View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Concordia University Research Repository

An Efficient Run-Time System for Concurrent Programming Language

SHRUTI RATHEE

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

MARCH 2014

© SHRUTI RATHEE, 2014

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **SHRUTI RATHEE**

Entitled: **An Efficient Run-Time System for Concurrent Programming Language**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Sudhir P. Mudur ______________ Chair

Dr. Terry Fancott ______________ Examiner

Dr. Rene Witte ______________ Examiner

Dr. Peter Grogono ______________ Supervisor

Approved by Dr. Sudhir P. Mudur ______________
Chair of Department of Engineering and Computer Science

Dr. Christopher Trueman ______________
Interim Dean of Faculty of Engineering and Computer Science

Date March 2014 ______________

Abstract

An Efficient Run-Time System for Concurrent Programming Language

Shruti Rathee

Our objective is to construct a *Run-time System* for Erasmus. Software systems nowadays are becoming very complex, so the last thing a programmer would want to do is to worry about the internal management of communication. We describe a system that provides Erasmus with a well-defined level of abstraction. This run-time system also provides processes with independence, which means that no two processes know each others' location.

Erasmus is a *process oriented* concurrent programming language being developed by Peter Grogono at Concordia University and Brian Shearing at The Software Factory in England. Erasmus is based mainly on cells, processes, and their interactions through message passing. For every cell there will be a manager to initiate the communication and also to route the communication, whether it is local or remote. Programmers should only be deciding which kind of data they want to send and the rest will be taken care by the run-time system.

For communication within a cell, channels with local communication methods will be used; for communication over the network or on different processors, a broker with remote communication methods will complete the communication link. There is also a separate protocol for a manager process to decide between local and remote communication. This thesis discusses in detail the issues related to process independence. It also explains how processes can be separated from the communication protocols. This run-time system will provide software of greater simplicity and fewer irrelevant details.

Acknowledgments

I would like to acknowledge Dr. Peter Grogono for his pateint guidance , enthusiastic encouragement and useful critiques for my research work. I wish to thank my parents for their support and encouragement throughout my study. Finally, I want to thank Mr. Kasim Doctor for all the support and help for proofreading my thesis.

Contents

Contents v

List of Figures viii

1 Introduction 1

1.1 Concurrent Programming 1

1.2 Shared Memory vs. Message Passing 2

1.2.1 Reasons for choosing message passing 3

1.3 Process independence 3

1.4 Organization of thesis 4

2 Background 6

2.1 Sequential and concurrent languages 6

2.1.1 Communication Sequential Process (CSP) 7

2.1.2 Erlang 9

2.1.3 Java 13

2.1.4 Ada 16

2.2 Paradigms for concurrency 19

2.2.1 Shared memory 19

2.2.2 Transactional memory 20

2.2.3 Message Passing 21

2.3 Synchronous or Asynchronous 23

2.4 Message Passing Syntax 24

3 Erasmus 27

3.1 Design goals 27

3.2 Communication 28

3.3 Special Requirements 31

3.4 Implementation of CSP like languages 33

3.4.1 Occam 33

3.4.2 Joyce 36

3.4.3 Mozart/Oz 37

3.4.4 Limbo 40

3.4.5 Go 41

4 Communication in Erasmus 43

4.1 Design Issues 43

4.2 Existing Resources 45

4.3 Experiment with different resources 47

4.4 Unix as most appropriate and adaptable 48

4.5 Detailed description of implementation 48

4.5.1 Basic message transfer 50

4.5.2 Selection among local and remote 53

4.5.3 Channels and Broker 54

4.6 Problems encountered and their solutions 56

4.6.1 Local versus Remote 56

4.6.2 Distinguishing local and remote 57

4.6.3 Symmetrical selection 58

4.6.4 Concurrent communication 58

4.6.5 Idle processes 59

5 Results 60

5.1 Performance measurements 60

5.2 Comparison with other languages 63

5.2.1 Communication via shared variables 63

5.2.2 Communication via message passing 64

6 Conclusions and Future Work 65

6.1 Conclusions 65

6.2 Future Work 67

Bibliography 68

List of Figures

2.1 CSP Communication between processes 8

2.2 Erlang Communication between processes 10

2.3 Message passing using Process Id's 12

2.4 Communication in Java using rmi with stub and skeleton 15

2.5 Message Passing Syntax 26

3.1 communication in erasmus between client and server 31

3.2 Message passing communication using three ports 38

4.1 Implementation using three ports 50

4.2 Remote Communication Implementation 56

5.1 Performance measurement with varying number of sending process repetition times 61

5.2 Performance measurement with varying number of bytes 62

Chapter 1

Introduction

1.1 Concurrent Programming

In concurrent programming several streams of operations execute together concurrently. A single stream of operations in concurrent programming executes similar to sequential programs but with a key difference being that in concurrent programming those streams are able to communicate with each other as opposed to the sequential programming. Concurrent programs due to their very nature have a number of operations instructions and sequence of such instructions are called as threads and due to this reason sometimes concurrent programming is also known as *multi-threaded programming* [38].

Concurrent programs in general are very difficult to implement due to non-deterministic nature of the program execution which in turn is caused because of the multiplicity of possible inter leavings of the operation among the threads. Also there are multiple number of the threads interacting with each other making it very difficult to analyze the programs [35]. In some languages like Java, thread communication for concurrency is maintained through shared variables and an explicit signaling mechanism. There are possibilities of operation interleaving and the execution of a program being

non-deterministic which means that its very difficult to reproduce program bugs [7]. To consider the concurrent programming example, lets say we have a variable `sr` at two places `A` and `B` and we have to increment `sr`. The preferable steps would then be as follows:

```
fetch sr=0
B fetches sr=0
A computes the value sr++=1
A stores the value 1 in sr
B computes new value sr++=1
B stores the value 1 in sr
```

With the above program, we will get the value of `sr` as 1. In the case that B would have retrieved the value after A was done, `sr` would end up being 2 and this is what we call as a *race condition* [38]. To avoid this situation the best thing is to have a locking mechanism on the objects. So the data object that is locked by a thread cannot be accessed or modified by any other thread. In Java locking is used for interference prevention and only the operations declared as synchronized are inhibited from execution during locking of an object.

1.2 Shared Memory vs. Message Passing

Shared memory allows processes to read and write the data from the same location, that is, they share a common address space, while in message passing, processes send messages to each other. Both of these methods have their own advantages and disadvantages. In shared memory there are no delays on the other hand message passing can cause delays. In shared memory whenever any process writes a value at any address it can override the contents assuming no locking but this is not in the case in message passing. In asynchronous message passing there are chances that if

nobody is receiving a message on the other side, there will be a buffer overflow and messages will be lost. Also, there are no common distributed memory platforms that exist for the shared memory but for message passing libraries have been available since 1980s. There is also a standard for message passing, which is the *Message Passing Interface* (MPI) [36].

1.2.1 Reasons for choosing message passing

Message passing was chosen for our project because it gives us more control over process communication. In message passing the client will send message to a server and the server will reply to the client rather than invoking a method. Unlike method invocation in which concurrency cannot be achieved, message passing allows the client to send a message and instead of waiting for a reply, the client can execute other tasks and later on can collect results from the server at any time. This has an advantage that it does not disturb the entire communication process [1].

In shared memory one of the biggest problems is race condition. So if we have two processes `P1` and `P2` and process `P2` is trying to do an operation on a variable for instance and at the same time `P1` changes it then this can lead to a race condition and can cause non-determinism or in worst case it can lead to incorrect results, as shown in Section 1.1. In shared memory this can be avoided by using locks or creating cache coherence but then it has a possible disadvantage of consuming excess of resources, which can be a bottleneck for the communication at a later stage. So by keeping these aspects in mind message passing was chosen.

1.3 Process independence

Our choice of the communication model is message passing, for the reasons explained above. The communication is assumed to happen between distributed processes and

thus location transparency is important. This means processes will operate independently of their location. Message passing is our main communication method and it should be achieved in a way that no two processes know each others location. But to send data from one process to another they need to know their address or memory location, hence location transparency is very difficult to attain.

We proposed a solution to use a manager, channel and broker for the communication. In this proposed model, processes do not communicate directly instead they will communicate through a channel in the case of local communication and through a broker in the case of remote communication.

Manager is responsible for providing the address of the process to the channel or broker and then either of them will communicate to the process on the other side.

Both the broker and the channel have similar configurations in terms of communication protocols with a key point being that the broker will be used on the network and the channel for local communication. Two brokers on the network can communicate with each other. When a process wants to communicate with another process, the manager creates the instance of that process and checks whether that process is local or remote. Accordingly, it sends a request to either channel or broker.

A broker is just like a process that is used to assign ports and initiate communication. For this reason, this forms an integral part of our solution to the problem at hand since the interface of the broker is the same as a process and hence communicating processes think they are interfacing with another process while at the same time inducing distributed behavior.

1.4 Organization of thesis

In Chapter 2 we will have a look at some of the sequential languages that don't support concurrency but are widely used for implementing concurrency. Also we will

discuss about some concurrent languages that will be used later to compare with Erasmus in Section 5. Then we will have a thorough examination of the paradigms that give rise to concurrency with some examples and which ones are suitable for differing situations. We will go on further to do a discussion on the syntax of message passing mechanisms in various programming languages. This will be followed by some more discussion about how these languages decide whether to do a remote or local invocation when talking about communication protocols.

In Chapter 3 we discuss Erasmus, the language for which this communication model is being implemented. Also we take a look at the design goals for Erasmus and then go on to explain how communication takes place in Erasmus. After that we explore if there are any special compilation requirements in Erasmus for the deployment of the programs from one platform to another. Furthermore we do a discussion on the CSP-like languages and will explain some languages in detail with their implementations.

Chapter 4 looks at the design process in the implementation of communication in Erasmus and in doing so we will discuss about the experiments we did with different types of resources and about the problems that we encountered during implementation. We will then explain briefly why we chose Unix as an appropriate platform for development. Besides, we will provide details of our implementation.

In chapter 5 we take a note on the performance measurement and compare this communication implementation with some other languages of interest, which we have mentioned in Chapter 2 and Chapter 5.

Chapter 6 summarizes the work we have done and new prospects for future work.

Chapter 2

Background

2.1 Sequential and concurrent languages

Sequential programming languages are languages in which sequences of the statements are executed one by one on a single processor. These types of languages differ from concurrent programming as in the latter the sequence statements may be executed concurrently that is at the same time. Although some of the sequential language provide some concurrent features for increased efficiency. Such languages include C, C++, JAVA, Erlang [11].

We have already discussed concurrency in Section 1.1. We have a vast number of concurrent languages that can be divided into three classes, namely, procedure oriented, message oriented, and operation oriented [8]. In *Message oriented* languages processes communicate using send and receive operations. These types of languages do not have any shared objects so the processes have no access to any object directly. But, there is a third-party known as the manager, which takes care of objects. Whenever any action is required message is sent to manager to process it and on completion manager sends a reply confirmation [8].

