


FIGURE 19. Layer Structure

In the following paragraphs, will be presented the object and interface models of the implemented classes, as well as the description of the class interfaces. To provide a uniform representation, we choose a notation heavily based on the Fusion method described in [10]. Unhappily this method has not been made for multi thread modelisation, but the slight modifications should be easy to understand.

4.2. First layer

The first layer handles the ***communication*** between the different processes, using the three different communication models. The following presents the classes used for the FirstLayer implementation.

Sockets

• class FirstLayer

Provides the basic communication methods.

Variables

Socket[] **groupSockets**	array of all the sockets established at connection time.
DataInputStream[] **datain**	array of all the input streams.
DataOutputStream[] **dataout**	array of all the output streams.
int **myID**	the dynamically assigned ID number of the process.
int **serverNum**	number of processes connected.
int **Time**	logical Lamport process clock.
String **REG_SERVER**	address of the registry server

Methods

FirstLayer(String regserver)	Class constructor: regserver is the address of the registry server.
int connect()	connects the class to a registry server and returns the dynamically assigned ID number.
void send(String msg, int dst)	sends the string msg to the destination process with ID dst.
String receive(int receiveType)	reads the next message of the indicated type. Blocking if none is available.
void RBroadcast(String msg)	sends the string msg to all processes fixing the type to REQUEST.

String RDeliver()	reads the next message of type REQUEST.
void setTime(int Time)	sets the logical clock to the value Time.
int getTime()	returns the value of the logical clock.
int getServerNum()	returns the number of connected processes.
void emptyBuffers(int timestamp)	deletes all messages with a timestamp lower than timestamp from the buffers.

• class PollInput : Runnable

This class is started by the FirstLayer instance as a Thread. The run() method is basically an infinite loop reading the **datain** input streams and dispatching the messages to the appropriate buffer (see “Transmission” on page 50).

Methods

PollInput(FirstLayer _FirstLayer)	Class constructor getting a reference to the FirstLayer instance in order to write directly to the buffers.
void run()	run method of the Runnable class polling the **datain** stream.

• class MessageList

Message List implements a FIFO message queue, using instances of the Message class to construct a chained list.
The FirstLayer class instantiates several buffers of type MessageList to buffer the input stream coming from the PollInput thread.

Variables

Message **begin**	pointer to the head of the buffer.
Message **end**	pointer to the end of the buffer.
int **elementNum**	number of messages in the queue.

Methods

MessageList()	Class constructor
void appendBuffer(String msg, int TS)	appends the message to the buffer, with msg being the message body and TS the timestamp of reception.

String removeMsg()	destructive read of the message at the head of the buffer.
int getMsgNum()	returns the number of messages in the buffer.
void removeTSMessage(int TS)	removes all messages with a timestamp smaller than TS.

• class Message

This class represents one message.

Variables

String **Msg**	the body of the message.
int **timestamp**	the timestamp of the message reception.

Methods

Message(String msg, int TS)	Class constructor, creating one Message object containing the message body and the timestamp TS.

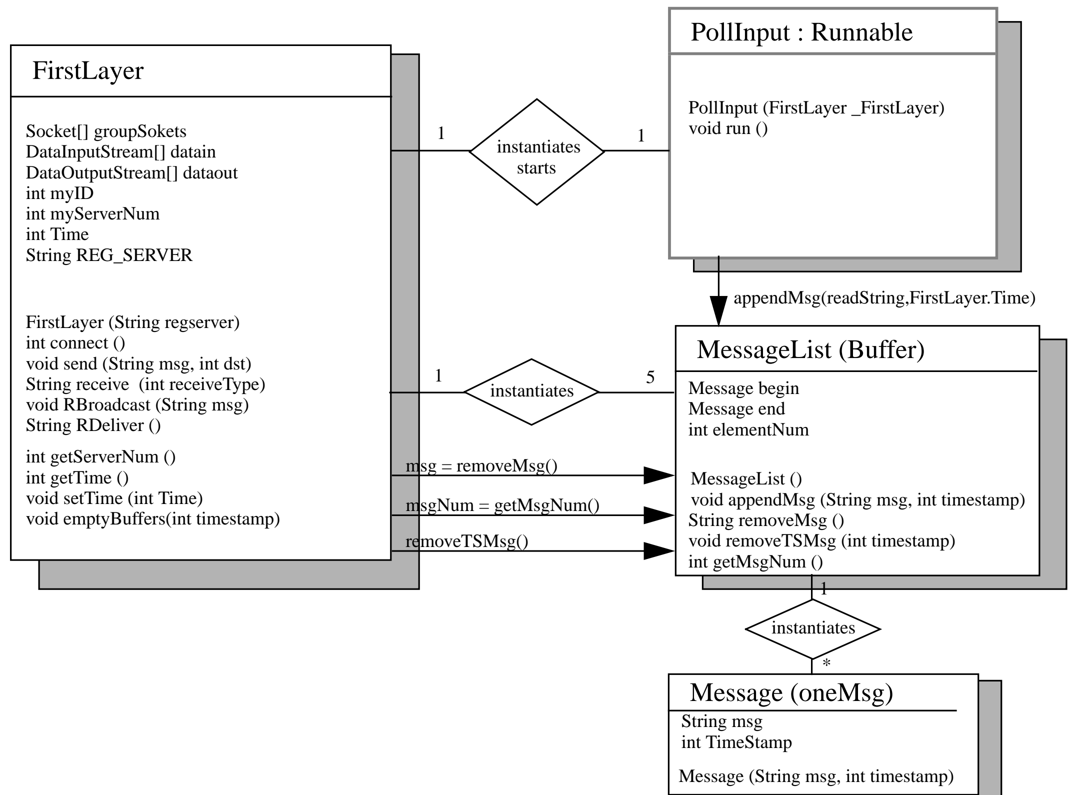


FIGURE 20. Socket object and interaction model

RMI

• class FirstLayer : UnicastRemoteObject

Provides the basic communication methods.

Variables

int **myID**	the dynamically assigned ID number of the process.
int **serverNum**	number of processes connected together.
int **Time**	logical Lamport process clock.
String **REG_SERVER**	address of the registry server.
String **BIND_STRING**	represents the name of the bound process.
FirstRMIInterface[] **remoteObjectReference**	array containing all references to remote objects, to speed up transmission.

Methods

FirstLayer(String regserver)	Class constructor: regserver is the address of the registry server.
int connect()	connects the class to a registry server and returns the dynamically assigned ID number.
void send(String msg, int dst)	sends the string msg to the destination process with ID dst.
String receive(int receiveType)	reads the next message of the indicated type. Blocking if none is available.
void RBroadcast(String msg)	sends the string msg to all processes fixing the type to REQUEST.
String RDeliver()	reads the next message of type REQUEST.
void appendBuffer(String msg)	implementation of the remotely invocable method defined in the **FirstRMIInterface** class.
void setTime(int Time)	sets the logical clock to the value Time.
int getTime()	returns the value of the logical clock.
int getServerNum()	returns the number of connected processes.

void emptyBuffers(int timestamp) deletes all messages with a timestamp lower than timestamp from the buffers.

• class FirstRMIInterface : java.rmi.Remote

Defines the RMI interface for the FirstLayer class and thus the methods that can be remotely invoked.

Methods

void appendBuffer(String msg) appends the message msg to the appropriate buffer. It is used by the send and RBroadcast methods of the sender process, calling this method directly on the destination process.

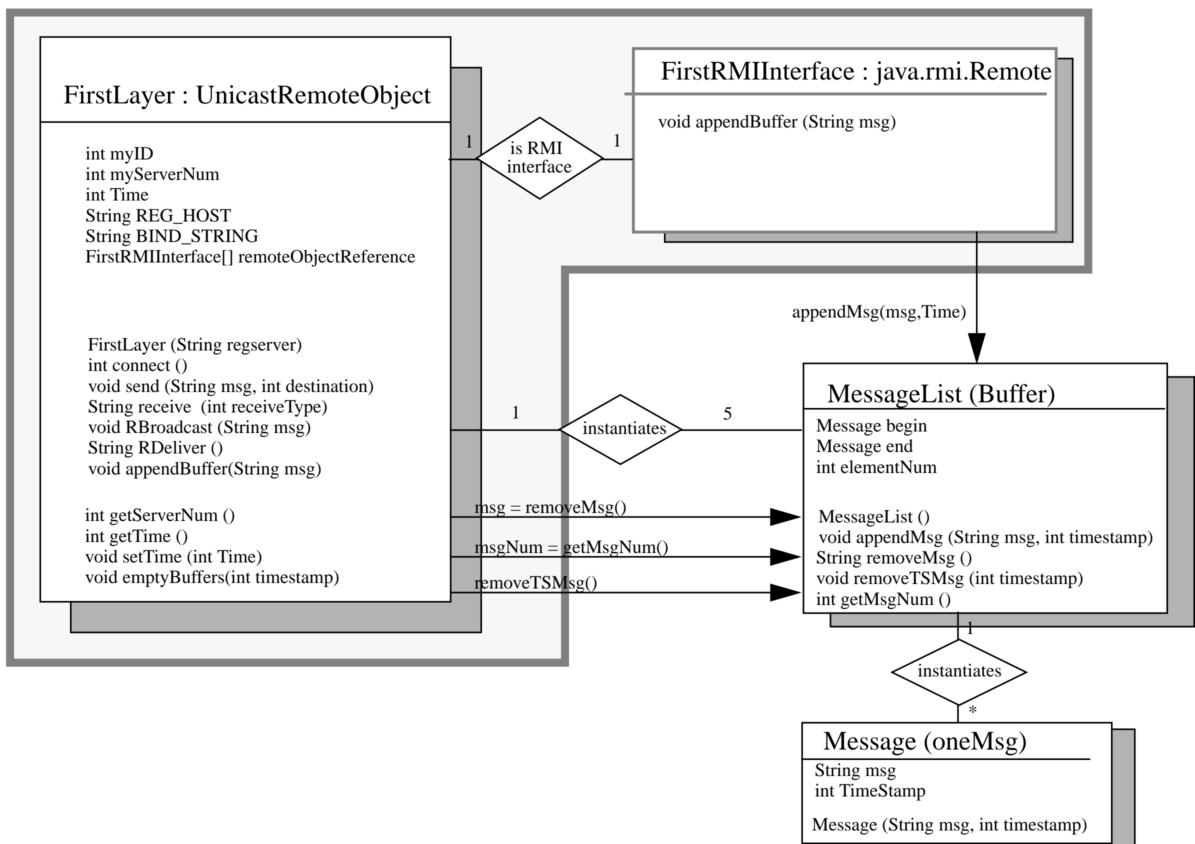


FIGURE 21. **RMI object and interaction model**

CORBA

• class FirstLayer

Provides the basic communication methods.

Variables

int **myID**	the dynamically assigned ID number of the process.
int **serverNum**	number of processes connected together.
int **Time**	logical Lamport process clock.
String **REG_SERVER**	address of the registry server.
String **BIND_STRING**	represents the name of the bound process.
CorbaFirstModule[] **remoteObjectReference**	array, containing all references on remote objects, to speed up transmission.

Methods

FirstLayer(String regserver)	Class constructor: regserver is the address of the registry server.
int connect()	connects the class to a registry server and returns the dynamically assigned ID number.
void send(String msg, int dst)	sends the string msg to the destination process with ID dst.
String receive(int receiveType)	reads the next message of the indicated type. Blocking if none is available.
void RBroadcast(String msg)	sends the string msg to all processes fixing the type to REQUEST.
String RDeliver()	reads the next message of type REQUEST.
void appendBuffer(String msg)	appends the message msg to the appropriate buffer.
void setTime(int Time)	sets the logical clock to the value Time.
int getTime()	returns the value of the logical clock.

int getServerNum() returns the number of connected processes.

void emptyBuffers(int timestamp) deletes all messages with a timestamp lower than timestamp from the buffers.

• class FirstCorbaImpl : CorbaFirstModule._FirstLayerInterfaceImplBase

Represents the implementation of the remotely invocable method.

Methods

FirstCorbaImpl(String name, FirstLayer _FirstLayer) Class constructor for the remotely invocable methods. The name represents the identification String under which the object will be registered.

void appendBuffer(String msg) implementation of the remotely invocable method defined in the **FirstCorbaInterface** IDL. Redirects the call to the same method of the FirstLayer class.

void bound() signals the **FirstLayer** that the object is properly bound.

• class CorbaServer : Runnable

Class started as thread from the FirstLayer instance to register the FirstCorbaImpl object.This class is necessary, because the impl_is_ready() call would block further execution.

Methods

CorbaServer(FirstLayerImpl _FirstLayerImpl) Class constructor for the registering thread.

void run() run method of the Runnable class registering the FirstLayerImpl instance.

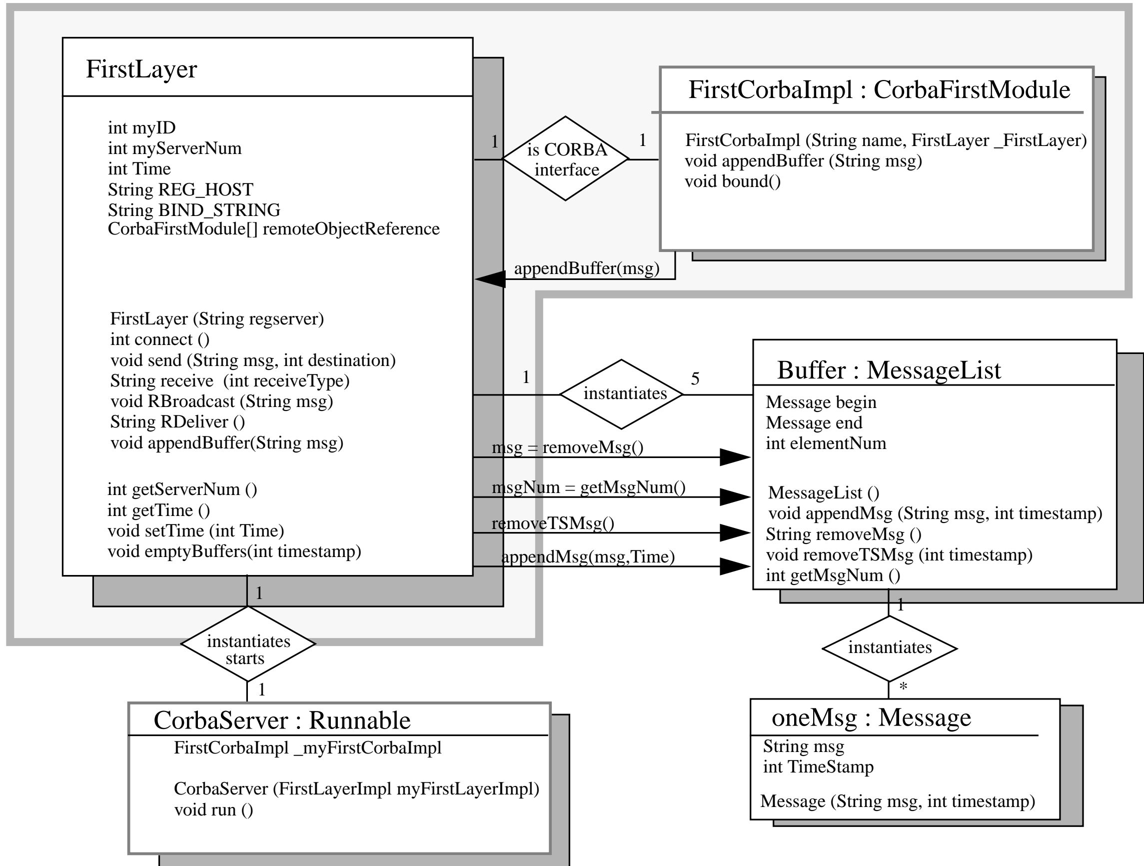


FIGURE 22. CORBA object and interaction model

4.3. Second layer

The second layer implements the different ***algorithms*** providing the Total Order propriety. Thus it has to implement a protocol based on the functions provided by the First Layer.

• class SecondLayer

This class provides the basic methods for Total Order communication.

Variables

int **myID**	the dynamically assigned ID number of the process.
int **serverNum**	number of processes connected together.
int **myAlgorithm**	indicates the chosen algorithm.
FirstLayer **myFirstLayer**	instance of the FirstLayer used.

Methods

SecondLayer(int algorithm, String regserver)	Class constructor: algorithm represents the algorithm used and regserver is the address of the registry server. Creates an instance of the FirstLayer class.
int connect()	calls the connect method of the FirstLayer instance.
void TOBroadcast(String msg)	TO Broadcasts the string msg to all processes.
String TODeliver()	TO Delivers the next message.
int getServerNum()	returns the number of connected processes.

• class Consensus : Runnable

This class provides the modified consensus algorithm by Chandra and Toueg. See chapter 2 for more detailed information.

Variables

int **round**	round of the actual consensus.
String **DECIDED**	message accepted by all processes.
FirstLayer **myFirstLayer**	instance of the FirstLayer used.

Methods

Consensus (FirstLayer _FirstLayer, String propMsg, int myID, int server Num)

Class constructor: where propMsg is the message proposed by the process.

void run() — run method of the Runnable class initiating the consensus.

String getDecided() — returns the result of the consensus. Blocking until a result is available.

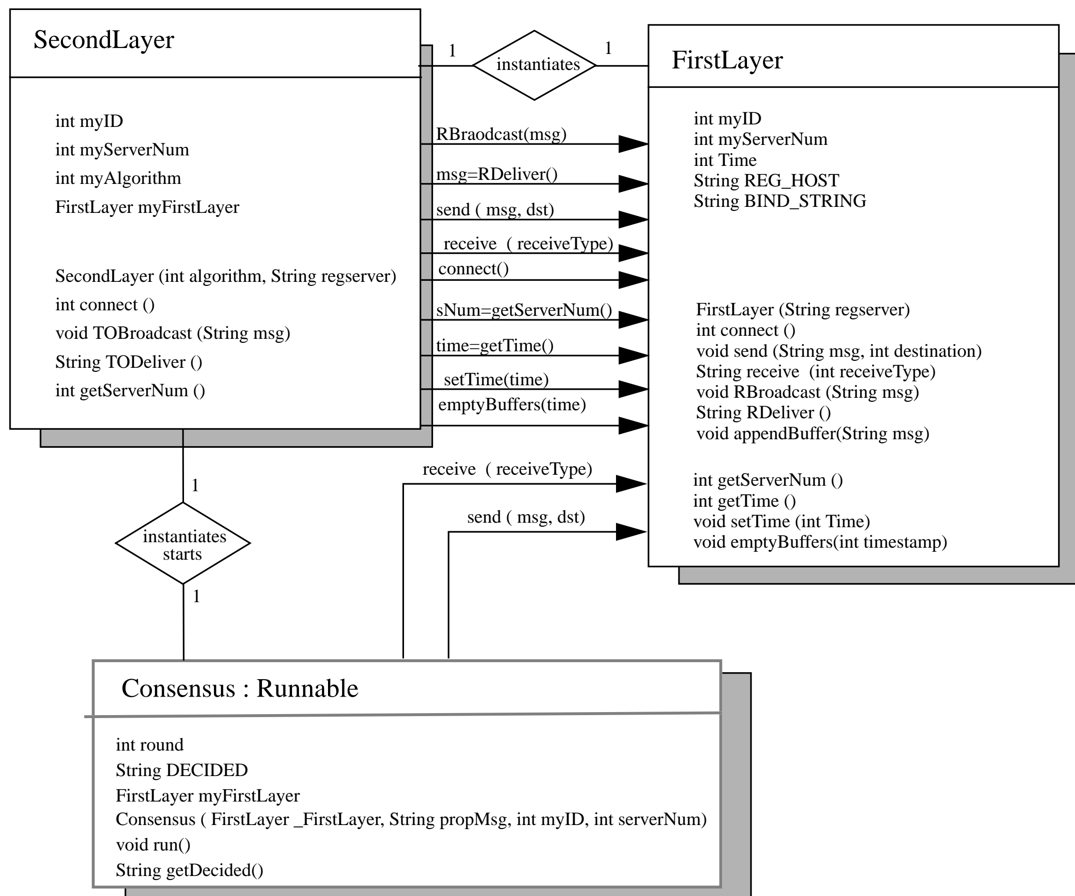


FIGURE 23. **Second Layer object and interaction model**

As it can be seen in Figure 23, the consensus (for the Chandra Toueg algorithm) is started as an independent Thread, so that it doesn't prevent the message reception handling in the First Layer. This is because the methods (mainly receive) in the First Layer class cannot be invoked by RMI or CORBA while the algorithm is waiting for a message to arrive (i.e. acknowledge), which would lead to a deadlock situation.

Application

The application finally uses the methods of the SecondLayer class to handle the exchange of information between Clients and Servers.

Normally there would be a Client using the TO-Cast method to send a request to all Servers. The message will be automatically dispatched to all the Servers which will finally TO-Deliver it.

For our purpose we choose to create an application acting as both, a Client and as a Server. For each round, there is one application taking the Client role, TO-Casting a message. All processes then act as Servers and try to TO-Deliver the message.
Like this we can be sure that the average execution time does not depend on the relative speed of the machine where the process runs.

A script will distribute the applications on different machines where they register to the registry server and wait for all other servers to be connected.

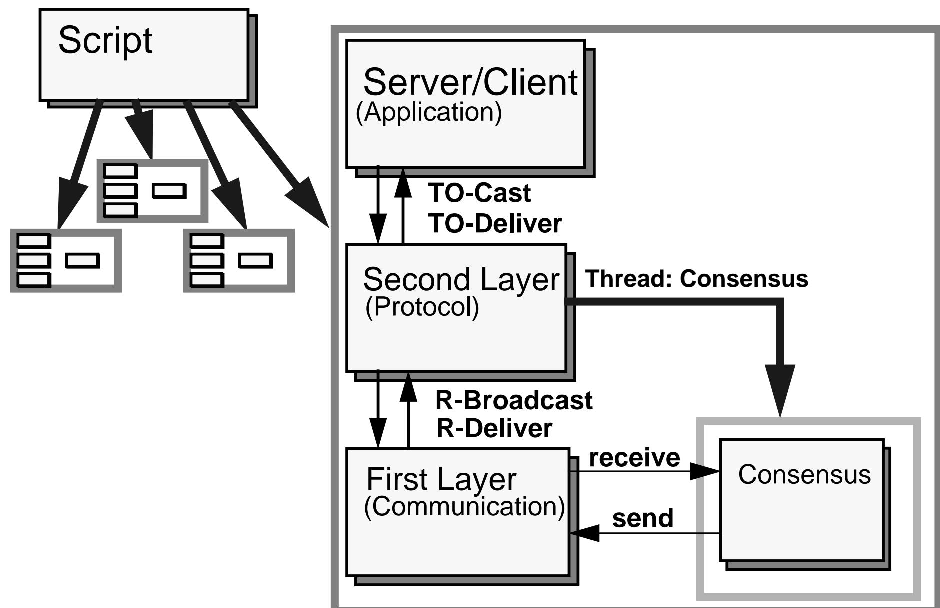


FIGURE 24. **Interfaces between the layers**

The following example shows how the performance measures could have been done. More detailed explanations and examples can be found in chapter 5.

```
method Test(){
    int round = 0;

    myId=connect();
    myServerNum=getServerNum();

    while(true){
        if (myID=round % myServerNum){

            TOBroadcast(msg);      ]  Time1

        }
        else{

            msg=TODeliver();       ]  Time2

        }

        round++;
    }
}
```

FIGURE 25. **Example of performance measuring method**

The final performance measures will have to consider a mean value of Time1 and Time2. It is very important to be aware what time is measured in fact. If the times are just added and then finally divided by the number of rounds, the result represents a very practical value. In that case it is a real-life performance estimation one could meet as a everyday user.
If only one of the two method calls is considered, the result either represents the maximum output of requests the Client can do (in the case of Time1) or an estimation of the request processing speed of the Servers (in the case of Time2).

4.4. Registration

To offer an easy to use solution, we choose to provide registry based communications. So there has to be a registry service where the service providing servers have to register their name and address.

Basically there would be no need to implement a registry service with the Sockets solution. But as RMI and CORBA use this type of connection creation, the same is used for Sockets, in order to keep the aimed transparency.

The following shows an example of two servers doing their registration with the registration server. Then a client wants to connect to the Server “MyServer2” and calls the service to know the address. Finally, after having received the destination address, the Client is able to send the request to the desired server.

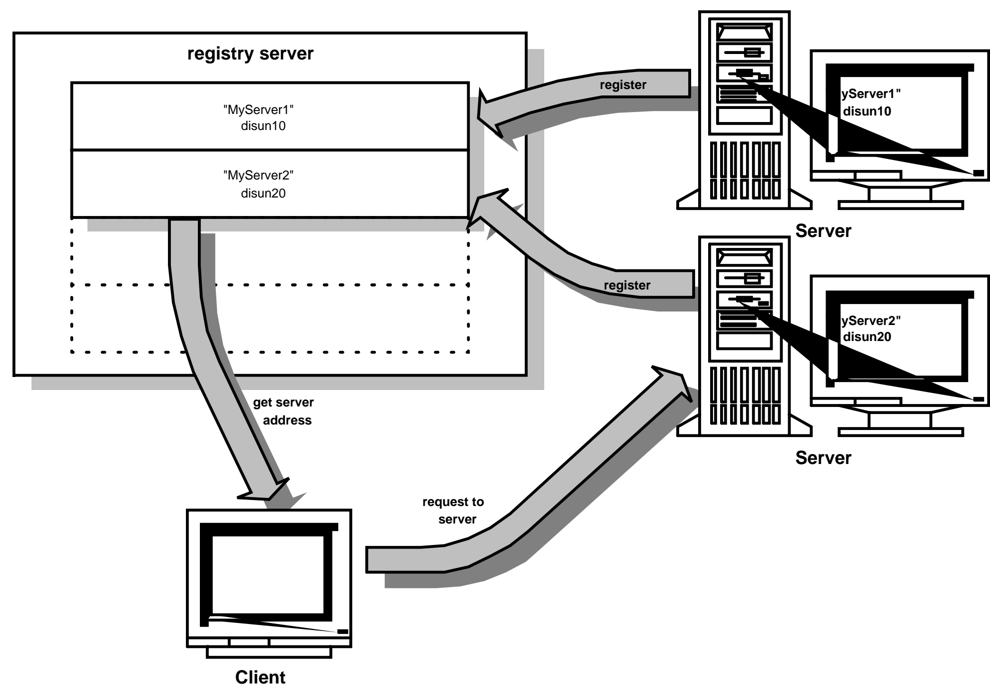


FIGURE 26. Registration process

As presented before, each process has its own and unique identification number. They are assigned dynamically by the registry server, accomplishing the work to find the next unused ID number. To do so, the registry needs to know the number of servers that will be connected.

The IDs are assigned following the temporal connection order.

The servers have to remain connected, because there is no possibility to transfer the state of one process to another. So a server can’t disconnect or get connected after beginning of transactions, because the coherence would be violated.

Server 1	Server 2	Server 3	Server 4	Server 5
4	5	1	3	2

FIGURE 27. **Dynamic ID distribution scheme**

As shown in Figure 27, the registry service had been started for 5 servers, and the ID numbers are transmitted at registration time.

Socket

The registry server for Sockets uses the following protocol to register the calling processes:

• class SockReg

This class provides the registry server for Sockets.

Variables

int **registryPort**	the port number the ServerSocket is connected to.

Methods

void main (String args[])

- Creation of a ServerSocket on a fixed port.
- Listening for connection attempts from other processes.
- When a process tries to connect, a connection is established.
- When all processes are connected, then the respective process ID and the total number of processes is sent back.
- Finally the addresses of all connected processes are broadcasted.

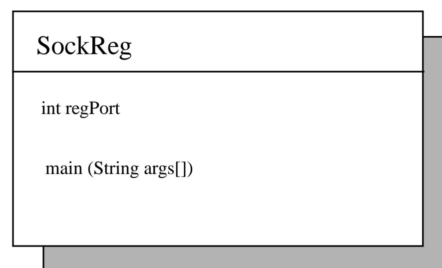


FIGURE 28. **SockReg object and interaction model**

RMI

The RMI version of the registry server is founded on the rmiregistry program, which comes with the RMI package. It makes it possible that the registering function is directly called on the registry server process.

• class RMIReg : UnicastRemoteObject

This class provides the registry server for RMI.

Methods

RMIReg(int serverNum)	Class constructor: serverNum is the number of processes, which will be bound.
int getServerNum()	returns the number of connected processes.
String getBindString()	returns the name under which the processes are registered.
int RegisterMe(java.rmi.Remote obj)	registers the calling process, returning its ID number.
void main(String args[])	main procedure to be executed at program start.

• class RMIRegServer : java.rmi.Remote

Defines the RMI interface for the RMIReg class and thus the methods that can be remotely invoked.

Methods

int getServerNum()	returns the number of connected processes.
String getBindString()	returns the name under which the process is registered.
int RegisterMe(java.rmi.Remote obj)	registers the calling process, returning its ID number.

For RMI and CORBA we choose to apply the following naming convention when registering the processes:

BIND_STRING + myID

For process 1 this would result (if the variable BIND_STRING contains RMI_BOUND_PROCESS) in “RMI_BOUND_PROCESS1”.

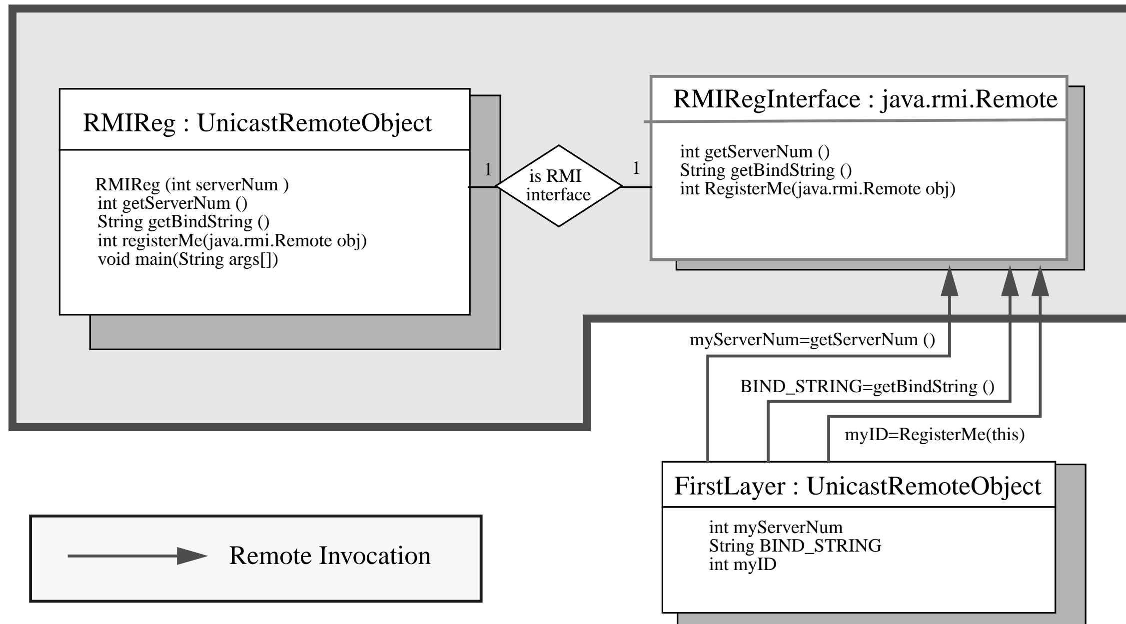


FIGURE 29. **RMIReg object and interaction model**

Corba

The CORBA version of the registry server is founded on the osagent program, which comes with the VisiBroker package. As with the RMI solution, it is possible that the registering function is directly called on the registry server process.

• class CorbaReg : CorbaRegModule._CorbaRegInterfaceImplBase

This class provides the registry server for CORBA.

Methods

CorbaReg(String name, int serverNum)	Class constructor: serverNum is the number of processes, which will be bound and name represents the identification String under which the process is registered.
int getServerNum()	returns the number of connected processes.
String getBindString()	returns the name under which the process is registered.
int RegisterMe()	registers the calling process, returning its ID number.
void main(String args[])	main procedure to be executed at program start.

• class CorbaRegServer : Runnable

Class started as thread from the CorbaReg instance to register the CorbaReg object.This class is necessary, because the impl_is_ready() call would block further execution.

Methods

CorbaRegServer(CorbaReg _myCorbaReg)	Class constructor for the registering thread.
void run()	run method of the Runnable class registering the FirstLayerImpl instance.