Objects in message-oriented languages are never concurrently or directly accessed

by other objects. Some examples of these types of languages are Gypsy and PLITS [8]. A second category of languages is *Procedure oriented* in which processes communicate by shared variables. Unlike message oriented they have active, passive and shared objects. In this class of languages, programs access the objects directly and procedures are called for performing the operations. Mutual exclusion must be ensured for shared objects. The languages in this category are concurrent PASCAL, Modula, Mesa and Edison. A third class is the *Operation oriented* with the processes interacting using remote procedure calls. These types of languages are combination of message oriented and procedure oriented. In this the process calling the operation synchronizes while implementing the operation and then later continues asynchronously. Examples of such languages are Distributed Processes, StarMod, Ada, and SR [8].

2.1.1 Communication Sequential Process (CSP)

Hoare first described CSP in his paper in 1978 [19]. It is known as a formal language, used for description of communication patterns in concurrent systems. In Original CSP, programs were parallel composition of some sequential programs and were able to communicate with each other. All the processes in CSP start together and are handled in parallel. Below is an example of such a parallel command:

```
parallel= *[a:character; process1?a -> process2!a]
```

This is a repeating process; repetition follows by receiving a character from `process1` and sending it to `process2`. The example for parallel command for concurrent execution is:

```
[S :: CL || R :: CL]
```

The sending process would name the receiving process as its destination and receiving process will name sending process as its source. The resulting communication that takes place is synchronous, so both the sending and receiving processes should be ready at the time of communication. In CSP we have semantics and type matching.

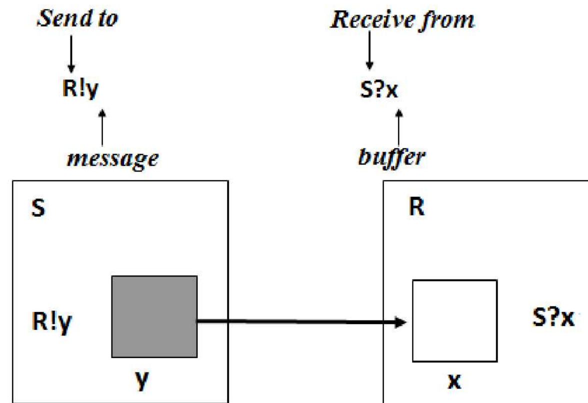

Figure 2.1: CSP Communication between processes

The type of the message sent by the sender should match the type expected by the receiver. To show how communication takes place let us take two processes `S?x` and `R!y` . Process `R` will name `S` as its source and process `S` will name `R` as its destination. When `R` will execute `S?x` the value `x` will be read from `S`. Same when `S` will execute `R!y` it will write `y` to process `R`.

CSP supports concurrent programming and can have problems like non-determinism, which we have already discussed in Section 1.1. To avoid non-determinism it uses guarded statements [29]. These guarded statements can be considered as the conditional statements which are series of Boolean expressions before $\rightarrow$. It has a subcategory too known as the alternative commands. These are formed by the concatenation of the guard statements with []. To run the alternative commands system will search for the guarded clause of a true conditioned statement. The statements can be represented as:

```
[ X ? k -> Sx
[]
Y ? k -> Sy
[]
Z ? k -> Sz ]
```

The above is an example of guarded command with three processes `X`, `Y`, `Z`. In CSP, `X?k` means *read* `k`. Whenever a process is ready it will write its value into variable `k` and will execute.

For concurrent communication with guards there can be three possibilities. First possibility is when a process is ready and will write the variable as discussed above. Second, when no process is ready it will iterate through processes `X`, `Y`, `Z` till the time it will find at least one ready process to communicate. We will put * before the communication block for iteration. A third possibility exists when more than one process is ready to communicate. A process will be chosen non-deterministically to start communication.

2.1.2 Erlang

Erlang was developed in Ericsson labs. The computer science lab at Ericsson wanted to develop a language that was suitable for developing next generation products for telecom. They found lots of languages but none had all the required features in one, so they developed their own language. The main contribution in the success of Erlang is its concurrency feature. In Erlang processes execute in their own memory space and have their own heaps and stacks. Communication in Erlang is performed using message passing in an asynchronous way [31]. Even if a receiving process is not ready to receive the sending process can send the message and continue processing requests from other processes. As sender is not blocked after sending message. Erlang has been made with location transparency in mind which means that for the programmer there is no difference if receiver is on same processor or different. By default the distributed programming model in Erlang has been implemented using TCP/IP [20].

In the Figure 2.2 we have communication between the server and the client. They use TCP/IP protocol; client sends the request to the server for calculating the volume of a square box. The client sends two attributes in the message, the volume it wants

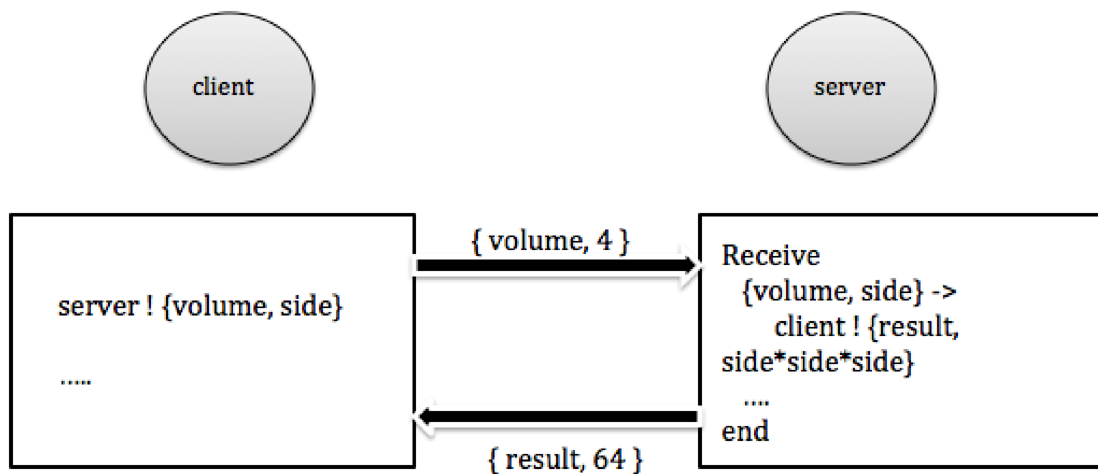

Figure 2.2: Erlang Communication between processes

to get and the side of the square to calculate the volume. On the server the function to calculate the volume will use the value `4` for side provided by the client. It then replies back with the calculated result.

Erlang is highly influenced by languages such as Prolog, LISP and SmallTalk, which explain its features of sequential programming, concurrent programming as well as functional programming. First we go on to discuss sequential programming methods and features. One of the features of sequential programming in Erlang is `case` construct, which relies on pattern matching. We will take an example to show `case` construct that will evaluate an expression and will match the result against list of matching clauses.

```
case lists:member(foo,Check_List) of
  true -> ok;
  false -> {error,unknown_element}
end
```

In the part of code above with case construct we check if `foo` is a member of list

named `Check_List`. If it is a member then atom `ok` will be returned otherwise a tuple `{error,unknown_element}` will be returned.

In addition to case construct we can also implement Guards which can be placed either in case or receive clause in Erlang. Let us take an example of calculating the factorial of a number with and without guards.

```
factorial(0) -> 1;
factorial(N) ->
   N * factorial(N-1)
```

In the above code we always need the `factorial(0)` clause in order to make sure that the function will terminate, but if we use guards for the same purpose we only have to select the recursive clause which will work with the condition that `N` is greater than `0`. The code below demonstrates that:

```
factorial(N) when N>0 ->
   N*factorial(N-1);
factorial(0) ->1
```

The execution of a process in Erlang is termed as an activity and if those activities run concurrently to each other they are termed as processes. They communicate with each other using message passing and this forms the basis of concurrent programming in Erlang. Messages are sent using their process identifiers [6]. If we want to send a message of any data type we can send it using the statement:

```
Pid ! Message
```

All processes in Erlang have their own mailbox in which a message will be stored. When a message is sent it is actually copied to the receivers mailbox. Erlang ensures that messages are stored in the receiver's mailbox in the order in which they are received. However, the receiver does not have to handle them in that order. This

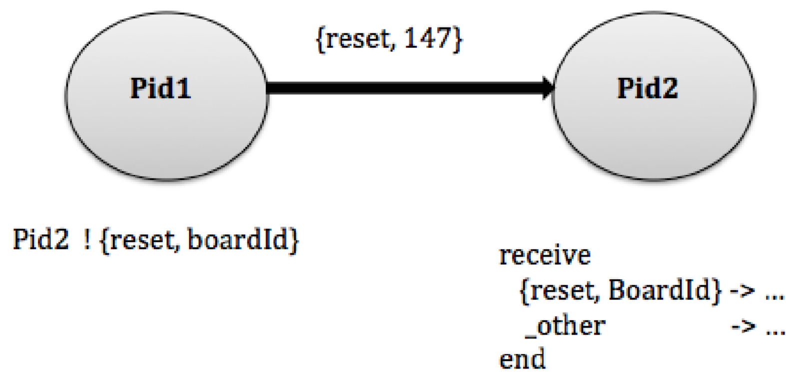

Figure 2.3: Message passing using Process Id's

is because the receiver can use pattern matching to choose messages, and the implementation will look for a matching message, not necessarily the first message in the mailbox. Erlang has asynchronous message passing so sending processes never waits, they just execute the next available problem statement while receiving process may wait if its mailbox is empty. Even if a message is sent to a process that does not exist, message sending will not fail. For receiving messages it uses the receive clause. Receive works like:

```
receive
   {reset, Board} -> reset(Board);
   _other -> {error,unknown_msg}
end
```

In the Figure 2.3 a message is sent as an integer number to the process executing `receive`. First the pattern will be matched so that the integer will be bound to the variable `Board` and at that time the function `reset` will be called.