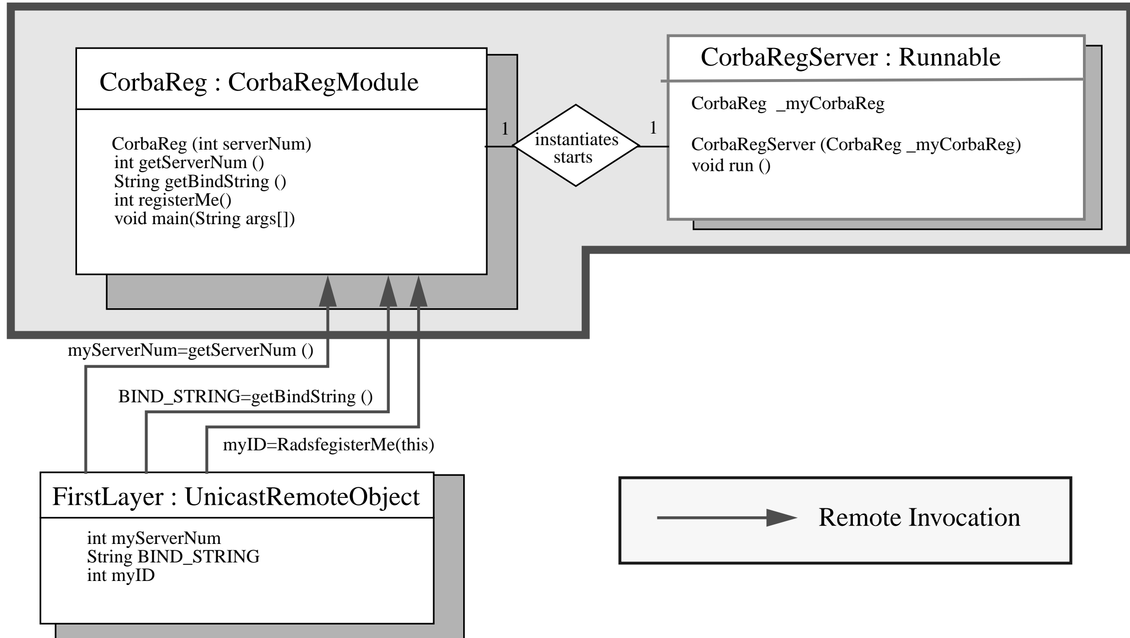


FIGURE 30. CorbaReg object and interaction model

4.5. Communication

Sockets

• Connection

In order to be able to communicate directly with each other process, the FirstLayer needs to have a connection to every destination process and so we have to build a completely connected network[1], allowing a communication at every moment between the processes. With the help of the registration service, the addresses of all the servers that will be bound in the network can be obtained. Knowing them communications can be established between the servers, using the following method, connecting all the processes through independent TCP/IP channels:

Hyp.:
- We introduce the notion of *round.*
- The round indicating the number of the connecting phase (Figure 31 on page 50).
- Each process has its own unique ID number.
- Consider *N* processes.
- The array addressOfServer contains the addresses of all servers in the same order for all processes.

- Step 1: Each process from p_{round} +1 to p_N listens for one new connection.
- Step 2: The process p_{round} tries to connect to all the processes listening (from p_{round} +1 to N.)
- Step 3: If a listening process gets the connection it is inserted in a table containing all the Sockets.

```
method connectGroup(int serverNum, int Port){
    SS= new ServerSocket(Port+ID);

for (int round=0; round<ID+1; round++){
     if (myID==round){
          for (int i=round+1; i< serverNum; i++){
               groupSockets[i]= new Socket(addressOfServer[i],Port+i);
          }
     }
      else{
          Socket sock = SS.accept();
          groupSockets[round]=sock;
      }
    }
  }
```

1. *completely connected network*: all processes have seperate connections to each other process.

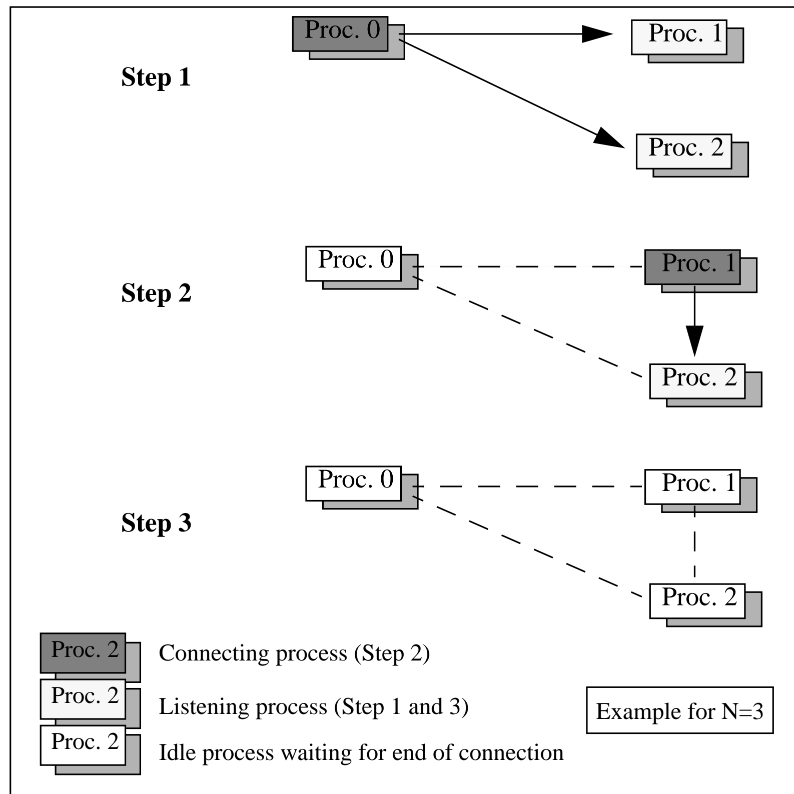


FIGURE 31. Creation of the network for 3 processes

- The advantage of this procedure is that it is the same for any number of processes.
- This method creates N! connections because the Socket connection is bidirectional. Unfortunately it is not possible to connect more than eight processes, because there seems to be a limitation on Socket connections, that can be open at a time.

• Transmission

Data transmission is done mainly by a protocol, which will now be described in detail.
When using Sockets one can not be sure to get the messages in the right order, because of the FIFO structure of the input queue. So when waiting for an Acknowledge message for m_1 it could happen that another message m_2 prevents its reception if the implementation does not use ***message buffering*** (Figure 32).

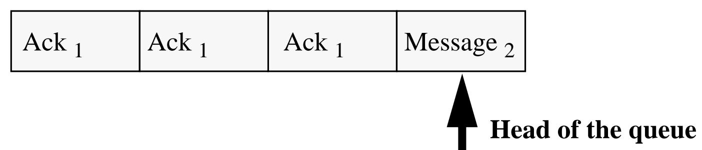


FIGURE 32. $Message_2$ preventing reception of Ack_1

In order to have direct access to different types of messages, there arises the need for message polling, introducing several buffers to avoid the situation mentioned before.

Buffers with Message Polling

Message polling is a basic procedure that reads all the messages waiting in the input buffer and dispatches them to different FIFO queues, according to their type. As shown in Figure 33, message polling will work as background task, to not slow down the main process.

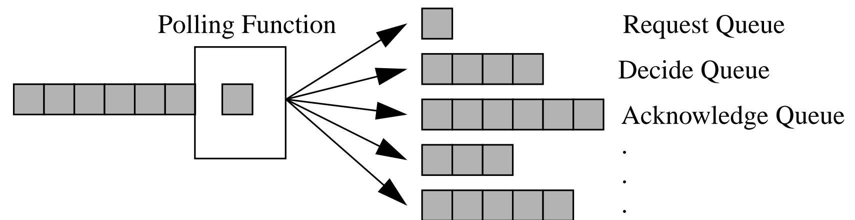


FIGURE 33. **Message polling**

Each message transmitted has the following format:

MessageType : Message Body : Timestamp : Sender

The ***Message Type*** is required, allowing us implement the solution using different queues.

The ***Message Body*** is only required when transporting data. It is not required when the message is of the Acknowledge or the Decide type.

The ***Timestamp*** is important so that the information in the queues can be handled. Messages that have a ***Timestamp*** below the time of the last TO-Delivered message can be removed from the queues.

The ***Sender*** is an information that might be used at some moment in the algorithms (e.g. to determine the primary process in the case of Skeen's algorithm).

Each time a message is read, its type is determined and the message is stored in the matching buffer.
A function that cleans up the buffers deleting all entries having a timestamp lower than the indicated value is provided.

If no message of the desired type is available the process repeats polling until the desired message has arrived.

RMI

• Connection

As for Sockets, the RMI connection phase passes by an additional registry server, based on the basic registry server supplied with Java 1.1.
As shown in Figure 29, the registering process has to create an object reference to the registry server and then calling the *RegisterMe* method. The registry server then registers the calling object with rmiregistry. The *getServerNum* method returns the value -1, until all servers are registered. Knowing the *BIND_STRING* (by calling the *getBindString* method) and the process numbering conventions, each process can establish the *remoteObjectReference* array containing all references to the registered remote objects. This, to speed up transmission, preventing from calling the registry at every transmission for getting the remote reference.

• Transmission

The basic idea of the RMI implementation is based on the fact, that methods of processes can be invoked remotely by other processes. This leads to the solution adopted: Each process has the same queues known from the Socket's message polling implementation. We introduce a method *appendBuffer(String msg)*, that can be called by any remote process with a message in the above format.
On the receiving side, the message is then directly dispatched to the according buffer.
So in this case a remote process can write directly to another process' input queue. This solution differs from Socket's message polling in the way, that the former has an active, request driven message reception, whereas the latter can be considered as sort of passive.

CORBA

• Connection

For CORBA, the additional registry is based on the supplied osagent registry. Nevertheless, the program code for RMI and CORBA can be considered almost identical. The only differences in the connection process can be found, where the differing registration method calls have to be respected.

• Transmission

The method chosen for the RMI implementation has been used for CORBA as well. We have also the same remotely invocable *appendBuffer(String msg)* method for writing directly into the buffer of the receiving process. The differences in this part are minimal, concerning only the remote method call.

This wanted similarity between the two implementations has been adopted, to prevent the performance measures from being biased by differing program parts.

5

Performance Test

The performance measures presented in this chapter, are using the different solutions showed before. The result will be compared with the theoretical estimations and possible differences are explained.

5.1. Test environment

All our measures have been made on Sun Sparcstations 20.
We used the internal network of the Swiss Federal Institute of Technology to dispatch the processes on locally different machines to eliminate **execution time variations**.
All tests have been run several times on varying day and night time to get an objective estimation, as independent as possible from **network charge**.
Every FirstLayer implementation has been tested with the same SecondLayer implementation and with the same test application.

As the measures are not deterministic and are characterized by small variations, the results will always be only approximated values!

5.2. Estimations

Before beginning with the real measures, it is important to recall the theoretical performance estimations found in the previous chapters.

Algorithms

For the different algorithms the following estimations have been made:

Algorithm	Nb. of Steps	Messages transmitted
Sequencer	2	2 (n-1)
Skeen	3	3 (n-1)
Chandra Toueg	4	4 (n-1)

TABLE 2. Theoretical algorithm performance

It seems clear, that the performance will be best for the sequencer algorithm. Skeen's algorithm should be slower by 50% and Chandra Toueg's algorithm about 100%.

Performance will be better with small messages, reducing the amount of data being transferred. This should introduce a linear factor in our measures, depending on the size of the message transmitted.

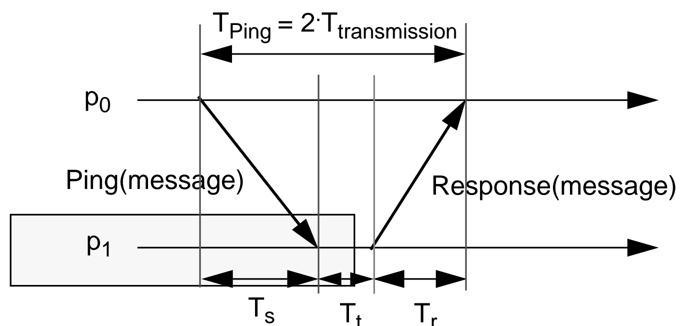
Middlewares

As presented in [1] RMI is usually 42% slower than CORBA. For a simple Ping invocation, the authors have found the following values:

Middleware	Ping Time
Socket	2.0 ms
RMI	5.5 ms
CORBA	3.3 ms

This shows, that CORBA is usually 40% slower than Sockets. These tests have been performed for a message size of 5 Bytes.

For measuring transmission time, a simple Ping protocol (where each transmission carries a message of a certain size) has been used.



The method which has been applied, consisted in keeping T_t neglectable by **avoiding buffer dispatching**. Then the sum of T_s and T_r is divided by 2. Supposing that T_s and T_r. are approximately equal, the result will present a correct estimation.

$$T_{transmission} = T_{Ping}$$

$$T_{transmission} = (T_s + T_r)/ 2$$

The following values have been measured for 5 byte messages:

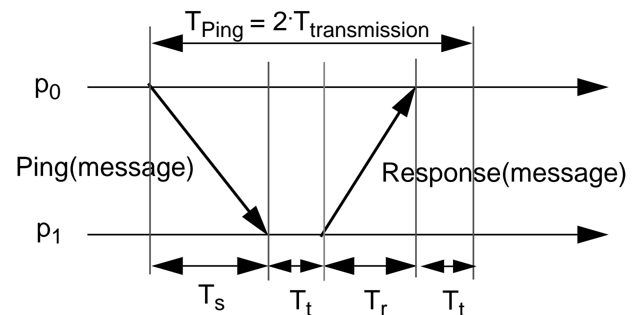
Middleware	Ping Time	$T_{transmission}$ (5 Bytes)	$T_{transmission}$ (100 Bytes)
Socket	2.7 ms	1.35 ms	1.5 ms
RMI	7.5 ms	3.75 ms	5.2 ms
CORBA	6.7 ms	3.35 ms	3.6 ms

TABLE 3. Measured performance

The exact values could not have been reproduced, as the present system configuration is very different from the one used in [1]. The value for the CORBA implementation using VisiBroker is way too high. This seems to be due to the system differences to the one mentioned above.

Further estimations will be based on these measured values, being representative for the configuration used for this project.

Using **buffer dispatching** introduces a non-neglectable time T_t (present for both ways), so the transmission time $T_{transmission}$ can be evaluated as with the above method, simply dividing the ping-time by 2.



The equations:

$$T_{transmission} = T_{Ping}$$

$$T_{transmission} = (Ts + Tr)/ 2$$

are still correct.

Until now, only fixed size messages have been considered. In the figure below the transmission time of a message with **variable size**, introducing buffer dispatching is represented.

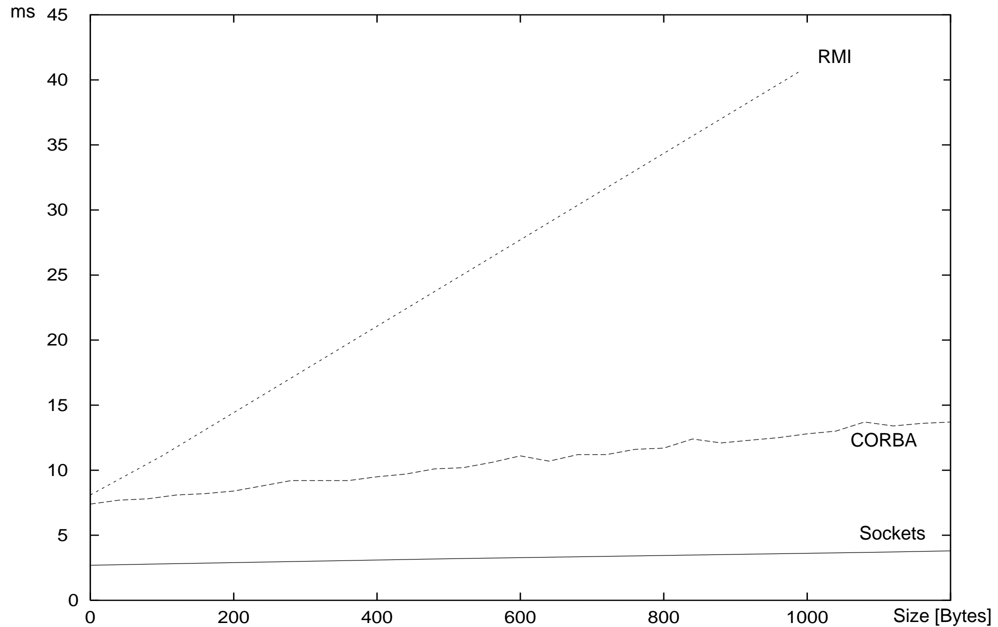


FIGURE 34. Transmission time for one message of variable size

Also the above measures are confirmed by this graphic for small message sizes. Only the RMI values for important message sizes grows very fast. This is due to the difference in object serialization:

- CORBA uses native methods which have been optimized for this purpose.
- RMI is entirely based on JAVA 1.1 and thus uses its own java.io.Serializable interface, which is very slow compared to the one used with the VisiBroker implementation.

The measures have been made with the layer structure of the project implementation and have been confirmed by separate measures made with the programs provided in [1].

Complete estimation

Now all the basic facts are known to perform execution time estimation of the TO-Broadcast implementations.
The formula to evaluate this would be:

$$T_{TO\text{-}Broadcast} = s \cdot (n-1) \cdot T_{transmission}$$

where:		
	s	number of steps taken by the algorithm
	n	number of processes
	$T_{transmission}$	Transmission time for one message (Figure 34)

This estimation would introduce a linear graph as the following, passing 0 for one process:

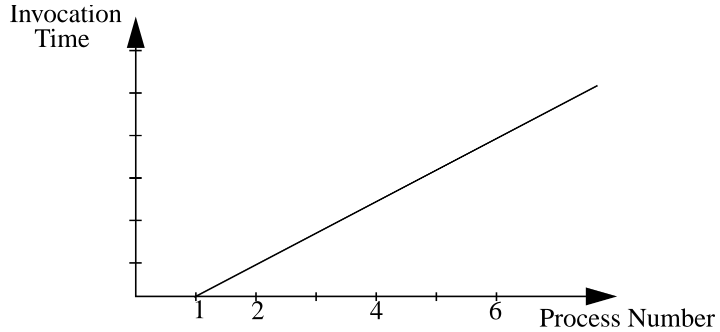


FIGURE 35. Invocation times

In reality this is not true, because the processes need a certain internal time to execute the algorithms. This time is called overhead and introduces an offset to the above graph.

1. A first possibility to determine this value, is to simply measure the execution time of the algorithm with one process:

Algorithm	Overhead
Sequencer	2.1 ms
Skeen	2.6 ms
Chandra Toueg	4.2 ms

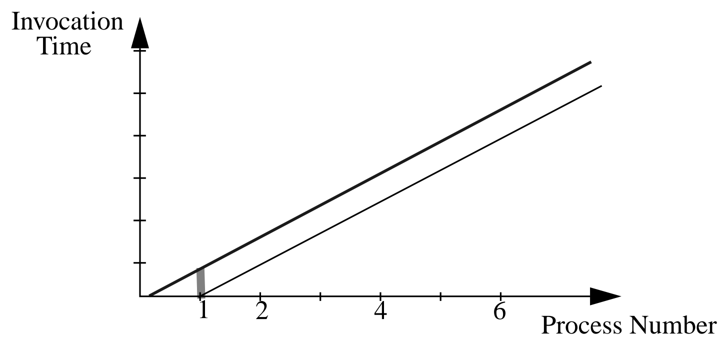


FIGURE 36. Method 1: Invocation times with overhead

2. The second possibility is to try to find an approximation, supposing that the execution time for no process is theoretically zero. Using in addition an estimation for the time taken by one process, using transmission (by dividing $T_{2processes}$ by 2), leads to a linear function passing through (0,0) and (1, $T_{2processes}$ / 2). This estimation should come closer to the measured values, because its overhead contains all the additional, time consuming factors, transmission may introduce.

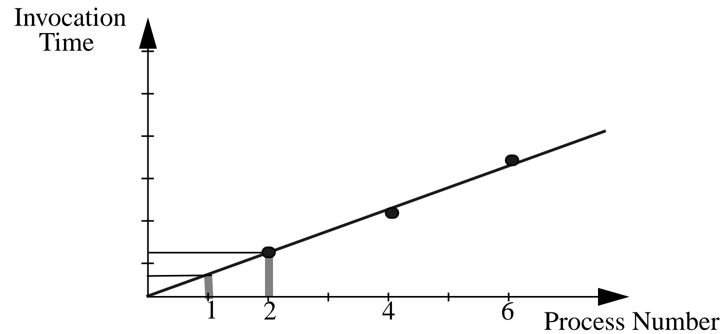


FIGURE 37. Method 2: Invocation times with overhead

Both solutions are not precise and they are based on very different assumptions. But they both give approximated estimations of the overhead to use in the final formula:

$$T_{TO\text{-}Broadcast} = s \cdot (n-1) \cdot T_{transmission} + T_{overhead}$$

where:

s	number of steps taken by the algorithm
n	number of processes
$T_{transmission}$	Transmission time for one message (Figure 34)
$T_{overhead}$	Algorithm overhead

5.3. Closed System Model

The measures presented next are obtained by using the program of Figure 25 in chapter 4.

Consider again a closed system, where all processes are basically servers. The round robin method is used to determine a different server for each round, acting as client process. Like that, the influence of the executing machine's relative speed is minimal.

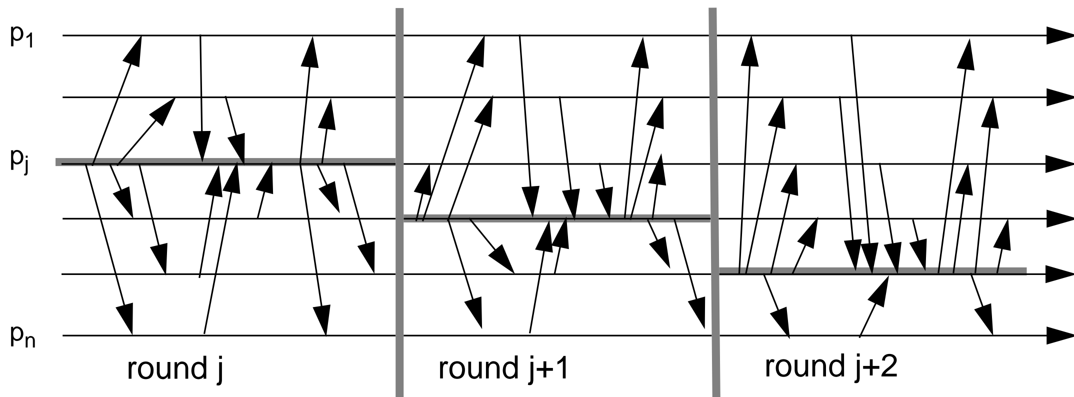


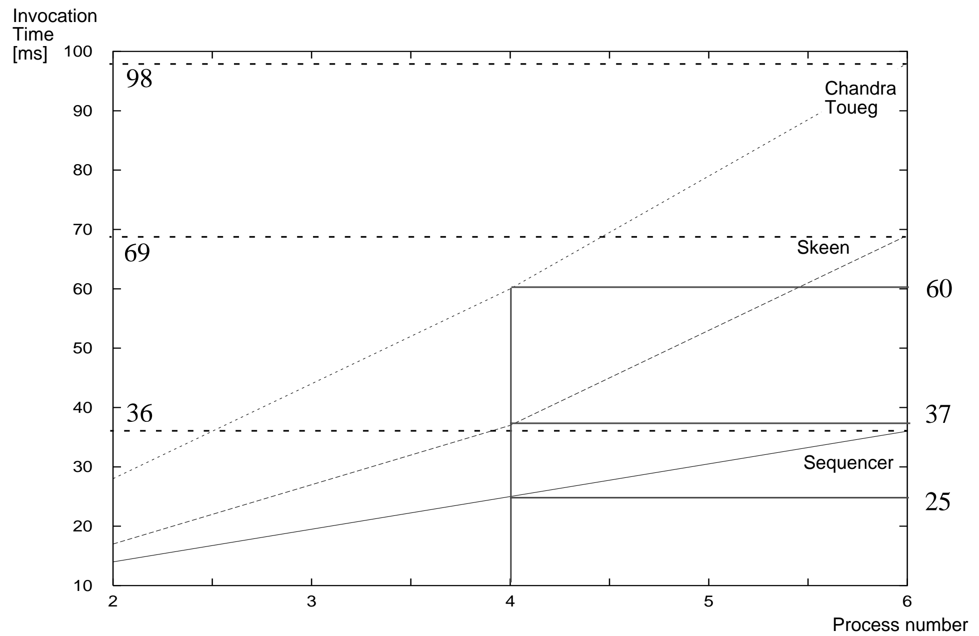
FIGURE 38. **Round robin (sequencer mode)**

The following measures have been obtained by executing the test program on two to six SparcStations20. The message size is (if not indicated explicitly) 100 Bytes.

For the following paragraphs some figures will be used to illustrate the results obtained. As there is only a small selection, showing typical proprieties, the other figures have been placed in Annex A.

• Algorithms

In order to do a comparison between the different algorithms, the next figure shows the execution time for a fixed middleware (in this case CORBA).



It can be seen, that the Sequencer algorithm is the fastest of all three. Also the estimated relative speed difference is approximately confirmed.
Taking the exact values:

Algorithm	Value	Difference
Chandra Toueg	98	
		29
Skeen	69	
		33
Sequencer	36	
		36

It can be seen, that the estimated relative speed difference between the algorithms is confirmed.

Considering the estimated execution time of a TOBroadcast, the measured value can be verified.

For 4 processes and the **first method** for overhead calculation:

$$4 \,.\, (4-1) \,.\, 3.6 + 4.2 = 47.4$$
$$3 \,.\, (4-1) \,.\, 3.6 + 2.6 = 35.1$$
$$2 \,.\, (4-1) \,.\, 3.6 + 2.1 = 23.6$$

These estimated values are too low for all algorithms (right-side of the figure).

For 4 processes and the **second method** for overhead calculation:

$$4 \cdot (4\text{-}1) \cdot 3.6 + (28/2) = 57.2$$
$$3 \cdot (4\text{-}1) \cdot 3.6 + (18/2) = 41.4$$
$$2 \cdot (4\text{-}1) \cdot 3.6 + (14/2) = 28.6$$

These values are generally better than the ones above, but still they do not give the exact estimation.

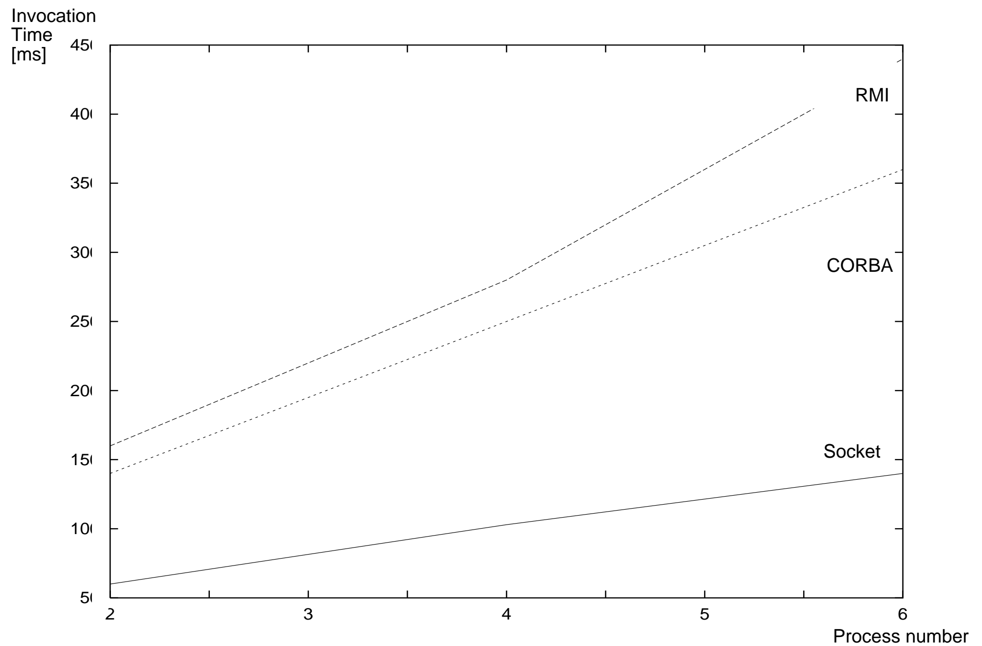
At this point it is very important to recall, that the performance is measured on a **asynchronous distributed system**. So no predictions on the relative overall performance can really be made. As the sending process does not have to wait for delivery, when sending a message, it is very difficult to measure the temporal behaviour of every process.

With Skeen's algorithm for example, it is possible, that the good performance result is destroyed by one or several processes, having a poor performance. This resides in the fact, that the sequencing process has to wait for a certain number of proposals before the decision can be made and thus can be slowed down, if the proposals do not arrive rapidly enough.

- The Sequencer algorithm is about 40% faster than Skeen and 100% faster than Chandra Toueg's algorithm.
- Execution time depends in a linear way on the number of processes.

• Middlewares

The following figure shows the execution time of one TOBroadcast call using a fixed algorithm (Sequencer in this case). It is clearly visible, that the estimation comes near to the real measures made. RMI is much slower than Sockets, and CORBA lies between them, above the middle, according to the measures in Table 3.

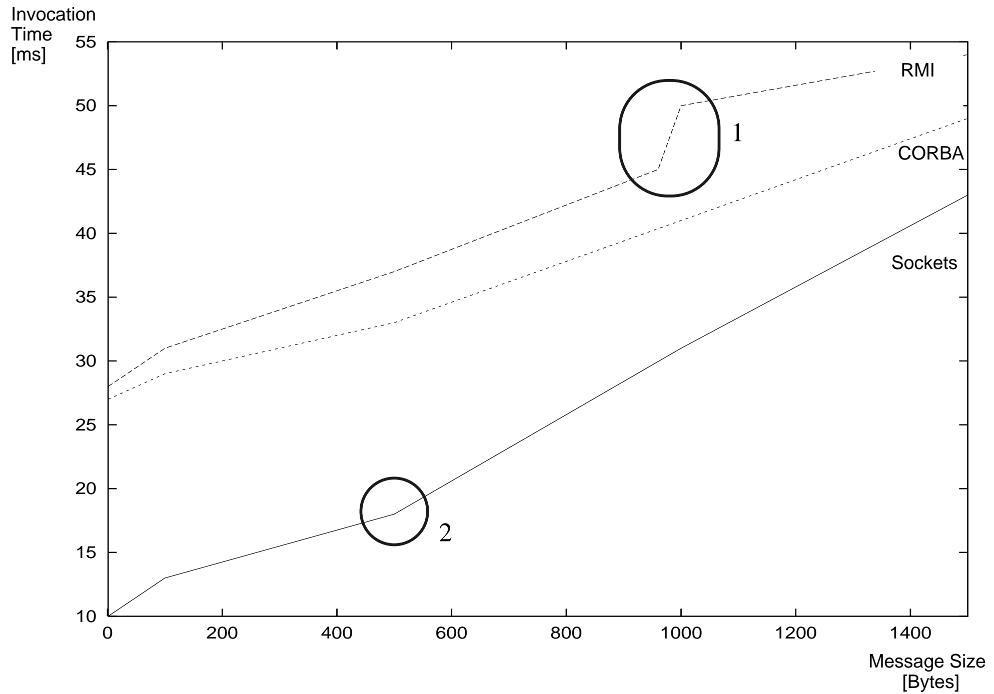


Execution time grows in a linear way to the number of processes used, because it depends directly on the number of messages transmitted.

- Sockets are 140% faster than CORBA, and 180% faster than RMI.
- Also in this case, the number of processes introduce a linear factor to the measures.

Message size

Consider a fixed algorithm (Sequencer) and a variation of the message size.



The difference to the Figure 34 for direct transmission is obvious. The effects are visible, because they become more important as the transmission volume grows. Executing the algorithms introduces a message flow, way beyond the volume of a simple Ping flow.

- **1** For RMI there is a jump at around 1000 Bytes indicating clearly, that RMI is working with a fixed packet size of approximately 1KB.
- **2** The change for Sockets at about 500 Bytes is due to the default stream buffer size of 512 Bytes.

5.4. Client Server Model

This measures finally do not represent the everyday use of the algorithms, as they form a closed system, without connection to the outside. To change this consider the following two models:

Case A

In **Case A**, the **Client TO-Broadcasts** the message to all Servers, which TO-Deliver it. Then they process the request and send back a reply to the calling Client. The time considered is measured between the send of the last request and the first reply.

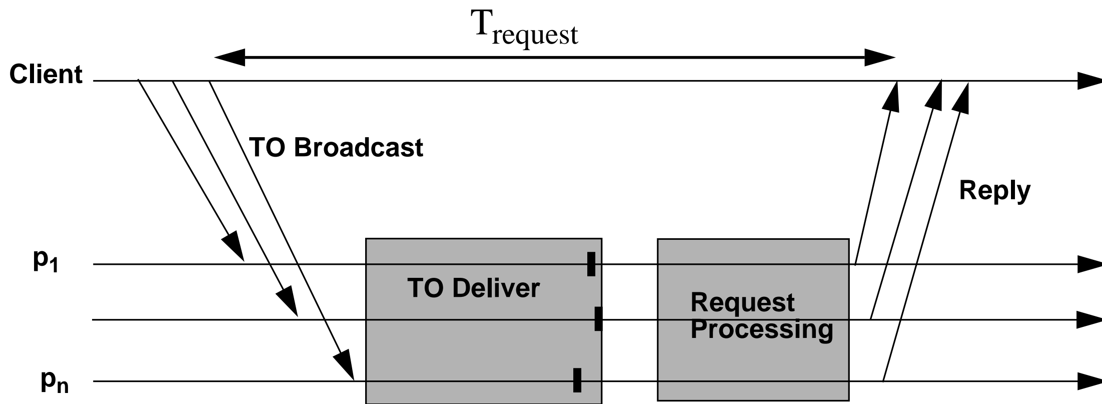


FIGURE 39. Case A Interprocess Communication

The following graph shows the measures of $T_{request}$ for a fixed algorithm(Sequencer), with a reasonable message size of 100 Bytes.