2.1.3 Java

Java [37] was developed at Sun Microsystems by James Gosling with multi paradigm features such as object-oriented, structured, sequential and others. Java is considered as one of the most famous languages of the 2000s, which is contributed by its feature called WORA or "write once read anywhere". When java programs are compiled they produce a machine independent code called byte-code. They can be executed on any system running JVM (Java Virtual Machine) independent of the architecture of the system. In Java threads and processes support concurrency. In java 1.5 concurrency can be achieved by using the package `java.util.concurrent`. Threads communicate using shared variables that can create problems like race conditions. In order to circumvent these problems Java supports locks and synchronization of the threads. For implementing synchronization we use `synchronized` keyword, which ensures that only one thread at a time is executed. This can be done in code by using the `synchronized` keyword with the method name.

```
public synchronized void method1(){
   //definition here
}
```

In Java, threads are non-deterministic in nature so every time a program is run, different behavior may be seen. There are two ways of creating a thread, one by implementing the `runnable` interface and another by extending the `thread` class. The method of implementing `Runnable` is considered as the easy one because we just have to implement only one method `run ()` and a class implementing `runnable`. After all this we can initiate the object of the thread and use another method to start the thread.

```
public void run()
Thread(Runnable objthread, String namethread);
```

```
void start();
```

The other method requires the class to extend the Thread class and then override the run method in it. This is not very helpful since Java does not allow multiple inheritances and thus makes it difficult to extend any other behavior.

```
public class FirstThread extends Thread
{
  public void run()
  {
    for (int i=1; i<=5; i++)
    {
     System.out.println( "Message from First Thread : " +i);
     try
       {
         Thread.sleep(1000);
        }
     catch (InterruptedException interruptedException)
        {
       System.out.println( "First Thread is interrupted when it
                                     is sleeping"+interruptedException);
         }
     }
   }
  }
```

Now the main class to start the thread:

```
public class ThreadDemo
  {
```

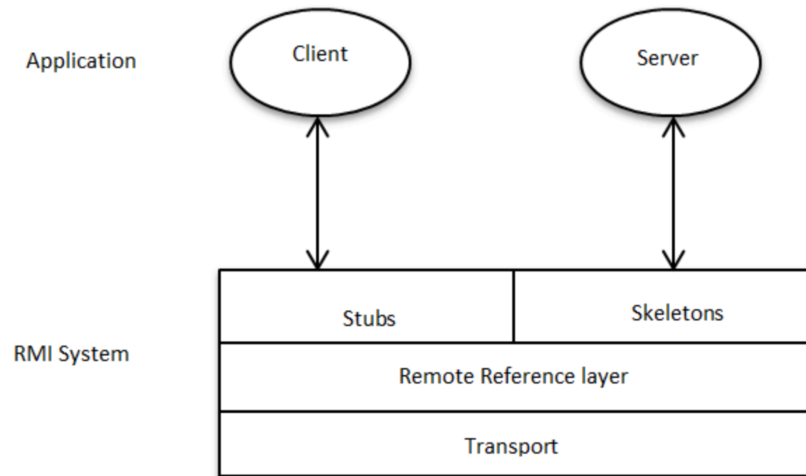

Figure 2.4: Communication in Java using rmi with stub and skeleton

```
public static void main(String args[])
 {
    FirstThread firstThread = new FirstThread();
    firstThread.start();
} }
```

In the above examples threads have been used for communication.

In Java distributed programming is supported with the help of *Remote Method Invocation* (RMI) [26]. It is basically an extension of local method invocation applied over a network. When two Java virtual machines want to communicate, RMI will initiate the object calls between them. A typical RMI always has two parts: a *stub* and a *skeleton*. The communication flow proceeds as shown Figure 2.4. Stub is the object on the client side. Whenever a client makes a call for an object it will communicate with the stub. Skeleton serves the same purpose for the server side and hence the stub communicates with the skeleton in order for communication to proceed.

Having supporting the distributed concepts of communication Java has location transparency and to get references for communication it uses a service called as naming service. Java uses an interface that has all the methods, which can be called by the objects. Below is an example of such an interface:

```
package montrealServer;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface montrealServer extends Remote {
   <T> T startExecute(Task<T> t) throws RemoteException;
}
```

In the code above `montrealServer` is extending the interface `Remote`. Extending the `Remote` interface will let the JVM recognize it as RMI ready and it can be invoked by any other JVM. In Java, object serialization is used for transporting the object between java virtual machines. For the use of serialization the class is implemented as `java.io.Serializable`.

2.1.4 Ada

A team at CII-Honeywell-Bul led by Ichbiah developed Ada in 1980s but later in 1990s it was revised under the advisory of Tucker and was released as an internationally standardized object oriented language known as ADA 95 [24]. It is highly influenced from languages like Algol68, C++, Smalltalk and Pascal. Ada was developed for the design of very large software systems with various features. Among those features are compile time as well as run time checks to check all kind of possible bugs and errors. It has dynamic memory management, automatic garbage collection and its syntax is similar to the English language. It allows users to define their own data types. These are examples of data types: there are many more.

```
type new_data is range 1 .. 42;
type another_data is range 1 .. 29;
type third_data is mod 67;
```

In Ada the basic unit of concurrency is a task that is also a type. It supports concurrent tasks, which are also sometimes synchronized. Ada uses the rendezvous for communication that has similarity to the object calling in object oriented programming languages [30]. There is usually a call statement that will call the entry using the name of the task directly and the task that responds will execute an accept statement that is defined for it. This is very similar to the client-server model in which a client will make a request and the server will complete the request using a service among the pool of available services. Let us take an example of an online mobile recharge system in which the website will have some predefined tasks which any client can use. Below is an example of a task:

```
task type mobile is
   entry Recharge (PhoneNo :in Phone; Amount: in Money;
      Result:out Boolean );
   entry Message (PhoneNo :in Phone; Amount: in Money;
      Result:out Boolean );
   entry changePlan (PhoneNo :in Phone; Plan: in Name;
      Result:out Boolean );
end mobile;
Website : mobile; -- creates a mobile task
```

In the above code `Phone` and `Money` refers to the predefined types. There are three different types of functions created inside the server task, which can be requested by the client. The call by Client for the task is:

```
-- customer task
```

```
Website.Recharge(09487652802, 100);
```

When this task is requested, the website task will process the request and will call the function to recharge the mobile phone. In these kinds of tasks the client and server both have to wait if no task is being serviced or being asked by any client. Both the client and the server should be ready to communicate to create a rendezvous. Like many other languages Ada also provides the `select()` statement by which any task can choose which request it wants to process. The syntax for selection is:

```
select
  accept ...; -- entry point logic
or
  accept ...; -- entry point logic
or
  accept ...; -- entry point logic
end select;
```

Or we can use `select` with the guard statement like:

```
select
  when <BOOLEAN condition> =>
  accept ...; -- entry point logic
or
  when <BOOLEAN condition> =>
  accept ...; -- entry point logic
or
  when <BOOLEAN condition> =>
  accept ...; -- entry point logic
end select;
```

2.2 Paradigms for concurrency

We have already discussed concurrency in Section 1.1. We will now have a look at its paradigms.

2.2.1 Shared memory

Shared memory is the fastest form of inter-process communication. Processes share memory, which can be accessed simultaneously. This memory provides a very fast way to communicate and avoids the problem of redundant copies. All the processes share the same view of data making them easy to program. Apart from being the fastest and efficient way of communication it is limited to the same machine. If two processes share memory are on different processors they should be cache coherent. This implies that if any process makes a change to some data, that change is immediately reflected for all the other processes or there is a chance of conflicting data [2].

Race condition is another problem that can occur during simultaneous access for writing. In race condition the order of execution of the process actions affects the output of the process [3]. As discussed before in Section 1.1 race condition and cache coherence in shared memory, there is a need of synchronization that can be achieved by using one of the following methods, namely semaphores, mutex, read-write locks and condition variables. Let us take an example of how communication takes place-assuming use of semaphores on the server [22]:

1. Server gets access of the memory object (shared) using semaphores.
2. Server will read the input from the input file to the memory object.
3. After read completion the server notifies client.
4. The client will take the data from same memory object and will write it to the output file.

During communication the data will be transferred for a total of two times with

synchronized communication using semaphores.

2.2.2 Transactional memory

Transactional memory is a model that controls the memory access in concurrent programming. In concurrent programming it should always be ensured that no two processes change the same memory space at same time otherwise we can have race condition. We use locks or other methods for the synchronization of the communication. Transactional memory can be used as an alternative to locks. In this approach fragments of the programs are marked as transactions and are executed atomically. The threads in the program execute the transactions so if at any time two threads access the same memory and have a conflict the transaction will be aborted. Transactions in transactional memory should be executed serially so that no two processes should be interleaving each others steps. There is also atomicity, which means that the processes have a sequence of the transaction and after completion, it can be flagged either as a commit or abort. As discussed, it is a way of controlling the memory access and hence has some defined ways of accessing the memory, some of which are `Load-Transactional (LT)`, `Load-Transactional-exclusive (LTX)`, and `Store Transactional (ST)`. They all differ in how they access the memory and perform transactions. `LT` reads value from a shared memory to register, `LTX` reads value from shared memory to register but with a hint about memory update and `ST` writes a register to shared memory [21].

It is important to know about the states of the transaction and for that there are some instructions such as `commit` to make the changes and inform all the other processes about the changes to avoid a race condition. Another instruction, `abort`, is used to discard all the changes and the changes that are usually discarded are the ones for which the location with which a commit has been performed has been already accessed by another process. The instruction `validate` informs whether a

transaction has been aborted or not by returning a Boolean value, which is false in case if the transaction was aborted. An example follows:

1. Read from location (shared memory) using Load-Transaction exclusive or Load Transactional.

2. Validate read using Validate instruction to return True or False.

3. Copy the value read using Store transactional.

4. At last do a commit (instruction) to make changes that are visible to other processes.

2.2.3 Message Passing

Message passing is another way of concurrent programming for inter-process communication. Two basic functions of message passing are `send` and `receive`. It can either be synchronous or asynchronous. Synchronous message passing works with a blocking send and receive. Sender will be blocked until receiving process is performing receive function and receiver will be suspended until a message sent is to be received. It is very easy to implement. Sending and receiving have no overhead because messages are not buffered. While in asynchronous mode the sender is not blocked and if no one is ready to accept the message, it is queued at the receiver side but if the receiver becomes ready to receive and no messages have been sent then the receiver will be blocked. On the other hand, there is another concept of message passing called rendezvous mode, which is a synchronous request reply service. Asynchronous and synchronous message passing are one-way communication that is from sender to receiver but rendezvous mode is two way communication. When in client-server receiver is responsible for sending a reply message rendezvous mode is used. The client is blocked until server sends a reply [25].

In message passing all the processes have their own local memory and they use messages to communicate with other processes. To make things more clear we can tie

up the three models of message passing with three different types of communications likely channels can be represented by synchronous message passing, ports by asynchronous message passing and entry by rendezvous mode. We will take small pseudo codes to explain them all one by one. Below the first pseudo code is for channels doing synchronous message passing.

```
Channel ch = new Channel();
tp.start(new Sender(ch,sendng));
rp.start(new reciever(ch,recvng));
```

At one time there can be more than one channel, which can be ready to communicate, so for these type of issues we can use selective receives. We will use select() for choosing between channels and channel will start the communication when both sender and receiver is ready.

Next is port that can be associated with asynchronous message passing. This is a many to one communication so two senders or more will communicate with a receiver by the use of unbounded port. In this type of communication sending process is never blocked and receiving process should not necessarily be ready.

```
Port pt = new Port();
tp1.start(new asysend(pt,sendng1));
tp2.start(new asysend(pt,sendng2));
rp.start(new asyrecv(pt,recvng));
```

Last one is entry, which is used as a rendezvous communication. We will have more than one client requesting for services but only one will get the entry and the receiver will reply to the sender after receiving the message.

```
Entry ent = new Entry();
tp.start(new Client(ent,sendng,"Q"));
tp.start(new Client(ent,sendng2,"w"));
```

```
rp.start(new Server(ent,recvng));
```

In all the above listings of the pseudo codes `tp`, `tp1`, `tp2` are the instances of the threads, `sendng`, `recvng` are the instances for the slots.