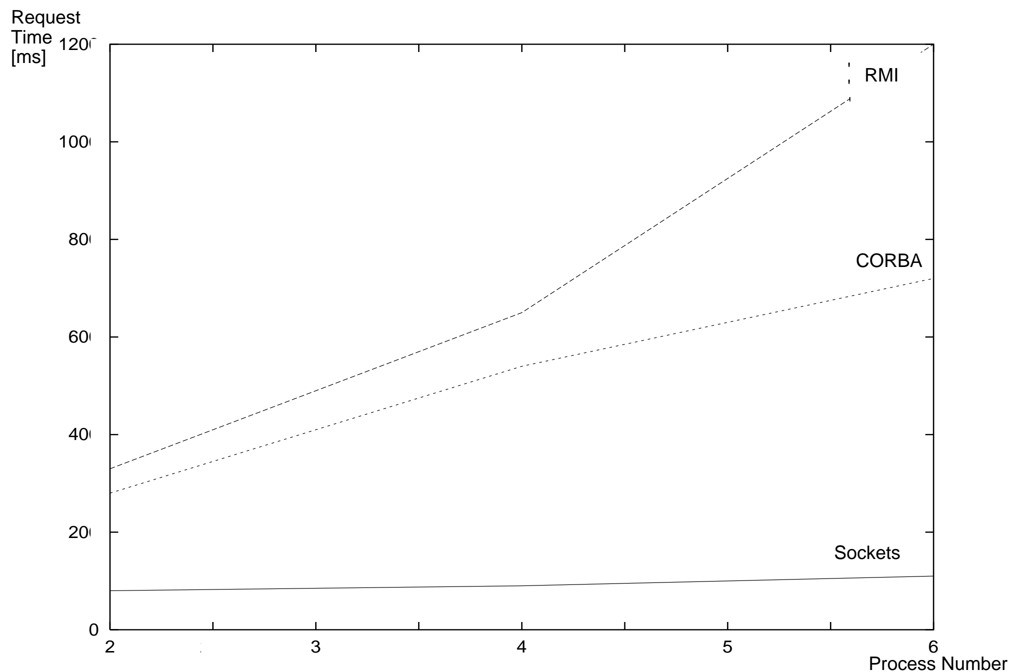


FIGURE 40. Case A: Interprocess Communication

Case B

In **Case B**, the Client sends its request to one of the Servers, who TO-Broadcasts the message after reception. The **TO-Broadcast** is executed only **in the closed system** of Servers, so this case is more similar to the one presented at the beginning.

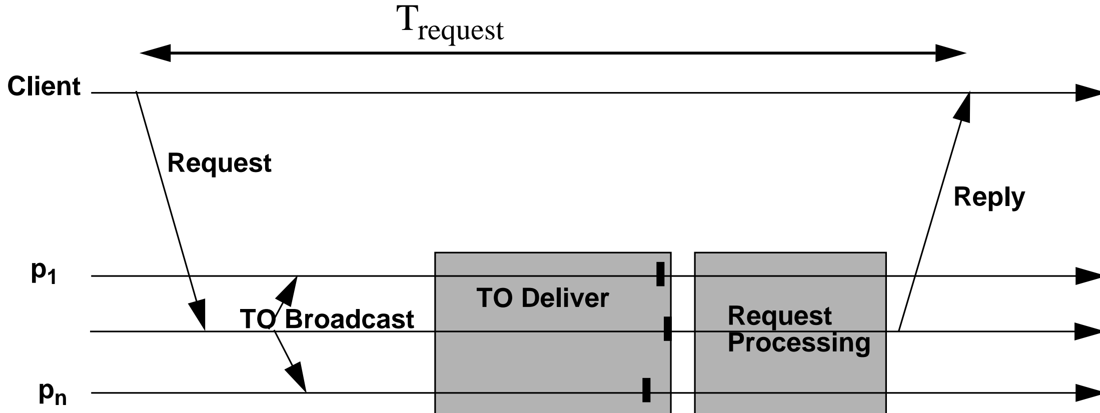


FIGURE 41. **Case B: Interprocess Communication**

The following graph also shows $T_{request}$ for a Sequencer algorithm.

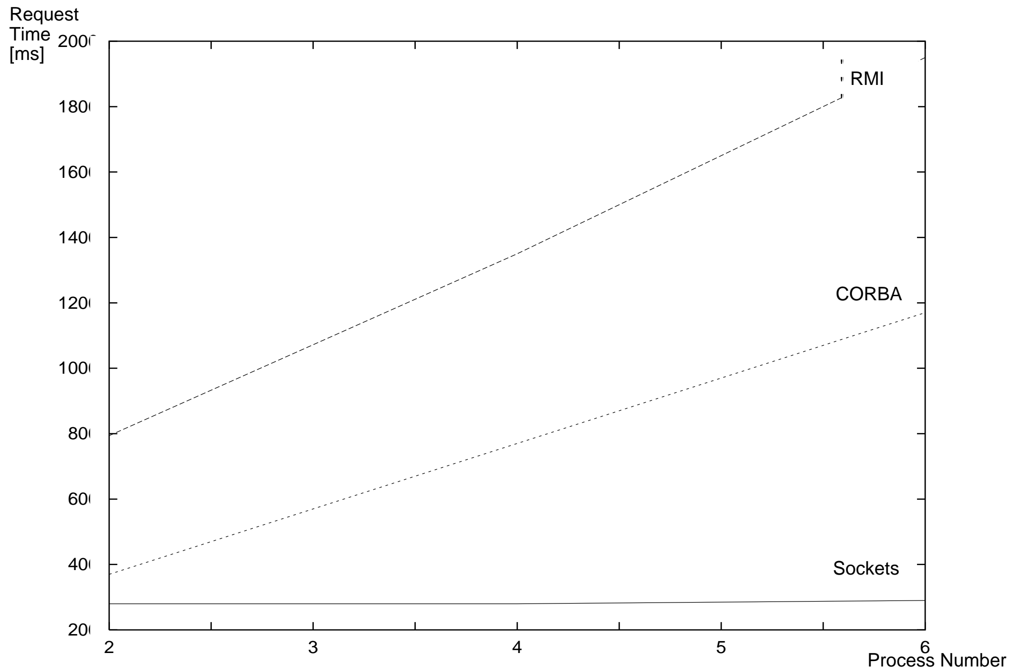


FIGURE 42. **Case B: Invocation Time**

Conclusion:
It seems clear, that the second solution (**CaseB**) is **slower than** the first one (**CaseA**). For every invocation the Client has to wait for the reply of the Server to which it sent the request. The Case A solution just has to wait for the reply of the fastest of the Servers. The remaining graphs can be found in Annex B.

Sequencer saturation

With this model it is also possible to show the bottleneck-situation in the Sequencer algorithm when meeting high request throughput from the client side.

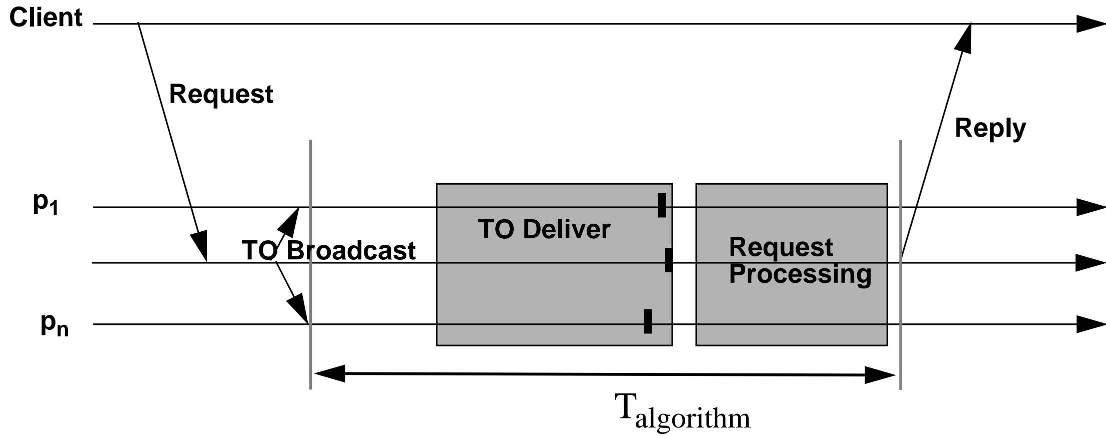


FIGURE 43. **Sequencer saturation: $T_{algorithm}$**

The following graph shows this situation giving the time $T_{algorithm}$ for a 3 Client / 4 Server configuration.

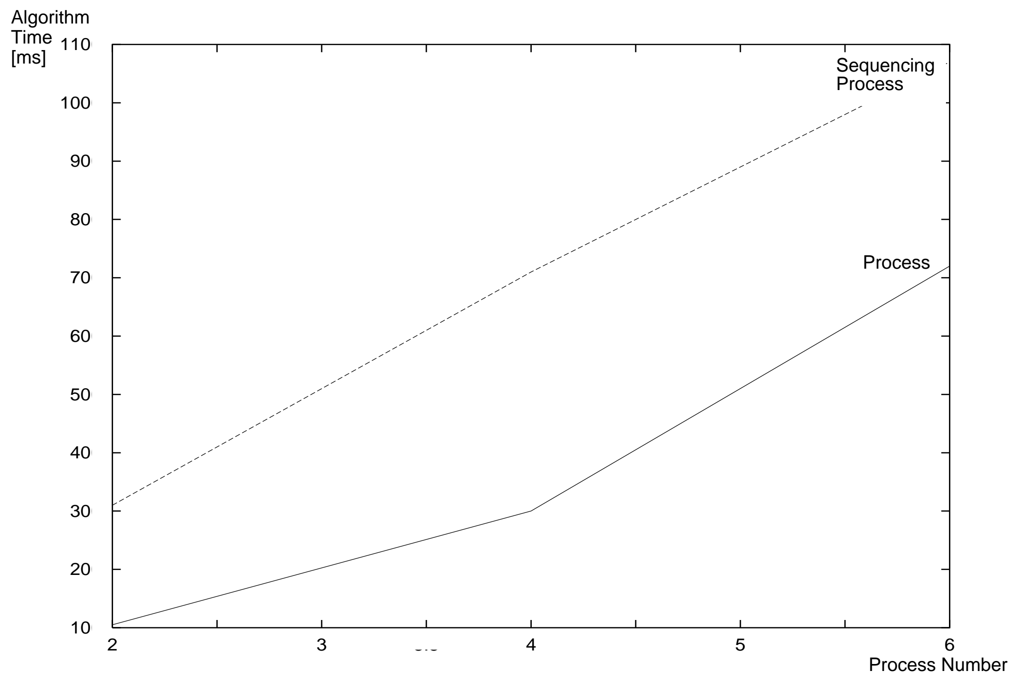


FIGURE 44. **Sequencer saturation**

Clearly the Sequencing process is slower than the others, slowing down the whole system. This is visible especially for Case B, because the Client has to wait for the reply coming from the Sequencer.

6

Conclusions

Project task

This diploma project had to **study** the development of an application for **atomic broadcast**, considering only failure free runs, and to **compare several possible solutions**, working on different middlewares and using different algorithms.
The goal was to **implement** the studied solutions and to provide example programs.
Finally **performance measures**, varying different parameters, had to be made in order to provide objective conclusions by **comparing with** the preceding **theoretical performance estimations** made.

Solution

After having **analysed** the different **algorithms and middlewares**, we proposed a possible **implementation**, introducing the three layer model, providing uniform interfaces.
The goal was to provide a solution, which is **transparent** for the user, completely concealing the underneath structure of middlewares and algorithms.
A **registry** based process administration has been introduced to guarantee the mentioned transparency by making it possible to connect the processes always using the same procedure.
The **performance estimations** have been elaborated, based on theoretical considerations.
Finally the estimations have been compared to the **measured** values. If necessary, the differences have been verified with other programs and/or explained.
For easier use, a slightly **modified implementation** has been done, providing the Client / Server Model described in chapter 5.4.Client Server Model on page 65.

Based on the accomplished work it should be possible to rate the different solutions.

•Algorithms

Sequencer
This algorithm is very fast but it can decrease in performance if the request throughput is high (Figure 44 on page 67).

Skeen
Skeen's algorithm is almost as fast as the Sequencer algorithm, avoiding the bottleneck by better distributing the request load.

Chandra Toueg
This is the only algorithm, that could be used in real systems, admitting failures. Chandra and Toueg's algorithm has a poor performance compared to the two others.

•Middlewares

Sockets
Sockets present the fastest communication of all three middlewares. However they are not very easy to handle and in certain cases the performance can heavily decrease (Appendix B, Sockets). Unfortunately it is not possible to implement dynamic Client-binding (as with RMI / CORBA), without performance loss. The most important point is, that Sockets are a part of all JAVA versions, and though are largely available.

RMI
RMI is very slow, due to the used JAVA based procedure (e.g. object serialization). Performance is stable, only for request having a big size it decreases dramatically. In this case, the biggest advantage is also the RMI integration with JAVA 1.1. But as it seems, RMI will soon be replaced by other concepts.

CORBA
CORBA is the most powerful middleware used for this project. Even though it is faster than RMI, it provides a very stable performance, so CORBA is less sensible to environmental influences (e.g. network charge) than the other two solutions. The only inconvenient is the fact, that CORBA is not very present on the actual systems. Surely, CORBA will take the inverse way of RMI, and become one of the most used middlewares, due to its power, flexibility and performance.

•Recommendations

The algorithm / middleware combinations should be made with Sockets / CORBA and Sequencer / Skeen. RMI and Chandra Toueg should only be used if there is a need for it.

So, if request processing is long, then a more sophisticated combination can be used (e.g. Skeen / CORBA), because the longer TO-Delivery time will not be noticed by the user.

If the application is time-critic and the request throughput is low, the Sockets/ Sequencer combination should be used, providing a mean request transmission time of about 8 ms. If request throughput is high, use Sockets/Skeen (11.5 ms).

Future Work

There are several directions for improvement. First of all, Chandra Toueg's algorithm could be extended by *introducing failure detection* and thus be able to run in an environment allowing process failures.

Another change would be to improve flexibility by offering a *more sophisticated registration process*.

There could be attempts to *improve performance*, modifying the used algorithms and using other middlewares, like Orbix or the soon to be released JAVA 1.2.

References

[1] R. Orfali, D. Harkey *Client/Server Programming with JAVA and CORBA* Wiley Computer Publishing 1997

[2] E.R. Harold *Java Network Programming* O'Reilly 1997

[3] P.M. Tyma, G.Torok, T. Downing *Java Primer Plus* The Waite Group Inc. 1996

[4] S.R Davis *Learn Java Now* Microsoft Press 1996

[5] R. Guerraoui and A. Schiper *Fault-Tolerance by Replication in Distributed Systems - A Tutorial* Proc International Conference on Reliable Software Technologies (Invited Paper), Springer Verlag (LNCS), 1996

[6] R. Guerraoui and A. Schiper *Total Order Multicast to Multiple Groups* IEEE International Conference on Distributed Computing Systems (ICDCS-97), Baltimore, May 1997

[7] R. Guerraoui and A. Schiper *Consensus: the Big Misunderstanding* IEEE International Workshop on Future Trends in Distributed Computing Systems (FTDCS'97), Tunis, October 1997

[8] R. Guerraoui *Revisiting the relationship between Non Blocking Atomic Commitment and Consensus problems* Distributed Algorithms (WDAG-9), Springer Verlag (LNCS), Le Mont Saint Michel (France), September, 1995

[9] VIsiBroker Online Reference Manual and Programmers Guide.