2.3 Synchronous or Asynchronous

Both synchronous and asynchronous are equally good during implementation, each of them have their own share of advantages and disadvantages. For instance in synchronous no buffer space is required, there is no overhead, and at any particular time only one message is overdue. Sender always has a confirmation of successful communication [16]. But it can be a disadvantage too; the sending process can't do anything but just keep on waiting for the confirmation receipt. In asynchronous we have the advantage of increased concurrency and receiver and sender can do another tasks while the communication is in progress. Receiver can periodically check the buffer for messages at certain intervals of time. But sometimes it can be a bottleneck as buffer can overflow and we can lose some messages. A good example for synchronous communication is a real time chat server while email services can serve a good example for asynchronous communication.

So, it's safe to say that the choice depends on the requirements; in our system we have decided to choose synchronous communication. We are using channels and whenever a process requests a service, that request is sent to a channel and then waits for the channel to respond. Channel will then send that request to the server; if server is busy it will respond with a busy message otherwise it will send the message of processing and communication starts. Our processes communicate indirectly in a synchronous way and only one process is processed at a time by any channel. All similar technologies like RMI, RPC and CORBA are based on synchronous communication.

2.4 Message Passing Syntax

The syntax for message passing depends on the programming language and the libraries that we use for the communication. In message passing the sender sends a message to the receiver that can be considered as an object and all objects are different from each other. There are three things to consider in the syntax for message passing, namely, receiver, message selector and arguments. In Section 2.2.3 above, we discussed about different kinds of message passing classes that affect the syntax of message passing. Some languages use message passing for distinguishing between local and remote invocation. Java supports message passing and it uses a channel for the distinction between local and remote, which is well known as remote method invocation [10]. Below Figure 2.5 is giving a small overview of the methods for communication in different languages.

First we have Java, which is among the most famous and most widely used programming language. Java due to its vast architecture has different syntax for local and remote invocation. For local communication it uses shared variables. As there are many disadvantages to this technique, it uses locks to synchronize processes to do a task without any non-deterministic behavior. When talking about remote communication it uses `RMI (remote method invocation)`, which we have already explained in detail in Section 2.1.3.

Then comes Erlang, which is well known for its concurrency. When talking about local object calling it uses asynchronous message passing. It uses the `PID (process identifier)` for identifying the receiving end. It uses asynchronous methods for all the sending and therefore the receiving processes have mailboxes. Data is copied from sender to receiver's mailbox during communication. For remote communication by default it uses TCP/IP ports over the network.

CSP (communication sequential process) uses synchronous message passing for

its local communication. It has indirect naming channels for remote communication [34].

Joyce programming language is influenced by CSP and likewise it also uses channels for synchronous message passing for local as well as remote communication. But in contrast to its descendant languages its channels are bidirectional.

Occam is among the very few languages that use same syntax for local as well as remote communication. Although for local communication it uses unidirectional synchronous message passing, for remote communication it uses single ended channels that are called ports but they both use the same interface.

Go is a new language and is very close to what we have in Erasmus. Although it uses message passing but there is a feature in the programming language that while doing communication we have to create a buffer for the channels and that buffer value will decide that if it will use synchronous or asynchronous communication. For network communication it uses network channels and a pre-defined library of its own known as netchan. It has many other methods for remote communication.

Mozart is the programming language with multi paradigm feature. For concurrency it uses threads and for communication it uses message passing. It is a concurrent language so its model is often mentioned as concurrent message passing model in which we use ports for channel communication.

Language	Local	Remote
Java [26]	Shared Variables, Locks, Synchronization	Remote method invocation with stub and skeleton
Erlang [20]	Asynchronous message passing with PID of messages	TCP/IP Ports
CSP [19]	synchronous message passing	indirect naming via communication channel
Joyce [15]	synchronous bidirectional channels	synchronous bidirectional channels
Occam [4]	unidirectional synchronous message passing	single ended channel called ports
Go [5]	message passing with both asynchronous and synchronous	network channels called netchan
Mozart [18]	Message passing using ports over channels	Message passing using ports over channels

Figure 2.5: Message Passing Syntax

Chapter 3

Erasmus

3.1 Design goals

These are the main goals of the Erasmus project [33]:

1. Level of abstraction should be more defined, as programmers do not necessarily have the details of how the process communication is taking place and how they are mapped to processor at run time. For example, feasibility of modification of Fat client to thin client should be there without the modification of code.

2. Instead of having different entities for everything we can just have a fixed entity that can be used from time to time at all the scales in the software.

3. A language should have facility for interfaces in it so that software can be built with interfaces and when all these interfaces are combined with different specifications, it gives a complete solution of the software. It gives software more flexibility for future.

4. Encapsulation should be implemented in a language. In Erasmus we have encapsulation property by having cells that have processes in them which are isolated from each other. Processes inside cells can communicate to each other

through shared variables and cells can only communicate by synchronous message passing.

5. For any software one thing that can be a big challenge in the long term is refactoring. As software grows it can be very costly for an organization to refactor software each time they want it to release it on some new platform. So a language should be easy to refactor. In Erasmus we have separated the compilation and the deployment. Any software that is developed in Erasmus can be very easily transferred from one platform to another and refactored.

6. A language should be type safe so that it can detect all kinds of errors and doesn't give problems later in the software by giving non-deterministic bugs. In Erasmus we have static type checking for language safety.

7. Programmers today should be more concerned about adding functionality rather than to worry about adding an additional process tomorrow.

3.2 Communication

Erasmus uses `p.m:=e` to send message `m` on port `p` and `x:=p.m` to receive message `m` on port `p`. In these statements, `p` is a port, `m` is a message, `e` is an expression (rvalue), and `x` is a variable (lvalue). For running a process it should be deemed as runnable. Processors that are not running are stored in queues. Usually we can have either a global queue for all ready processes or we can have separate queues for different memory partitions. But both can be tricky when considering different situations, and in some cases their combination could prove to be better. There can be two queues so that if a process is ready to execute it is in the ready queue and if it is blocked we can transfer it to another queue [11].

Erasmus in contrast to other programming languages communicates with processes

using message passing. The structural unit of Erasmus is called a *cell*. Cells define the boundaries of the processes and the control flow is always inside this defined boundary. Cells can exchange data but cannot invoke methods in one another. In Erasmus processes are linked using channels that in turn are associated with a protocol that decides what kind of message one can send and in a given order [1]. Cells own all the channels and each channel has its own global unique identifier. For the structure of the channels they have protocol and fields. In pseudocode, we use `P` to denote a protocol, `p` to denote a port, and `f` to denote a field. Another entity needed is the reader and writer queue so that we can assign them as readers and writers. There are queues for `p.f` (field of protocol `P`) as `p.f.writers` and `p.f.readers`. There is also a queue for each field of protocol. In Erasmus when a process is in a queue it will be blocked until it can receive data using its field protocol. Similarly it will be used for writing that the process will be blocked until it writes using its field protocol. Let us first consider a simple message transfer in Erasmus:

It is important to recall that in order to receive a message, port expressions are usually used as an rvalue. So if a process ”rho” wants to read something it will do so as:

```
v := p.f
```

`v` is the variable represented as lvalue. So for message transfer there can be two cases, first one that no process is ready to send any information and second one when a process wants to send information. If the process is ready to send information then it should be in writers queue. For the first case it can be said that no writer is waiting, that is, `p.f.writers` queue is empty, so we can put that process in the readers queue and the process will be removed from the ready queue. In the second case we can remove the first entry from the writers queue and the process ready will be put into the ready queue. The pseudo code for all this is:

```
if empty p.f.writers then
```

```
        add(rho,v) to p.f.readers
    else
        (sigma,e) := first p.f.writers
        v := e
        resume sigma -- become runnable
```

It will be same for the writing process too except the fact that the write operation only waits if reader is not ready, while read operation always waits even after storage of lvalue. The pseudo code is given as:

```
    p.f := e -- e can be an expression or a rvalue
    if empty p.f.readers then
        add(rho,v) to p.f.writers
    else
        (sigma,e) := first p.f.readers
        v := e
        resume rho -- become runnable
```

Erasmus includes the functionality that allows us to choose the channel to communicate on using a `select` statement [9]. "In Erasmus there is a condition that at most only one process connected to a channel may access the channel with a select statement" [1] [11]. With a `select` statement there can be one of two cases that either a process is ready to send/receive or there is no process ready at that moment. In the second case we set the program counter back to the beginning of the `select` statement and restart the process of checking whether any channel has a process in its readers queue or not. If we have a request to process it will be completed normally and if we have more than one requests at a time then the choice will be made using any of the three policies, namely, `ordered`, `fair` or `random`. Execution with select pseudo code:

[1] This restriction has been removed in newer versions of Erasmus.

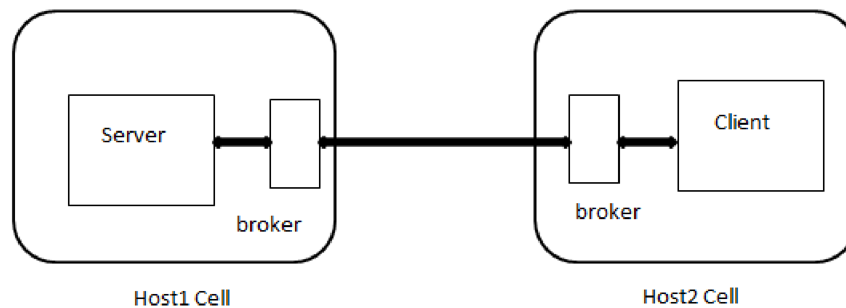

Figure 3.1: communication in erasmus between client and server

```
Select
  if (ch : writers(ready))
    read (p.f)
  else
    reset select();
```

3.3 Special Requirements

Software nowadays is very expensive to create or maintain and refactoring applications time to time with environment changes can be very expensive. One of the goals of Erasmus is to separate the deployment of a program from its compilation. Lameed (2008) demonstrated one way of achieving this goal. We do external mapping of the cells by which we can easily decide which cell to map on what processor reducing the communication time for the processes communicating too often. We can map cells on the processors that are closer to the scheduler to make tasks fast. Erasmus programs provide portability for the programs without changing them but we should have a separate configuration file for mapping cells to the processors. Lets take an example

of a simple program:

```
sqProt = [*(query: Float; ^reply: Text)];
square = { p+:sqProt |
  loop
    q: Float := p.query;
    p.reply := text (q*q);
  end
  };
```

When we compile the above program, it will generate the code for the uniprocessor but if we use the configuration file below it allows us to configure its cells on different processors. The configuration file in Erasmus is a valid XML file with some specific tags. We used XML because it is widely used by most of the programmers and easy to create and map. Below are the example tags we use in the XML file.

```
<Mapping>
  <Processor>spainish.encs.concordia.ca
    <Port>4534</Port>
    <Cell>firstcell<Cell>
    <Cell>secondcell<Cell>
  </Processor>
</Mapping>
```

There are only four tags so mapping is easy and fast. Whenever a program is compiled the compiler extracts the data from the XML file and organizes it into a table thus generating unique identification for all the cells. The information retrieved is about processor, cell name, port number and the data is appended to a text file, which is eventually read by the processor. If no entry is found then it means that the cell has

not been mapped yet and the default value of 0 is used for the local host and port number.