[10] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes *Object-Oriented Development The Fusion Method* Prentice Hall, 1994

[11] T.D. Chandra, V. Hadzilacos and S. Toueg *The Weakest Failure Detector for Solving Consensus* In Proc 11th annual ACM Symposium on Principles of Distributed Computing, 1992

Index

A
agreement 20
Algorithm 29, 39, 55, 61
asynchronous 62
    system 11
Atomic Broadcast 8
Atomicity 13, 16

B
bank account 9, 13
Berkeley Socket 23
bottleneck 17
buffer 50, 56

C
Chandra 20, 22, 29, 40, 55
Client 11
closed system 15
Communication 23, 49
Consensus 20
coordinator 21
CORBA 27, 29, 36, 47, 52

D
datagram 23
dispatching 32, 56
distributed system 10, 62
distributed systems 9

F
failure detectors 21
fault tolerant 9, 10
FLP Impossibility 20

I
ID 30, 31, 34, 36, 43
input stream 31, 32

L
Lamport 18, 31, 34, 36
Layer 30, 39

M
Message Polling 51
message size 57, 64
Middleware 23, 55, 63

N
network 54
non-triviality 20

O
Object serialization 25
OMG 27
ORB 27
output stream 31

P
performance 22, 42, 53, 62
Ping 55
port 24
process 11

R
R-Broadcast 16
R-Deliver 16
register 26
Registration 43
registry
  server 30, 31, 36
  service 26
registry server 34
reliable channels 16
Remote object 25
RMI 25, 29, 34, 45, 52

S
Sequencer 16, 18, 22, 29
Server 11
ServerSocket 24
Skeen 18, 22, 29, 55
Skeleton 25
Socket 23, 29, 31, 44, 49
stub 25

T
Termination 13, 16, 20
timestamp 18, 32
TO-Cast 13, 41
TO-Deliver 41
TO-deliver 13
Total Order 13, 15, 16, 39
Toueg 20, 22, 29, 40, 55
Transmission 50
transparency 11, 43

U
Uniform Agreement 13
Uniform Total Order 13

V
VisiBroker 27

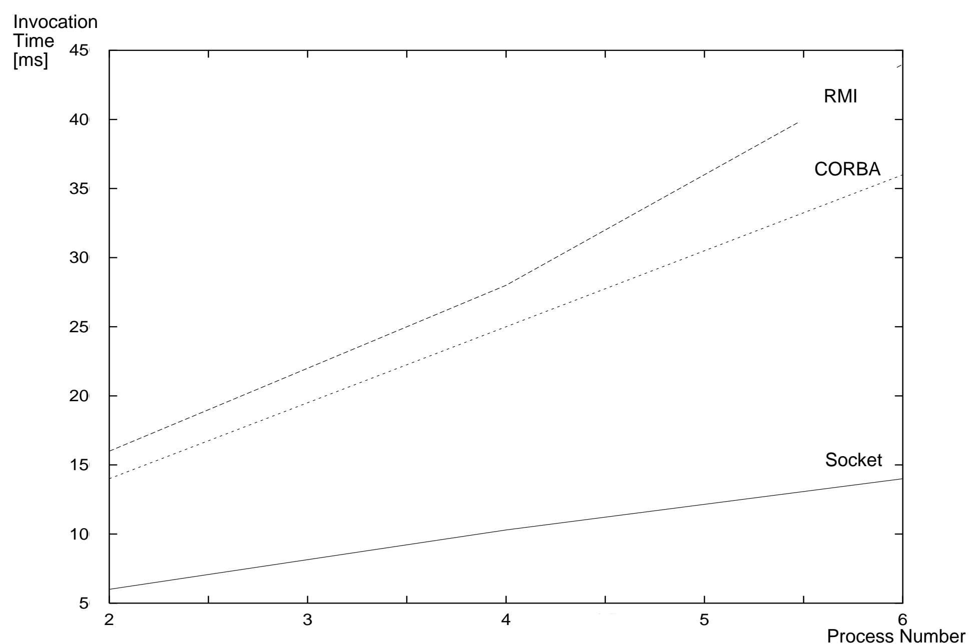
Appendix A Performance Measures Closed System Model

This chapter contains a collection of performance measure figures that had not been presented in the paragraph "Closed System Model" in chapter 5.

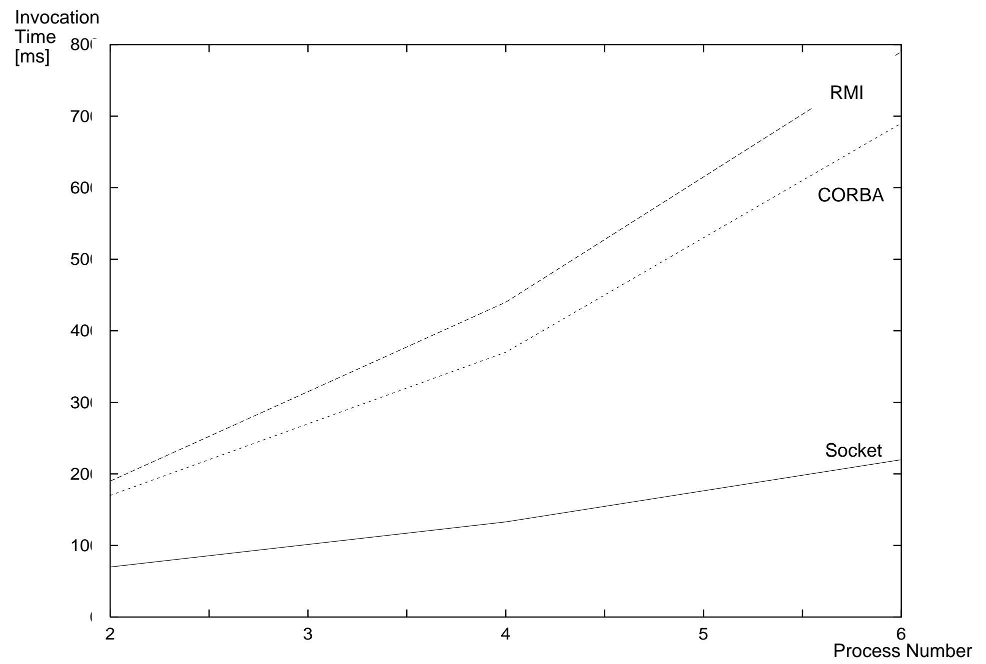
If not explicitely specified, all measures have been made for a message size of 100 Bytes.
The next three figures represent the measures made with a fixed algorithm, varying the number of processes. They are useful to compare middleware performance.

A.1 Invocation time performance

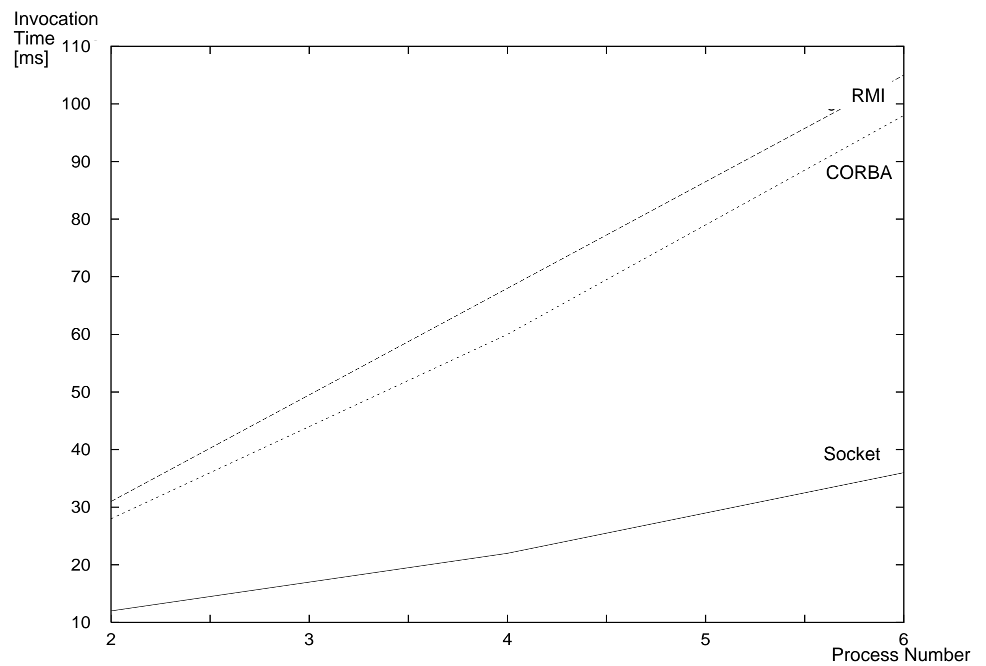
Sequencer



Skeen

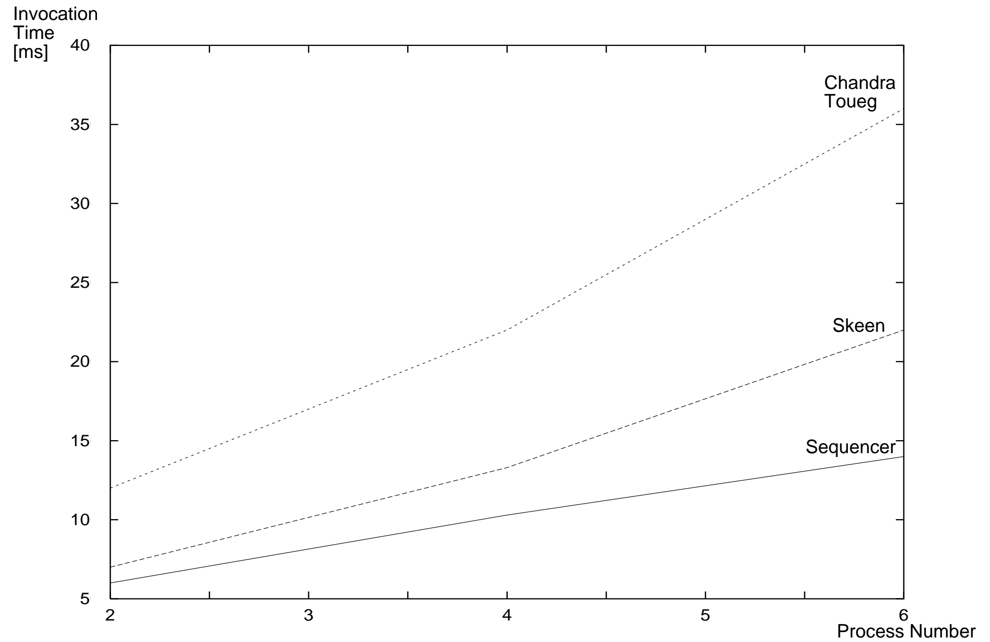


Chandra Toueg

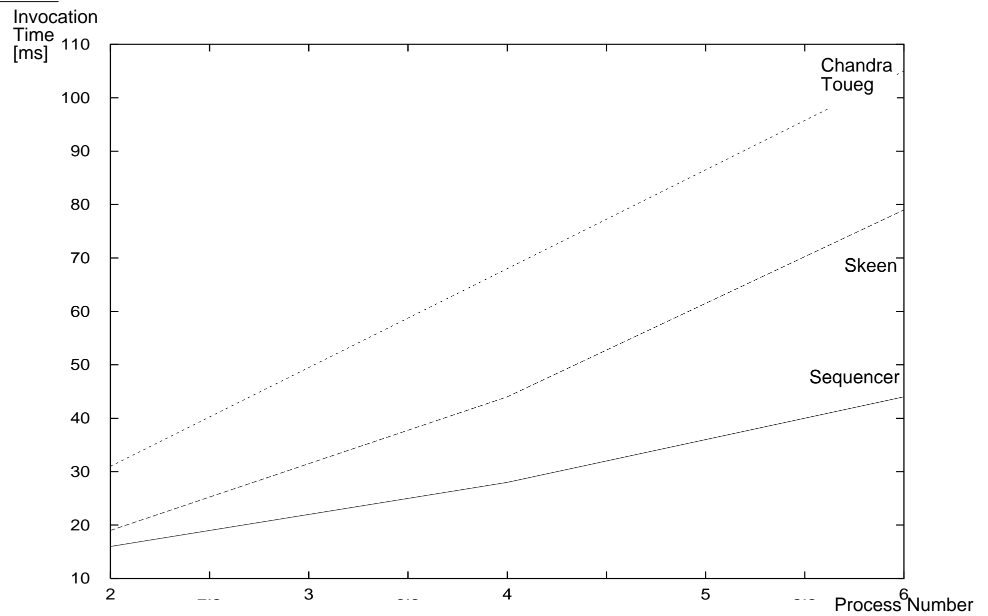


The next three figures represent the measures, made with a fixed middleware, varying the number of processes. It is useful to compare algorithm performance.

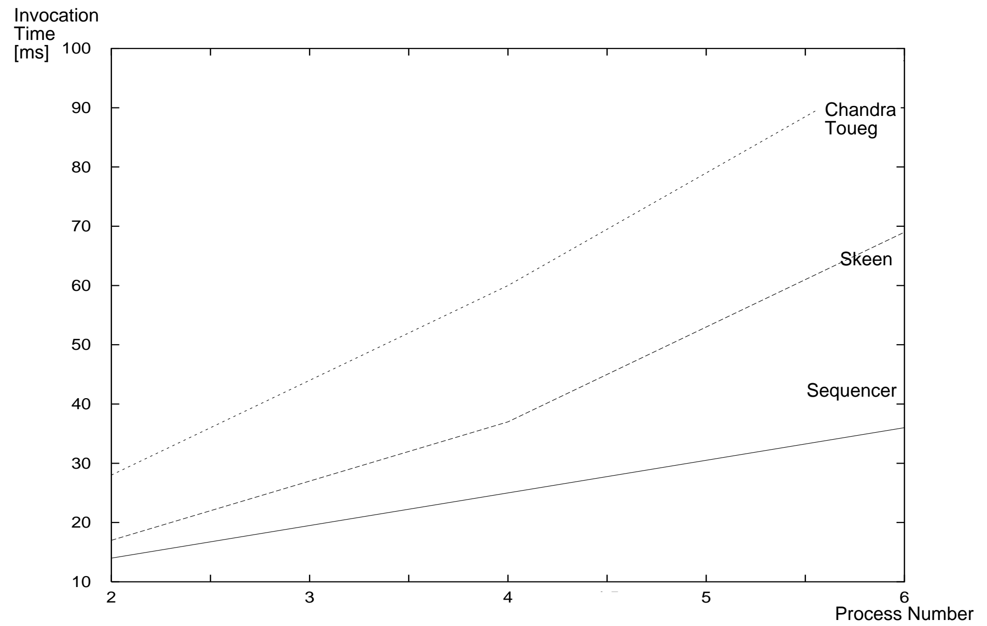
Socket



RMI

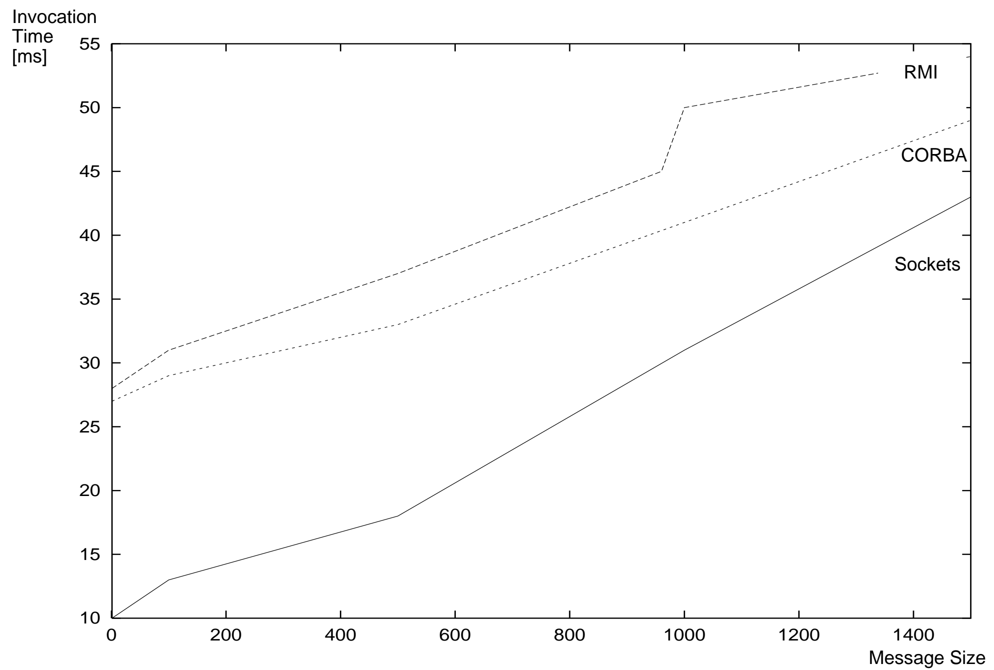


CORBA



Message size

The next figure shows the execution time of the sequencer algorithm, for varying message size.



Exact measures

The following tables give the exact execution time values for the different combinations, made for 100 Byte messages.