Although Lameed's implementation was completed and tested, it was an "existence proof" rather than a practical system. Consequently, Lameed's compiler was not incorporated into the mainstream of Erasmus development.

3.4 Implementation of CSP like languages

Hoare first described CSP in 1978 [19]. CSP stands for *Communicating Sequential Processes* and it describes how patterns interact in concurrent systems. Languages like Occam, Go, and Limbo are highly influenced by CSP. Go programming language took features of Newsqueak and Limbo (concurrency) from CSP [27]. Erlang is another language that is influenced by the CSP methodology. Armstrong, the creator of Erlang developed a new term, *Concurrency Oriented Programming* which represented the languages that used concurrency as one of their paradigms [23]. According to Hoare, in real life human beings work in a concurrent way so its natural to have programming languages that do the same and hence this paradigm of concurrency was influenced from CSP. Let's take few examples of CSP like languages.

3.4.1 Occam

David May developed Occam a parallel programming paradigm language at Inmos Limited, Bristol, England [12]. It was originally developed to program transputers. Its name has been derived from a thirteenth century philosopher known as William of Occam. Like many other languages it is based on the CSP of Tony Hoare. Researchers in the Inmos first developed an Occam concurrency model that led to the development of the programming language Occam [4]. Concurrent programming in

Occam is also based on message passing. It does not have any shared variables and the communication between concurrent processes is done with channels. Communication mechanism in Occam is one-directional, point-to-point and un-buffered. So if information needs to be sent to several processes together more channels will be needed and the same applies in the case of bi-directional communication. In Occam the syntax used for concurrent programs is [13]:

```
PAR
    ......Write code for the first process
    ......Write code for the second process
    // to give priority to some processes we need the syntax like:
PRI PAR
    ......Write code for the high priority process
    ......Write code for the low priority process
```

As shown in above code when there are more than one process priority code is written for them and priorities will be only served if the processes above have either of the states viz. terminated, stopped or waiting.

Two processes should be the components of a parallel program to pass the information to each other on a channel.

```
CHAN OF INT pipep:
PAR -- major process
  SEQ -- minor process 1
      .....
      pipep ! 5
      .....
      INT acc: -- minor process 2
   SEQ
```

```
.....
pipep ? acc
.....
```

In the above listing functioning of the process `pipep ! 5` and `pipep ? acc` are similar to a `process1` copying the value of 5 into the variable `acc` of `process2` and both the actions are synchronized. Both `send` and `recv` should be ready at the same time for communication. Although it is a very efficient method of communication in Occam but there are few limitations to this method such as for one component process a channel can only be used for either input or output communication. Another limitation is that one channel can be only used between two processes in parallel; more than two processes need more channels for communication. We can however accomplish this by adding more channels like `CHAN OF INT pipep, chnl`, which will declare channels `pipep` and `chnl` with same protocol. Here we have used `INT` (integer) protocol.

For communication with other processors Occam uses ports and they are called '*single-ended channel'* in Occam. The declaration of port and way of communication is same as that of a channel.

```
PORT OF INT sgnl.in:
PORT OF [12]BYTE beta.out:
  SEQ
    INT vrble:
  SEQ
    sgnl.in ? vrble
    ...
    beta.out ! "Good Morning"
```

Although input and output methods are similar to the channel, they are more restricted than a channel in few aspects. For the declaration of a port we cannot use

a named protocol or a sequential protocol. They can only handle the data types of real, integer, Boolean and their respective arrays.

3.4.1.1 Occam-pi

Occam-pi is a newer version of Occam language implemented by KRoC, the Kent Retargetable Occam Compiler. The symbol 'pi' is used for naming because it has several ideas inspired from Pi-calculus [14]. It has a number of extensions to Occam's previous versions like nested protocols, run-time process creation, protocol inheritance, data, processes, recursion, array constructors, mobile channels and extended rendezvous.

3.4.2 Joyce

Joyce [15] is a secure programming language based on CSP and Pascal. Per Brinch Hansen designed it in 1980s. It consists of the nested procedures that define a communicating agent. An initial agent is activated when a program starts its execution and "agents" can dynamically activate subagents. Variables of an agent are inaccessible by other agent. However they can communicate by transmitting the symbols through channels. On channel, communication will take place only once, however more than one agent can use the same channel and transfer symbols in both the directions. Communication in Joyce takes place when two processes are ready to send and receive a symbol through same channel i.e., synchronous. Processes can create channels dynamically and can access them through the local port variables. Whenever a channel is created its pointer is assigned to a port variable and that pointer can be passed to the subagents. When an agent has completed its procedures it will wait for the subagents to complete and it will then terminate. All the channels created by the agent also terminate with the agent. The variable types are known as port types and they are defined as follows:

$$T = [s_1(T_1), s_2(T_2), s_3(T_3), \ldots, s_N(T_N)].$$

Let us now examine in detail how communication takes place in Joyce. We will assume that we have two agents `p` and `q` that accesses the same channel through different variables `a` and `c` simultaneously and that port variables must be of same type. An output command like this: `b!s_i(e_i)` means that `b` is the port variable which outputs the message `e_i` of corresponding message type `T_i` and `s_i` is the name of one of the symbol of the classes of `T`.

For input command we have `c?s_i(v_i)` that denotes a port variable `c`, `s_i` denotes one of the symbol classes of `T` and `v_i` denotes the message of type `T_i`. This communication takes place when both `p` and `q` match that is when `p` is ready to output the symbol on a channel and `q` is ready to take the symbol from the same channel.

3.4.3 Mozart/Oz

Mozart was developed by Gert Smolka in 1990s at the Saarland University and later on developed by Seif Haridi and Peter van Roy [18]. It is a multi-paradigm language with features like concurrency, functional programming, object oriented, logical, imperative and constraint programming. Mozart is highly influenced by languages like Erlang, CSP, Prolog and LISP. The major strengths of this language are constraint logic programming and distributed programming which makes the design of network-transparent distributed model possible. In Mozart you can do several activities all together with the help of its concurrency feature. Concurrency in Mozart uses threads, which are the executing programs in the processor.

```
thread T in
   T={Funxtn 10}
```

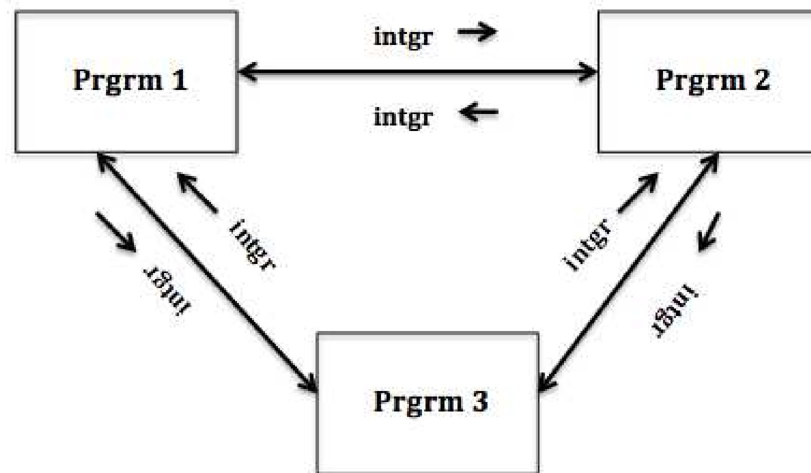

Figure 3.2: Message passing communication using three ports

```
    {Browse T}
end
    {Browse 15*15}
```

In the above part of code we have created a thread and later calling a function `Funxtn` inside the thread `T`. After that call, we used `browse` to print out the result. In Mozart, concurrency is also sometimes achieved with message passing. The model used in Mozart for message passing is the message-passing concurrent model that uses ports for the purpose of channel communication. Let us take an example to explain the message passing communication-using ports. Suppose we have three processes that pass integers to each other. When Process 1 receives the integer it randomly passes it to either `Process 2` or `Process 3`. This case applies to the other two processes as well. This is implemented by the use of port objects by sending the integer to the objects of the processes. The figure 3.2 is known as the component diagram. The instances of the process component are created to execute it and will use them to call the objects.

```
fun {Prgrm tple}
   {NewPortobject2
      proc {$$ Message}
      case Message of intgr then
        Randm={OS.rndm} mod {Width tple} + 1
        in
        {Send tple.Randm intgr }
       end
     end}
   end
```

Here `tple` is a tuple that contains other processes. We have the instances of the processes as `P1`, `P2`, and `P3`. To start the execution we will have *Send* command.

```
{Send P1 intgr}:
P1 = { Prgrm tple(P2 P3) }
P2 = { Prgrm tple(P1 P3) }
P3 = { Prgrm tple(P1 P2) }
```

In Mozart there is another kind of concurrency model with ports but in this model thread and ports both have access to the shared objects. We have already discussed about the disadvantages of shared resources and the use of locks to manage situations that arise due to sharing. Locks make sure that if a process accesses any critical region it should be the only one with exclusive permissions [32]. In Mozart we can implement the locks by using the entities in the language such as cells, variables and threads. However we also have a direct method called thread-reentrant lock. Locks have certain operations like:

1. To return a new lock we have `{NewLock N}`.

2. To check if we have a reference to a lock `{IsLock L}`.

3. To guard a statement in a program we can have `lock L` then there can be two cases, one if there is no thread executing the statement block any other thread can enter it for the execution. If there is already a thread all the other threads that try to enter will be suspended. Statement block is defined as:

```
<statement> end guards <statement>
```

3.4.4 Limbo

Limbo programming was designed by Sean Dorward, Rob Pike and Phil Winterbottom. It is used to develop applications for small, distributed systems [28]. It has various features and some of which are garbage collection and type checking at compile time from ML, Channels from Occam, declaration from Pascal, alternating on channels from Communicating Sequential Processes and many others. In Limbo inter-process communication takes place through channels. Channels are used as a kind of data type in the language and are considered to have reference semantics. When channels are assigned a value they are actually being assigned a reference of the objects. Here is an example to show how channels are created in Limbo [17]:

```
chan of data-type
```

With the statement above we can create a channel of specific data type but if a channel is declared without any assignment, it is assigned null by default. To use a channel we have to assign it as shown:

```
ch: chan of string;
ch= chan of int;
```

Also we can have the notation as:

```
chan of(int,string)
```

Channels transmit integer as well string tuples. And once the channel's object has been declared it can be used to send information. Let us assume `c` is an object:

```
c<-=(467,"Hey There")
```

In Limbo channels are un-buffered. Sender exists until there is any receiver but using the `bufchan` function of a module of Limbo it is possible to create buffers in the channel. The function bufchan takes a string `chan`, a size as an argument and creates an asynchronous task to accept the input from the channel with the argument. It can save number of strings equal to the argument string `chan`, meanwhile it tries to send the strings to the users.

3.4.5 Go

Go programming language was developed by Robert Griesemer, Ken Thompson, and Rob Pike in 2007 at Google [27]. It is a general-purpose concurrent language with multiple paradigms such as imperative and structured. Go programming language was developed for systems programming and has an explicit support for concurrent programming. Like other languages it has a wide variety of types such as Boolean types, numeric types, and string, array, slice, struct, interface and channel types. An interface type is basically used to implement interfaces and can store value of any type. All the variables that are uninitialized by default are assigned nil value. Example for an interface is:

```
interface {
Recv(w Buffer) bool
Send(w Buffer) bool
Close()
}
```

In an interface all the methods should have a unique name.

The type that is used for implementing concurrency model and synchronization of two concurrent functions is Channel Type. Like Interface type the value of all uninitialized channel is always nil.

```
ChannelType = ("chn" ["<-"] | "<-" "chn") ElementType.
```

In `ChannelType`, `<-` is used to specify the direction of the channel to denote if it is sending or receiving. If no direction is specified, it is assumed to be bi-directional. To create a new channel with some initialized value there is a built-in function and the value used to initialize is the buffer size of the channel [5].

```
make(chn int,178)
```

The initialized value is the one that decides if the communication will be synchronous or asynchronous. If the value is greater than zero, communication is said to be asynchronous and if it is not specified or zero it is said to be synchronous. It also has the select statements to help it decide which communication takes place.

```
SelectStmt = "slct" "{" { CommClause} "}"
CommClause = CommCase ":" StatementList .
CommCase = "case" ( SendStmt | RecvStmt ) | "default" .
RecvStmt = [ ExpressionList "=" | IdentifierList ":=" ] RecvExpr .
RecvExpr = Expression . (it must be the receiving function)
```

In Go language there is another entity called `Goroutine`; these are lightweight threads of the Go language. These are used to execute parallel tasks in the language. As mentioned in Google docs, Goroutines have a very useful application while doing parallel programming. "Goroutines are multiplexed onto multiple OS threads, if one should block, such as while waiting for I/O, others continue to run". Another important aspect of the Goroutine is that when used with channels it provides the paradigm of the Go programming language as is defined in Effective Go as "Do not communicate by sharing memory; instead, share memory by communicating." The main idea for communication is that if there is an un-buffered channel when receiving or sending, Goroutines will synchronize to avoid complexity otherwise they are free to communicate either concurrently or in parallel.

Chapter 4

Communication in Erasmus

In this chapter we will discuss about resources that have been used during implementation and detailed description of implementation of each module for the whole communication.

4.1 Design Issues

To implement the run time communication system for Erasmus, we had to consider some constraints that are also our requirements for the implementation. Erasmus has no shared variables and it only uses synchronous communication. Thus synchronous message passing was chosen as the main method of communication. In Erasmus processes belong to cells and the size of a cell can be as small as a few kilobytes or as large as the whole distributed system. A cell can have any number of processes in it fitting its size. The cell manages the communication of the processes that it owns as well as communication with other cells. A cell cannot invoke methods in other cells but can communicate with them using synchronous message passing. In our run time system a cell is implemented as a manager that invokes all the processes defined inside it. Communication among processes inside a cell takes place through channels and processes have ports that are connected to these channels. As Erasmus

programs may also be distributed, i.e., run on a network of processors. We have implemented broker for communication over the network. Each cell has one broker, which can handle two channels at the same time. A process knows nothing about what it is communicating with, whether the message stays in memory or has to go over a network. As a manager is the cell, it decides whether a process should use channel or broker for the communication. To do that selection manager has to know second process' location with respect to first process. At any instant there can be number of processes running at the same time, asking for resources, so for that instead of finding the location of a process we tried to find if a process is present on the same processor or not. We tried different methods such as signals, unnamed pipes and named pipes but except for named pipes all the methods either needed to communicate with the process or had an ambiguous nature. We have implemented a named pipe with read and write permissions on every process and when a process is invoked it opens a named pipe on one end. Manager looks for an open named pipe on the other end and if it finds an open pipe with same name then it reckons that the process is on the same processor and it choses channel for the process otherwise the communication request is sent to the broker.

Another requirement was that a process during communication with either channel or broker should have a feel of communication with another process and not with any middleware. So both channel and broker should have same interface similar to that of a process. Also we wanted communication among process to have the facility of selection among incoming connections i.e., with `select()`. Communication over the network requires a network protocol and TCP/IP is the obvious choice because it is the Internet standard and softwares that implement are widely available. TCP/IP also has a `select()` in it. For channel we tried many inter process communication protocol on windows but only named pipe with overlapped input/output was the closest one. Although it worked similarly like `select()` but for communication a process needed

different set of attributes than used for a `select()`. So we researched about inter process communication on other platform like Unix and after some research found out that communication with named pipe on Unix works similar to TCP/IP with `select()`. On Unix too we chose the protocol for communication over network as TCP/IP with `select()` for the reason mentioned above and named pipe was chosen for inter process communication. So the protocol for interface of a channel is named pipe with `select()` and for broker's interface the protocol is implemented using TCP/IP with `select()`. We will explain the implementation details and the problems encountered while implementation in further sections.

4.2 Existing Resources

We used Unix as the platform for development and for programming we have used the `C++` programming language. The programs are written in `Xcode` for `MAC OS X`. We have used the following libraries:

1. `iostream`: It is an input/output stream library used for performing actions on strings of characters.

2. `cstring`: It is needed for `memset`. In our program it has been used for copying the memory block using `memcopy`.

3. `sys/socket.h`: It is a header for the internet protocol family. Like other headers it also defines structures like `socklen_t`, `sa_family_t`, `sockaddr`, `msghdr`, `cmsghdr`. In our program we have used `socklen_t` and `sockaddr` for `sa_family_t`, `sa_family` and `char sa_data[]` for socket address

4. `netdb.h`: It is needed for the definitions for database operations of the network. It also defines the hostnet structures, `netnet`, `protent` and `servent` structures.

In our program we used the following hostnet structures: `int_addrtype` (address type); `int h_length` (the length of the address in bytes); `char **h_ addr_lists` (a pointer to an array of pointers to network addresses, in network byte order, for the host, terminated by a null pointer).

5. `stdio.h`: It is needed for standard input output . In our program we used it for stderr, which is a standard error output stream for printing out the error.

6. `vector`: It is used to define vector container class. We used it to contain the port numbers that are randomly assigned to the channels for the communication.

7. `sys/time.h`: This is the header used for defining the `timeval` and `itimerval` structures that has the following members: `time_t`, `tv_sec`, `seconds`, `suseconds_t` , `tv _usec`, microseconds. In our program we used it to create the structures for the time of `select() struct timeval`.

8. `sys/types.h`: This is used for the data types in the program. It includes the definition of various types and some of them are `clock_t`, `clockid_t`, `gid_t`, `id_t`, `size_t`, `useconds_t`, `time_t`, `pid_t`. We used this in our program for the process id using the `pid_t`.

9. `unistd.h`: This is the header that is used in the program to define various kinds of constant symbols, types and for the declaration of functions.

10. `fcntl.h`: It is a header file in Unix used for many functionalities but in our program we have used it to set the flags of the descriptors specially in `select()` for pipes and TCP/IP.

11. `sys/stat.h`: This is the header that is used to define the data structure of the data values returned from the functions like `fstat()`, `lstat()`, `stat()`.

4.3 Experiment with different resources

The first experiment was conducted by varying the platform. Initially some parts of the program were written on Windows platform in C++. We wanted communication on both local and remote machines to be universal and to use select for both local and remote connections. TCP/IP works similar for Windows as well as Unix. For both platforms, it has `select` with same type of arguments so we were able to achieve same level of transparency on both the platforms. But our local communication was not implemented the same way as remote. For the local connection we used named pipes, but on Windows this was not a suitable implementation for two reasons. First, it has the ability to communicate on network and hence it was very difficult to differentiate between local and remote communication. Second, there is no `select()` on named pipe in Windows. In Windows it uses overlapped I/O as its functionality for `select()`. Although it works similarly, we needed local and remote communications to be replicas of each other. We had to find something similar to remote functioning. In Unix we tried implementing named pipe and it only supported local communication. It had a `select()` statement with the arguments similar to remote communication. In named pipe `select()`, pipe names are used as selector instead of ports. Hence, we decided to change the platform of development to Unix since the features of named pipe and TCP/IP in UNIX met all our requirements with the added ability to communicate using `select()`.

Secondly, we performed experiments with different types of inter-process communication methods to differentiate between local and remote methods. First experiment was with signals. We thought that we could use signals to notify the processes when any other process is ready to communicate instead of sending messages. So signals were the best option to start a process, or to start communication and to abort it. But sending signals required us to know the location of the process beforehand and

also required the receiving process to reply which in turn lead to initiation of communication which we wanted to avoid. Next we tried using unnamed pipes on all the processes. As soon as any process starts it detects the presence of other unnamed pipes on the processor. But because of their unnamed nature they are able to detect pipes, which are not even related to our required processes. Hence we came to a conclusion to use named pipes. We will explain how it is implemented later on in Section 4.5.

4.4 Unix as most appropriate and adaptable

Motive of our research is to implement communication in a way that either process has the least role in communication, in other words insulated from the communication. We want our communication to be universal with the same protocol in effect for local and remote communication. Processing of processes shouldn't be different for different set of programs and it applies for local as well as remote. For all the requirements we needed a component like a channel in between the processes. We needed similar functionalities for inter process communication and also for remote communication. In Unix platform we have huge number of libraries that are deemed stable for use for a long time providing us enough choices and resources for our research development. We have used named pipe for local and sockets for remote communication in Unix that have same functionality as required, which is not present in the windows platform. Another reason to use Unix is the absence of licensing requirements.

4.5 Detailed description of implementation

In our program we have created a number of classes such as manager class for triggering the communication, local channel class to handle the connection for local services and remote channel class for remote connections. Process class to add the ports and

pipes dynamically whenever required by the process and to send the control either to local or remote depending on the current process requirement. The program starts by calling `main()`, which calls the manager to select the channel among local and remote. To do the selection we decided to check it on the local site instead of sending requests unnecessarily over remote processors. For that we have used a named pipe. All processes have a named pipe on them, which opens one end of pipe as soon as a process is initiated. Manager after process initiation checks for the presence of same name named pipes and if found any then local connection is selected and otherwise it can connect to the remote connection.