	Sequencer	Skeen	Chandra Toueg
SOCKET	6ms	7ms	12ms
RMI	16ms	19ms	31ms
CORBA	14ms	17ms	28ms

TABLE 4. Measures on 2 Stations

	Sequencer	Skeen	Chandra Toueg
SOCKET	10ms	13ms	22ms
RMI	28ms	44ms	68ms
CORBA	25ms	37ms	60ms

TABLE 5. Measures on 4 Stations

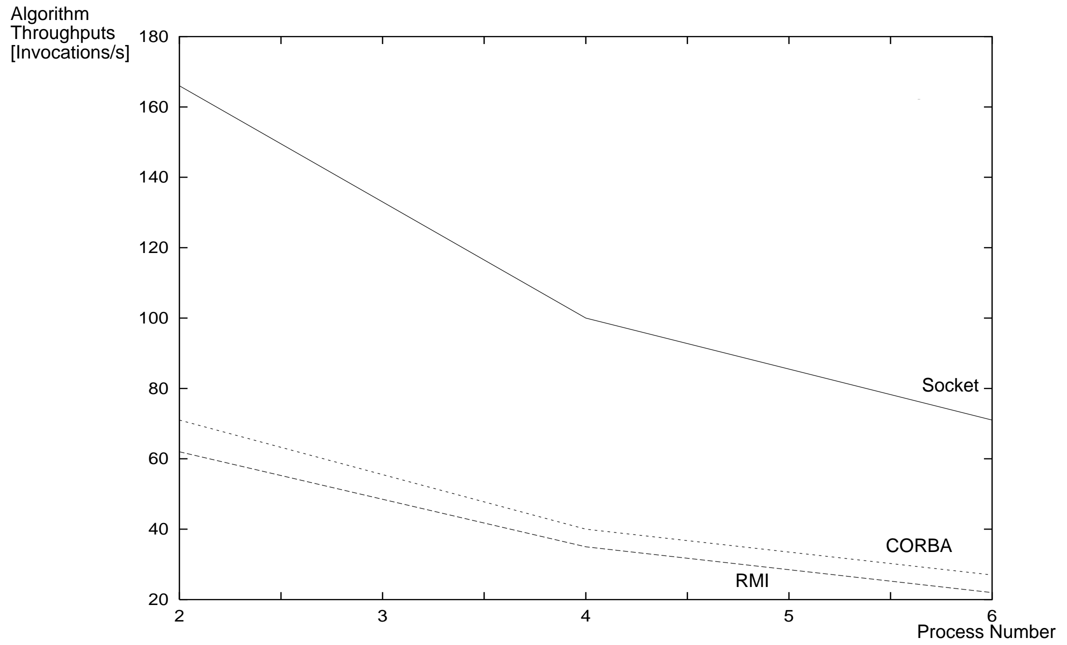
	Sequencer	Skeen	Chandra Toueg
SOCKET	14ms	22ms	36ms
RMI	44ms	79ms	105ms
CORBA	36ms	69ms	98ms

TABLE 6. Measures on 6 Stations

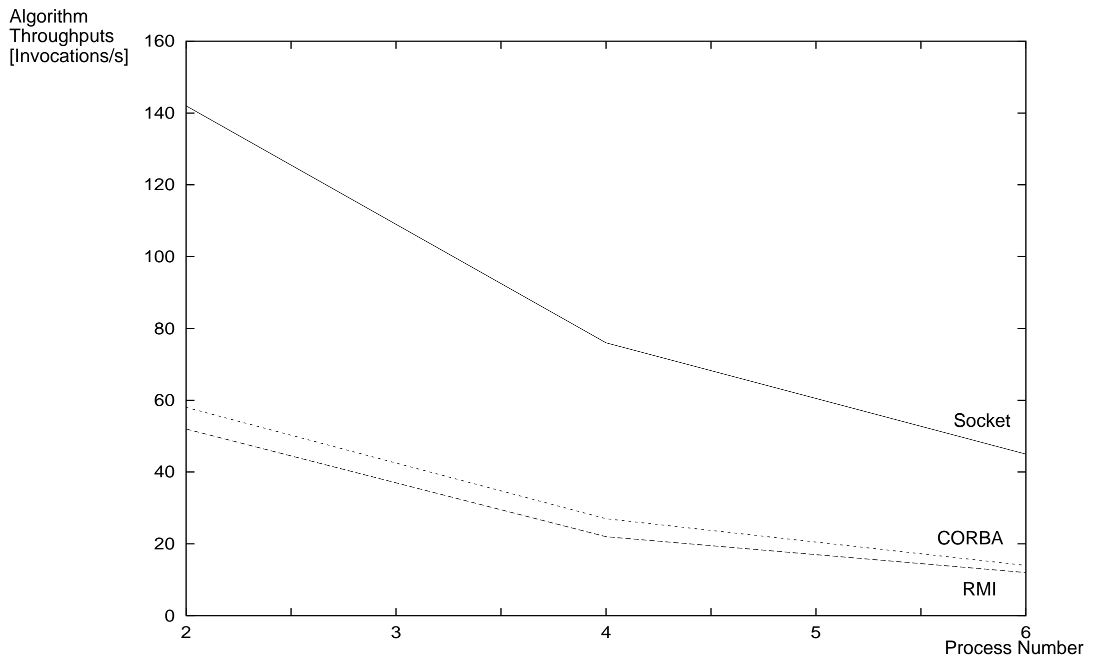
A.2 Algorithm throughputs

This paragraph shows the throughputs (invocations per second) of the six possible combinations of algorithm and middleware.
The next three figures represent the measures, made with a fixed algorithm, varying the number of processes.

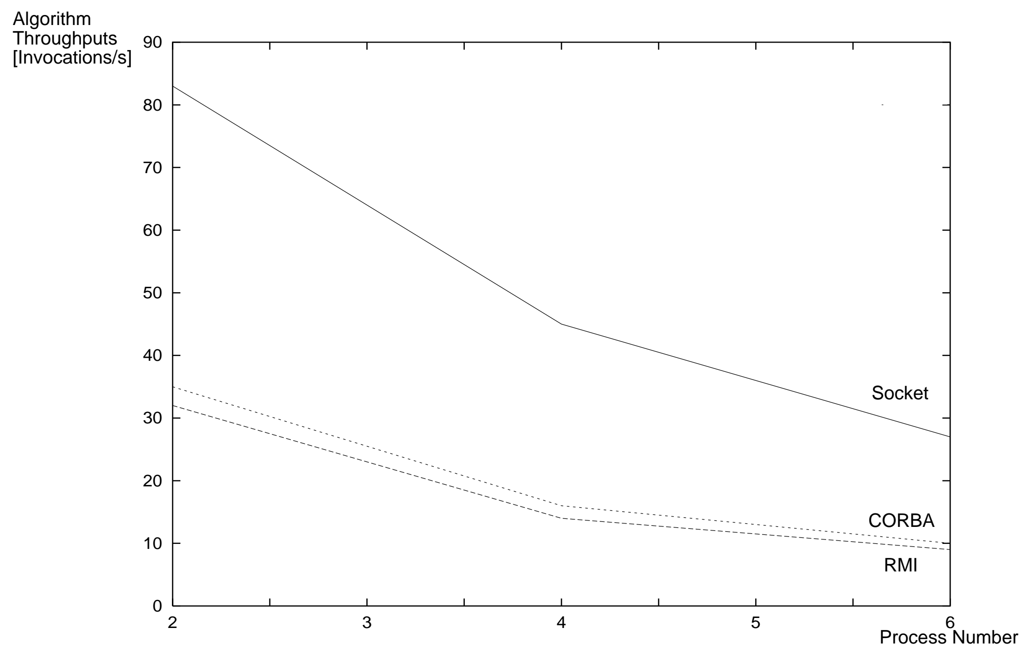
Sequencer



Skeen

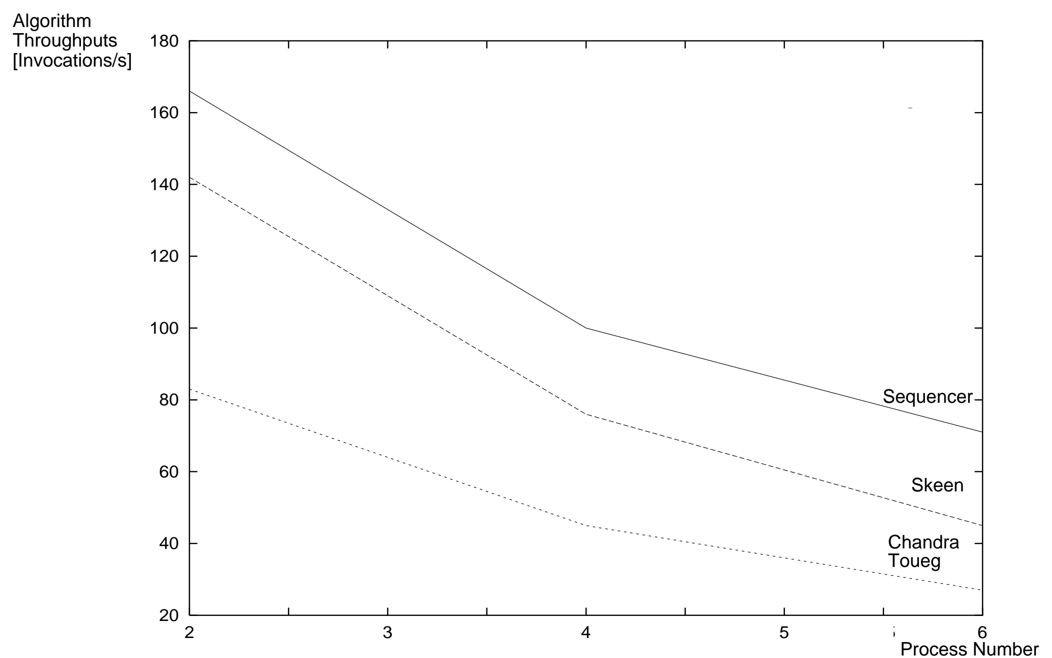


Chandra Toueg

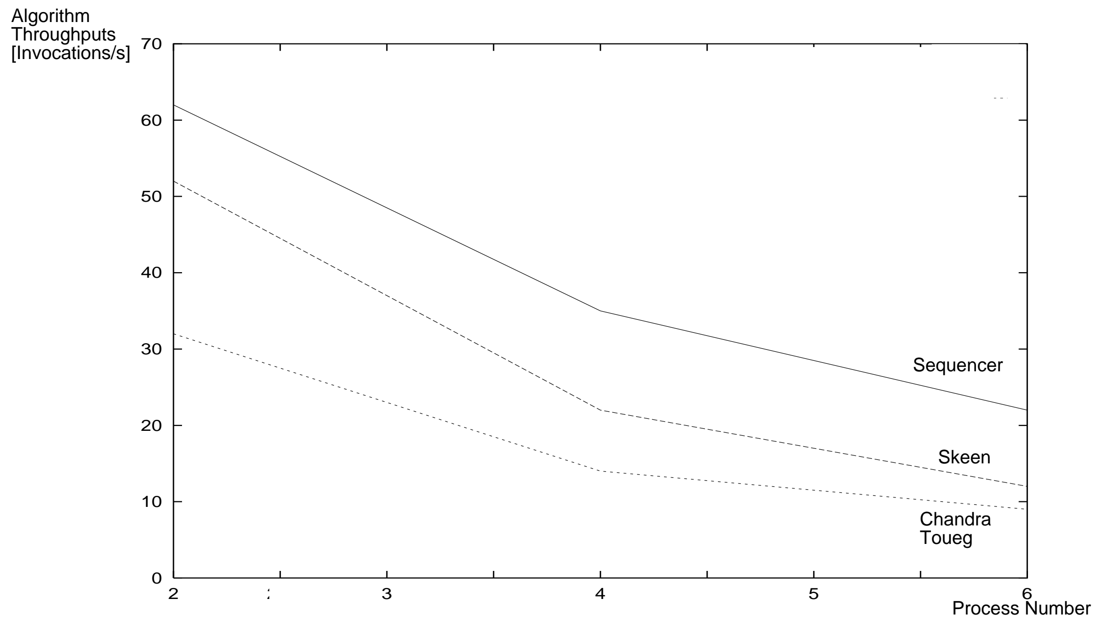


The next three figures represent the measures, made with a fixed middleware, varying the number of processes.

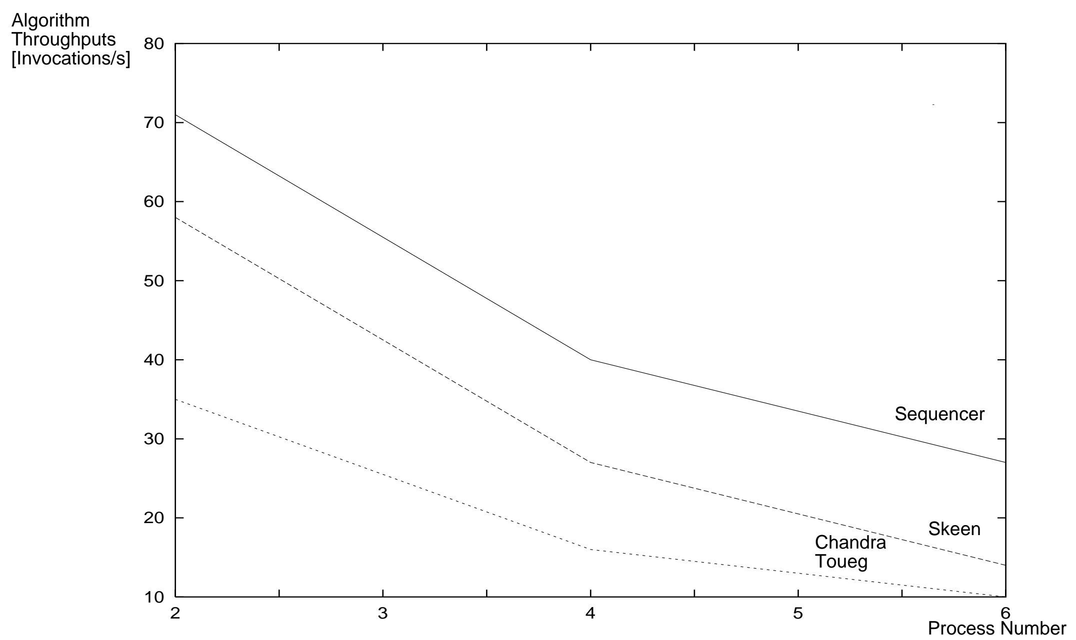
Socket



RMI



CORBA



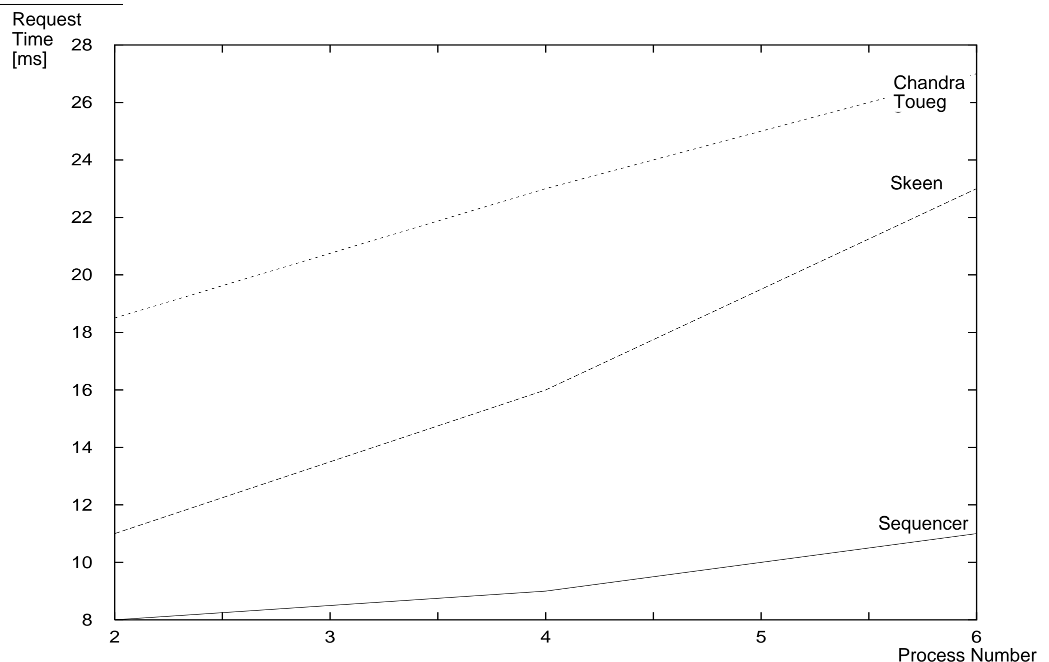
Appendix B Performance Measures Client Server Model

This chapter contains a collection of performance measures figures that had not been presented in the paragraph "Client Server Model" of chapter 5.

B.1 Case A

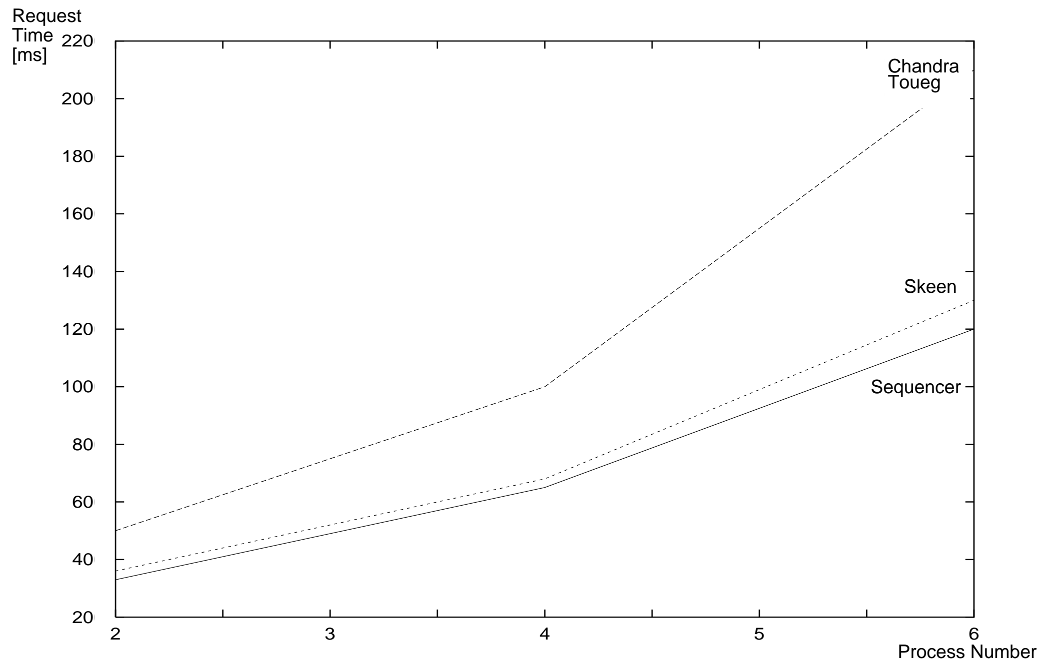
The next three figures represent the measures made for the Case A presented in chapter 5,with a fixed middleware, varying the number of processes.