If we have a local connection open, the next thing that we had to achieve was to detect incoming connections and select one among them. Processes don't know about the other processes on the processor so `select()` was used to check if there were any incoming connections from any process to send or receive. If it had a receiving connection the control would be passed to receive connection class else to the sending class. But if there was no incoming connection we can pass the control again to the `main()`. For remote communication everything proceeds in the same way as it does for local communication. First `select()` will check if it has any incoming connection for either receive or send and if not any, the control is passed back to `main()`. There can be a critical case for example if a process `Process1` wants to communicate with a process `Process2` but `Process2` is already busy with another process `Process3`. Then in such a case we have make use of a process control block for every process. Each time a process will send a request its state will be saved and the state has associated fields of processing, ready or done. Other than that, the process control block will store information about which process it sent the request to and how many times the request had been sent. The process control block can save up to five process states. Even after communication is over, a process will retain some communication information in its process control list. If a process denies serving an incoming request

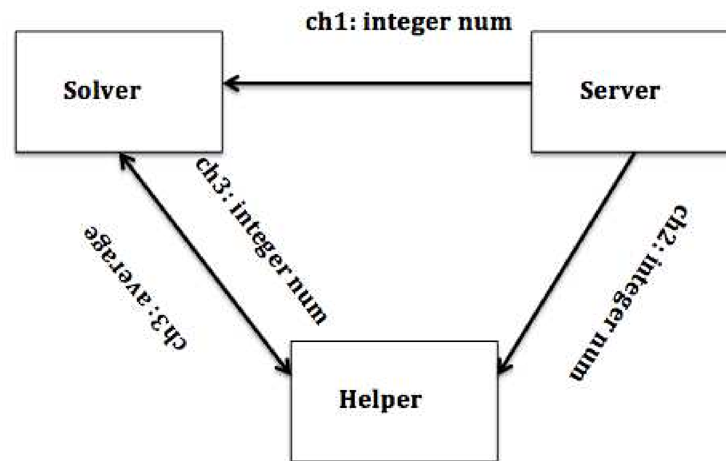

Figure 4.1: Implementation using three ports

from another process then the requesting process will try to resend the request for up to three times each request being sent after a fixed interval of time. There is a counter for each time a request is sent. As soon as the counter reaches a value of 3 the requesting process will drop the request and control is transferred back to `main()`. The process will then start over its communication with another process.

4.5.1 Basic message transfer

Now let us discuss the implementation in detail by taking some examples. There are three processes named solver, helper and server. Each of them has their own specific functions. They are implemented as part of a cell called manager. A channel connects server, solver and helper to each other and each connection has its own dedicated channel as shown in figure. We have channel `ch1` between solver and server; Channel `ch2` between helper and server, Channel `ch3` for helper and solver. A simple experiment of sending random numbers and calculating the average of those numbers on a processor inside a cell was performed. Server served the purpose of generating

the random number and sending them to either solver using `ch1` or to helper using `ch2` depending on whichever requests for it.

```
Num = protocol { ^w:Word }
```

Num protocol is for specifying which channel will be used for sending numbers. `Random(100)` function will be called to generate random numbers between 0,1,2....99. Small pseudo code to show the processing:

```
Server = process ch1,ch2: +Num
{
  loop select
  {
   || ch1.w := random(100)
   || ch2.w := random(100)
  }
}
```

Helper can send a request to either of the processes: server or solver. But its function differs depending upon which channel the request was sent. If a request is sent on channel `ch2` it will receive the number from server and will increment its numval. If sent on channel `ch3` it will get the number from solver and in reply it will send the average of the numbers it has received from the solver. Another protocol to calculate the average of numbers is also needed. The main objective of sending the average computation is to show that a channel can transmit different kinds of values.

```
Average = protocol {request; ^r:Real }
Helper = process ch2: -Num; ch3: +Average
{
   average: Real := 0;
   numvals : Word := 0;
```

```
            loop select
            {
              || average + = ch2.w;
              numvals +=1
              || ch3.request;
              ch3.r := average / numvals;
              average := 0;
              numvals := 0
            }
         }
```

Solver is the one that acts as client. It sends a request to the helper on channel ch3 and to the server on channel ch1. It retrieves the value from server and average from helper. Then, with the helper on channel ch3 solver performs two actions of read and write.

```
Solver = process ch1: -Num; ch3: -Average
{
  for (n := 0; n<100; n +=1)
   {
     w: Word := ch1.w;
     ch3.request;
     r: Real := ch3.r
   }
 }
```

After discussing about all the three processes, we go on to discuss the cell manager. Manager is the process that initiates the communication by invoking the methods for other processes. When a cell process starts execution it creates an instance of the

manager that will create the processes and start communication. The pseudo code is:

```
Manager = cell
{
  ch1,ch2: Num;
  ch3: Average;
  Server(ch1,ch2);
  Solver(ch1,ch3);
  Helper(ch2,ch3);
}
```

4.5.2 Selection among local and remote

The main part of our implementation was to choose between local and remote communication. After manager will start the communication it will call the function `check_local()` to start communication either local (channel) or remote (broker). Function `check_local()` returns a variable integer value used by manager to select among channel and broker. The code is:

```
check_local()
{
  int err;
  status = mkfifo("recv",0666);
  fd1 = open("recv",O_WRONLY);
  fd2 = open("sendd", O_RDONLY);
  if(fd1<0 && fd2<0)
     {
       err = 1;// a const for remote
```

```
    }
 else if(fd1>0 || fd2>0)
    {
      err = 2; // a const for local
    }
else
   {
     err = 3; // a const for progrm failure error
   }
return err;
}
```

So a named pipe is opened on the processes as soon as it is created by the manager and `check_local()` function will check if any pipe is open for communication just by checking the presence of named pipe on the system. After selection has been made communication will continue with `select()` statement on either local or remote.

4.5.3 Channels and Broker

When two processes communicate with each other locally they go through a channel while for remote communication they go through a broker. Named pipe has been used for channel implementation while TCP/IP for broker implementation. The broker is implemented in a way that it can handle two channels. It is important to point out that its implementation is same as that of the channel. Whenever manager starts the processes it decides whether to use a channel or broker but there will be no effect of this selection on the processes as all the processes follow the same protocol for communication. The common protocol for channel and broker in pseudo code is:

```
run(){
```

```
ProccessBlockList list[5];
int i =0;
char *num=0;
addChannel(new RemoteChannel());
for(int a=0;a<=3;a++){
if(a<3){
      if(ports[i]->readyToRecieve_remote()){
      remote_list[k].updateProcessBlock("waiting");
      ports[i]->recieve_remote(num);
      break;
      }
      if(ports[i]->readyToSend_remote()){
      remote_list[k].updateProcessBlock("ready");
      ports[i]->sendto_remote(num);
      break;
       }
    sleep(300);}
    else{
      manager();
      }
   }
}
```

In the above pseudo code we have shown that regardless of choosing a channel or broker, a list is created for every process that can store up to five process states with which it will communicate. After selection among channel or broker the request is sent to `select()` to check any incoming connection for read/write. Then the state of the process in the list created will be updated to *waiting*. But if `select()` does not

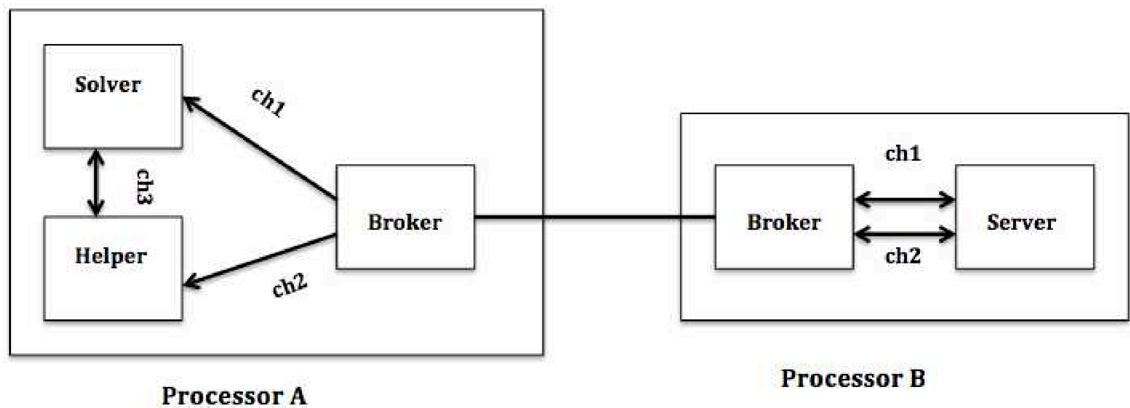

Figure 4.2: Remote Communication Implementation

receive any incoming requests, then instead of waiting forever it will wait for some time and will retry for three more times. If in the mean time it receives a connection it will proceed with the communication otherwise control will be passed to `manager()`.

4.6 Problems encountered and their solutions

In this section, we describe some of the problems that were encountered in the implementation of communication, and explain the solutions that were used.

4.6.1 Local versus Remote

Our goal was to have network as well as local communication with the ability that processes are able to select their communication medium. For remote communication, we chose TCP/IP with ports, which are assigned whenever a communication connection is ordered. It has the ability to check any incoming communications and to make a selection among them using `select()`. For local communication, many different approaches were tested but using named pipes was the closest match for our requirements. However, on windows if a named pipe finds any process on other

processor it can communicate remotely. Other than that there was the problem with named pipes that they had no `select()`, only overlapped input/output. The next best option to using named pipe was unnamed pipes. But it too proved to be futile since because unnamed pipes can communicate with any random anonymous pipe in the system and hence are preferable for child parent process communications.

Hence we decided to test the different methods of communication on different platform like Unix. In Unix the implementation of network communication was same as that of windows with same features, but our main reason to change platform was local communication. We tried using named pipe on Unix and it was just limited to local communication and had the `select()` which helped us to match the implementation of both local and remote communications. So we have implemented named pipe and TCP/IP with `select()` for local and network communication simultaneously on the Unix platform.

4.6.2 Distinguishing local and remote

We wanted to implement a distributed approach and hence we did not want processes to know about the location of other processes. When a process wants to start communication it should not be worried if a process is local or remote and it should use same parameters for both kind of communication. The only thing a process should know is what kind of data it has to send. We needed another process to help the other processes on the system to direct them for either local or remote communication. The idea behind our approach is that we can have thousands of processes in a system asking for resources at some point of time and instead of wasting resources remotely we can try to fulfill their requirements locally. We needed something cost efficient in our process cycle. We chose local process and created named pipes on them. If any process has access to that pipe then there are chances they can communicate directly. Whenever a process was online and their named pipe was accessible they were

considered local. Process calls the channel to start the communication and checks if a process is available or is busy. Even if at the same time two pipes tried to check local connection it will not affect the processing as named pipe can communicate with more than one pipe at the same time.

4.6.3 Symmetrical selection

One of our goals was to have linear communication between processes no matter if they have `select()` or not. For that we had channels, brokers and manager cells but the problem occurred when both the communicating sides had `select()` on them. We knew that if both communicating processes had select it would not be possible to communicate, as they both would have waited forever. This problem is very critical problem for any language and even for Erasmus. To tackle this the solution we have assigned a random timer with `wait(time)` for any process with `select()`. So, if `select()` does not receive any incoming connection for that amount of time it will start sending data directly to the process. In that case it will send the data and wait if somebody is listening so that it can communicate. Otherwise it will again start with the loop for local and remote because there might be the case that the processes it wants to communicate are busy.

4.6.4 Concurrent communication

Another problem that we faced was if a process P1 is already in communication and there is a request from another process P2 and hence P1 would not reply to that process since it was already busy. To tackle this, we created a buffer of size 5 to keep record of information about process to which a request is sent. It stores information like its process id so that it can retry sending requests after some time and complete its processing. Process P2 will resend request to process P1 for a total of three times. If in vain, it will stop trying and will continue sending request to another process.