Socket

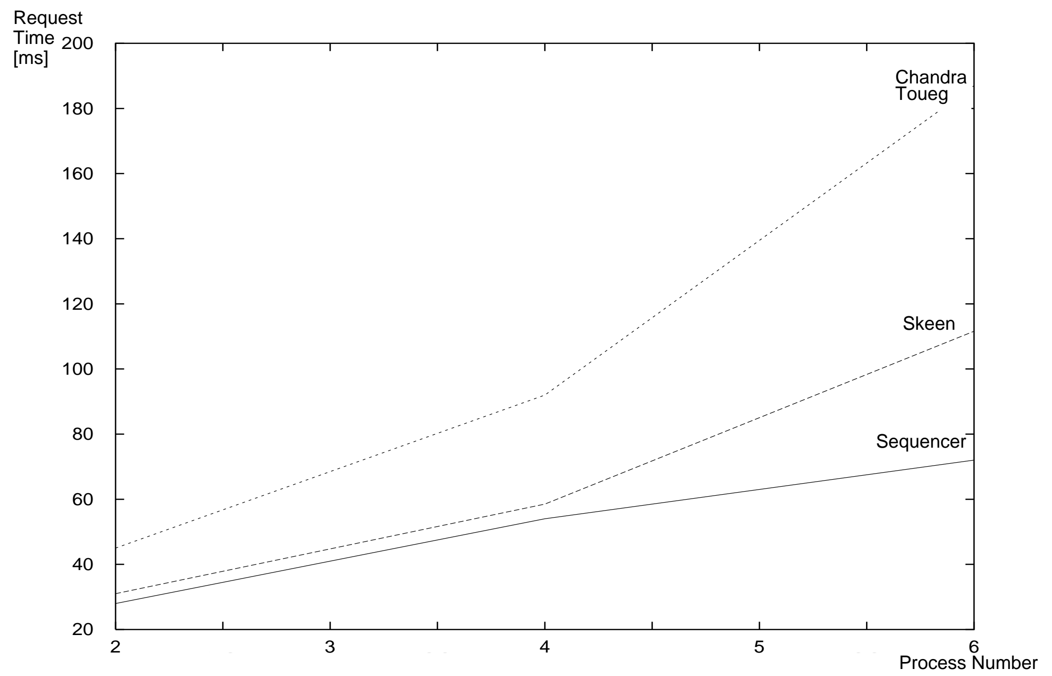


Case A

RMI



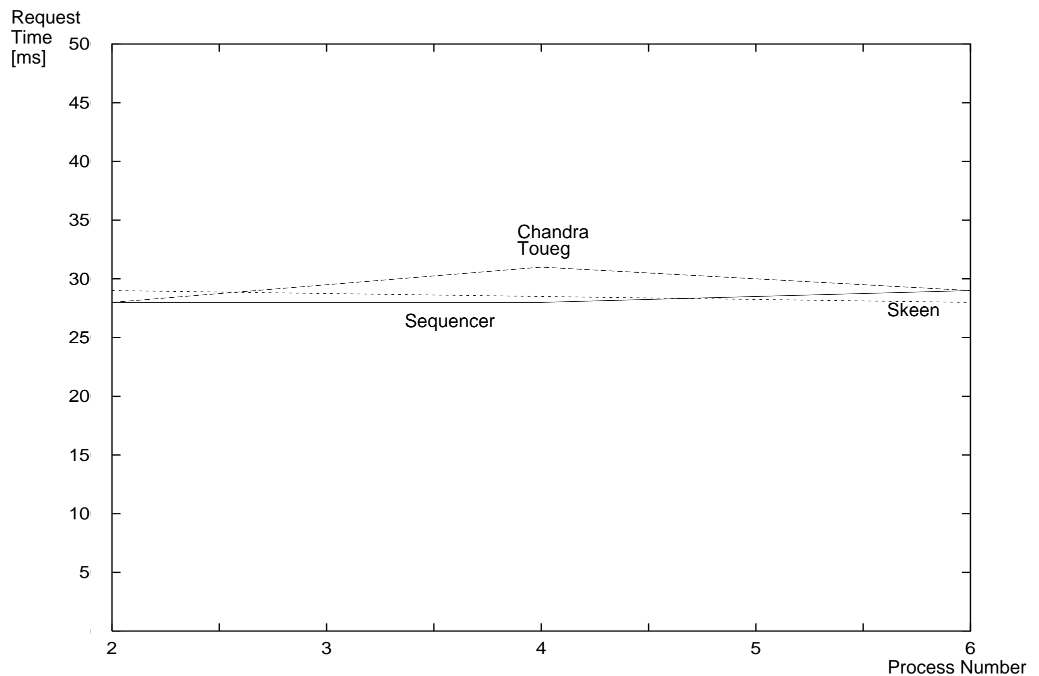
Corba



B.2 Case B

The next three figures represent the measures made for the Case B presented in chapter 5, with a fixed middleware, varying the number of processes.

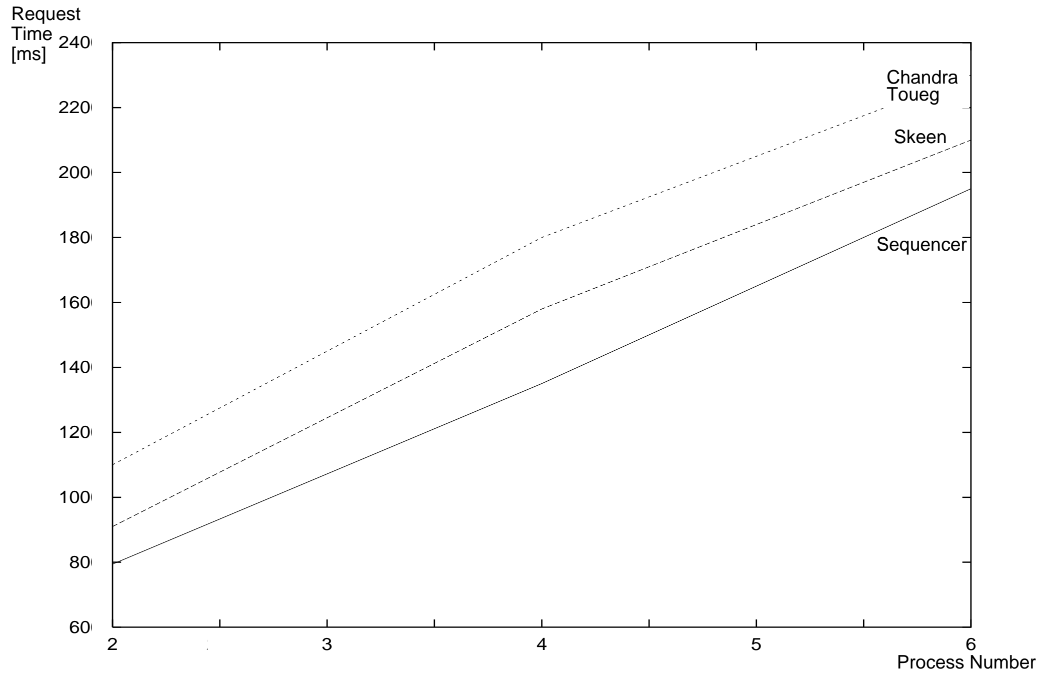
Socket



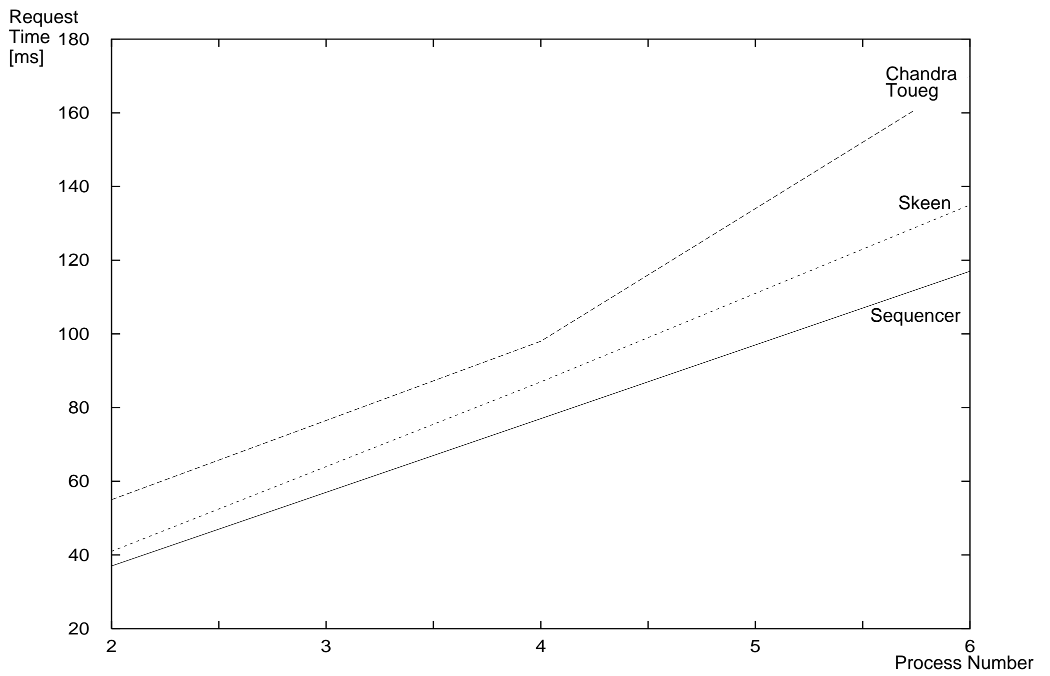
These surprising measures can be explained with the lack of speed of the invoking Client. As it has to wait for a response to every invocation, and as the time of reply depends on the speed of the called process, the number of processes does not influence the executing time of the algorithms. This effect is only visible with Sockets, because they are fast enough to be delayed. The other two middlewares are basically too slow to make this delay visible.

Case B

RMI



CORBA



Appendix C Example and Usage

In this chapter we show the usage of the provided classes for the Client Server Model presented in chapter 5. We will describe the implementation of a simple Client/Server and its correct installation.

C.1 Interfaces

•COMMON FUNCTIONS

public SecondLayer(int algorithm, String regserver, int processType)

constructor for the Second Layer class:

algorithm is the algorithm to be used:

0 Sequencer

1 Skeen

2 Chandra Toueg

regserver is the address of the registration server

processType indicates if the process is a Client or a Server:

0 Client

1 Server

public int connect()

connects the process to the registry server, returning the ID number

public int getServerNum()

returns the number of Servers connected

public int getClientNum()

returns the number of Clients connected

•SERVER FUNCTIONS

public String WaitForRequest()

gets the next request by using the TO algorithms. Non blocking, returns ““ when no message can be TO-Delivered.

public void SendReply(String msg)

sends the reply **msg** to the Client, which sent the processed request

•CLIENT FUNCTIONS

public void SendRequest(String msg)

sends a request **msg** to a Server, which will be TO-Delivered by them

public String GetReply()

receives the reply issued from the called Server(s)

Be careful not to forget, that in the Case A, all the Servers will send a reply. So the Client will get n replies. This value n can be determined with getServerNum().

C.2 Server

First we implement a test server only showing the messages sent by the client and sending them back, adding the String “OK_“.
The example is a standalone version.

- Create a class as you usually would in JAVA:

```
class TestServer
{

 public static void main(String args[])  {

     if (args.length==2){
```

- To initialize the Server we need the following information:

```
algorithm=Integer.parseInt(args[0]); /* Type of the algorithm used
regserver=args[1]; /* Address of the registry server

System.out.println("TESTServer coming up....\n");
System.out.println("Algorithm        : "+algorithm+"  \n");
System.out.println("Reg_Server      : "+regserver+"\n");
```

- Then we create the underneath Layers. This command performs automatically all the needed initialisations:

```
mySecondLayer = new SecondLayer(1,algorithm,regserver); /* The first
                                                          parameter
                                                          (“1”) means,
                                                          that we will
                                                          register a
                                                          server
```

- Connect (register) the server.:

```
mySecondLayer.connect();
```

- Infinite main loop reading and displaying messages:

```
for(;;){
```

- Try to TO-Deliver a message:

```
String actMessage = mySecondLayer.WaitForRequest();
```

- Test if a message could have been TO-Delivered (actMessage!=””) :

```
if (actMessage!=””){

    System.out.println("RECEIVED :"+myID+"    "+actMessage);
```

- Reply to the Client:

```
            mySecondLayer.SendReply("OK_"+actMessage);
        }

    }
  }
```

- If the number or type of arguments is wrong:

```
    else{
        System.out.println("APP alg regserver ServerID............\n");
        System.out.println("0=Sequencer \n  1=Skeen \n  2=ChandraToueg \n ");
    }
 }
}
```

C.3 Client

The creation of the client is strongly connected with the design of user interfaces. We won't deal with this, as after all this is a purely personal aspect. The client presented here is very simple and does only send one message to the servers and display the response.

- Create a class as you usually would in JAVA:

```
class TestClient
{

 public static void main(String args[])  {

    if (args.length==2){
```

- To initialize the Server we need almost the same information as above:

```
        algorithm=Integer.parseInt(args[0]); /* Type of the algorithm used
        regserver=args[1]; /* Address of the registry server

        System.out.println("TESTServer coming up....\n");
        System.out.println("Algorithm        : "+algorithm+"  \n");
        System.out.println("Reg_Server      : "+regserver+"\n");
```

- Then we create the underneath Layers. This command performs automatically all the needed initialisations.:

```
        mySecondLayer = new SecondLayer(0,algorithm,regserver);
```

- Connect (register) the client:

```
        mySecondLayer.connect();
```

- Infinite main loop reading and sending messages:

```
        for(;;){
```

- Read the message from the keyboard:

```
            String message= System.in.readln();
```

- TO-Broadcast the message:

```
            mySecondLayer.SendRequest(message);
```

- Get the response from the Server(s):

```
            System.out.println(" SUCCESSFULLY DELIVERED :"+myID+"    "+
            mySecondLayer.GetResponse());
        }
```

```
    }
```

- If the number or type of arguments is wrong:

```
    else{
        System.out.println("APP alg regserver............\n");
        System.out.println("0=Sequencer \n  1=Skeen \n  2=ChandraToueg \n");
    }
  }
}
```

It is very important that you understand the distribution of the ID numbers on the client side: Your process will be bound with the type you indicated when creating a class instance of SecondLayer.

As we have seen, the process IDs are automatically assigned by connecting order. For example with three clients (two connected) and five servers:

Client 1	Client 2	Client 3	Server 1	Server 2	Server 3	Server 4	Server 4
1	0	NC	6	4	7	3	5

C.4 Compilation

You should have ALL the Java files (including your code) in ONE directory. Modify the Makefile by replacing the Client.java and Server.java files by the filenames of your application, or modify directly the two files. Then you can create an executable by typing:

make *[socket rmi corba]*

The second keyword gives you the type of middleware that will be used. Be sure that RMI or CORBA are correctly installed and accessible on your computer and that your paths are correctly set (refer to the respective manuals for this purpose).

You should now get an executable of your application. Before starting, don't forget to start the registry servers:

Sockets: ***java Registry*** *NumberofClients NumberofServers* (exact numbers!)

start your Java applications with *java YourApplication*

RMI: ***rmiregistry&*** or *start rmiregistry*

java Registry *maxNumberofClients NumberofServers*

start your Java applications with *java YourApplication*

CORBA: ***osagent&*** or *start osagent*

vbj Registry *maxNumberofClients NumberofServers*

start your Java applications with ***vbj*** *YourApplication* and not with *java*

Then you can start your servers. Always start your servers first then start your clients.

Keep in mind, that when using the **Socket** Registry you have to give the **exact number** of server and clients and that in this case clients cannot be disconnected. For RMI and CORBA you have just to give the exeact number of Servers. The value *max-NumberofClients* represents the maximum of Clients that can be connected at a moment.