This way no process will be starved waiting forever and no process's resources will be wasted and no deadlock will occur because maximum of three chances are given to start the communication.

4.6.5 Idle processes

Another difficult problem was if a process has not got any connection demands and has been idle for a while. It is an unpleasant scenario and very difficult to solve if a process becomes idle as no process can communicate with that process. For that reason no process gets idle forever by itself.

Chapter 5

Results

5.1 Performance measurements

In previous chapter we discussed all about how we implemented our solution and now we will take a look at its performance in comparison with the well-known language Java. Usually C++ applications give better performance than Java due to the fact that Java has `JVM`. We did performance tests on MAC OS X for C++ using Xcode and using Eclipse Juno for Java on the same platform. For the first performance measurement we took data of fixed number of bytes (839 bytes) while the number of repetitions was changed and the time taken to send the data was then measured in seconds.

From the figure of performance measurement 1, we can see that C++ gave a better performance than Java when run for 5, 10, 15 times but on increasing the loop counter both seemed to achieve similar performance with Java being slightly faster for longer loops.

In second figure we have measured the performance on how performance will vary with the change of number of bytes so we ran our process for 15 times and increasing the number of bytes in each message gradually. We found that C++ in small byte data

M (Number of Run)	C++	Java
	T = time	T = time
5	0.017	0.037
10	0.025	0.088
15	0.042	0.107
25	0.348	0.305
35	0.371	0.317

Number of Bytes used for transferring is same in all cases i.e. 839 bytes

Figure 5.1: Performance measurement with varying number of sending process repetition times

N (Number of Bytes)	C++	Java
	T = time	T = time
210	0.013	0.018
425	0.025	0.034
856	0.049	0.084
1712	0.276	0.186
3424	0.289	0.203

Number of Run used for transferring is same in all cases i.e. 15 times

Figure 5.2: Performance measurement with varying number of bytes

performed faster than Java while later on they were almost same. At the conclusion it is not fair to say which language was faster as both of them performed similarly. In Java the performance varies with the use of different methods for writing data like if we use its usual method for writing its slow but when we used `Inputstream.read` it performed faster.

5.2 Comparison with other languages

In this section we will compare the implementation of our communication method with other languages that we have mentioned throughout the previous chapters. We have concurrent programming implemented in our module and hence we will compare on the basis of concurrent communication paradigms via shared variables and via message passing.

5.2.1 Communication via shared variables

Communication via shared memory has been around from many years. It is considered among the fastest method of communication but it comes with its disadvantages. Due to these disadvantages many new languages as well as old ones like CSP and its derivatives don't rely on shared memory communication. In Erasmus we use message passing for communication. Some high level languages like Java uses shared memory communication for inter threaded communication. Especially in sequential languages with concurrency, it can be a bottleneck as shared memory can result in duplication of data or data inconsistency and race condition as explained in an example above in the section of shared memory. There might be a case that a process wants to use a variable but it has already been changed by some other process. In such cases programmers have to be very careful about the implementation in order to avoid non-determinism. Some like Mozart/Oz use operations and types that allow the protection of data while

one process is using the memory space.

However in Erasmus we eliminated these problems with the help of synchronous message passing. Only processes within the same cells can communicate with each other. In Erasmus every cell has its own control thread and no two processes will access memory at the same time avoiding the disadvantages of shared memory [11].

5.2.2 Communication via message passing

We have briefly discussed message passing and its types in Section 2.2 and 2.3 in which we usually send a message when the receiver is ready and there is no wait in between (synchronous), and there can be information or data sent out even if nobody is receiving yet (asynchronous). In the second type that is asynchronous we need something to hold the data similar to a buffer. In functional languages objects are usually created only when they are needed and often they can be very large likely as computer's available memory. So the unbounded buffer will be an obvious fit for such kind of languages. In Erasmus we use synchronous message passing in contrast to languages like Erlang and Mozart/Oz that communicate using asynchronous message passing.Channel and message passing in Erasmus are highly influenced by Joyce programming language.

But in Joyce there is no facility to provide the sequencing of the messages when they are sent together, although in Erasmus we do have that.

Some languages like CSP and its derivatives use channels for message passing communication and as discussed above in Section 2.3 they have unidirectional channels. During communication receiver always passes a message of acknowledgement to the sender, however in Erasmus we have a request reply mode for communication [11].

Chapter 6

Conclusions and Future Work

In our Final chapter we will conclude our research and any future prospects will be discussed.

6.1 Conclusions

In the previous sections we have discussed about the various aspects of concurrent communication, a more effective way for concurrent communication and comparison amongst the many different languages. If we start with chapter 1 we gave an introduction to concurrent programming, since in order to implement it, we should be able to understand it completely. We gave some reasons that although concurrency can be achieved by shared memory as well as message passing; it is good idea to go with message passing. We wanted to implement concurrency in a distributed way so we have a provided a detailed problem statement of the message passing syntax and how it is independent of its local or over different processors.

In chapter 2 we gave a brief background on sequential languages as well as concurrent languages, which gave us an idea about their current implementation together with their advantages and disadvantages. For achieving concurrency it is very important to know about its different paradigms and how the communication should

take place so we discussed the features of these paradigms. Then we went on to discuss about how different languages manipulate their communication syntax when conditions vary.

In chapter 3 we discussed in detail about Erasmus. Since it is a new language, we discussed about its goals and how different it is from other languages that exist today. The most crucial point is that our thesis is about designing communication in Erasmus so knowing more about Erasmus gave us ideas for making its communication more efficient. Another thing we discussed about was need of platform independency in programming languages. In Erasmus we have compilation and deployment as separate modules so there is no need of porting the code for each new platform giving it platform independence. Later on we discussed about CSP as it has a high influence on Erasmus and languages like Occam, Go, Joyce, Mozart and limbo as they give a more clear idea about Erasmuss implementation.

Chapter 4 is the most important part of our thesis; this is where we discussed our implementation of communication in Erasmus and the resources that we required in terms of libraries, platforms and programming languages. We have discussed about all the options we tried for implementing communication and also discussed the ones we chose. We gave a broad view on implementation with state diagrams and implementation pseudo codes. While doing all these we encountered many different kinds of challenges and we have tried to discuss the possible solutions and how we tackled the problems by finding the appropriate solution for those problems.

Lastly in chapter 5 we compared Erasmus with all the languages we mentioned in chapter 2 and 3. We compared those languages by talking about the way they communicate, e.g. by message passing and shared variables etc. We also discussed how Erasmus is different from them at the same time inheriting some of their features. Some performance measurements have been done to check how efficiently our solution really works.

6.2 Future Work

What we did for this thesis is a small section of the Erasmus development and there are myriads of things left to do. So in this section we will highlight the tasks, which will serve as future work:

1. Right now our program for communication by sending the integer number locally and over the network but in future we would like to send all kind of data types.

2. We have designed the communication for local and remote but we still have to integrate this code in the Erasmus compiler.

3. After integrating the code in Erasmus we would like to take the performance measurements on the installed code.

4. We would like to do fairness tests with one server and more than one client, to check if all the clients are served equally by the server.

5. We have to evaluate the practical value of the techniques we have developed.

6. Generalization of our implementation from 1:1 (one client, one server) to m:n (m clients, n servers). Right now we just have a single point of communication but later on there should be a provision for multiple processes in a processor.

Bibliography

[1] Peter Grogono, Nurudeen Lameed, and Brian Shearing. *Modularity+ Concurrency= Manageability*. Technical Report TR E04, Department of Computer Science and Software Engineering, Concordia University, 2007.

[2] Ben-Ari, Mordechai. *Principles of concurrent and distributed programming.* Pearson Education, 2006.

[3] Laura Effinger-Dean. *Introducing Shared-Memory Concurrency, Race Conditions and Atomic Blocks.* November 19, 2007. `http://wasp.cs.washington.edu/atomeclipse/handouts.pdf`. Retrieved 2014-01-16.

[4] Hyde, Daniel C. *Introduction to the programming language Occam*, pages 3-11. Department of Computer Science Bucknell University, Lewisburg (1995).

[5] *Channel* types, *Select* statements - The Go Programming Language Specification. `http://golang.org/ref/spec`. Retrieved 2014-01-18.

[6] Armstrong, Joe, Robert Virding, Claes Wikstrm, and Mike Williams. *Concurrent Programming* in ERLANG, pages 67-81. Prentice Hall, second edition, June 2004.

[7] Finkel, Raphael A. *Advanced Programming Language Design*, Chapter 7. (1996).

[8] Gregory R. Andrews and Fred B. Schneider. Concepts and Notations for Concurrent Programming. *Computing Surveys*, **15**(1):3-43, March 1983.

[9] Jafroodi, Nima, and Peter Grogono. "Implementing generalized alternative construct for erasmus language." *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. ACM, 2013.

[10] Budd, Timothy A. *Introduction to Object Oriented Programming*, Chapter 5 Messages, Instances and Initialization. Addison-Wesley, 3rd Ed . Oct. 2001.

[11] Lameed, Nurudeen. *Implementing concurrency in a process-based language.* Master's Thesis Concordia University, 2008.

[12] May, David. Occam, *SIGPLAN Notices*, Vol. 18, No. 4, April, 1983, pp. 69-79.

[13] Wexler, John. *Concurrent programming in OCCAM 2.* Prentice-Hall, Inc., 1989.

[14] Barnes, Fred, and Peter Welch (2006-01-14). *Occam-pi*: blending the best of CSP and the pi-calculus. Online available `http://www.cs.kent.ac.uk/projects/ofa/kroc/`. Retrieved 2014-01-14.

[15] Per Brinch Hansen. *Joyce-A Programming Language For Distributed Systems. Software Practices and Experience*, **17**(1):1-24, January 1987.

[16] Orfali, Robert. *The Essential Client/Server Survival Guide*. New York: Wiley Computer Publishing. pp. 375-397, 1996.

[17] Ritchie, Dennis M. The *Limbo Programming Language*. Online available `http://doc.cat-v.org/inferno/4th_edition/limbo_language/limbo`. Revised 2005 by Vita Nuova.

[18] Van Roy, Peter, and Seif Haridi. *Concepts, techniques, and models of computer programming*, pages 1-41. The MIT Press, 2004.

[19] Hoare, Charles Antony Richard. Communicating sequential processes. *Communications of the* ACM 21.8 (1978): 666-677.

[20] Cesarini, Francesco, and Simon Thompson. *Erlang programming*, pages 45-102. O'Reilly Media, Inc., June, 2009.

[21] Herlihy, Maurice, and J. Eliot B. Moss. *Transactional memory: Architectural support for lock-free data structures.* Vol. 21. No. 2. ACM, 1993.

[22] Stevens, W. Richard. *UNIX Network Programming: Interprocess Communications*, pages 303-310. Volume 2, Second Edition. Prentice Hall, 1999.

[23] Armstrong, Joe. *Making reliable distributed systems in the presence of software errors*, pages 19-26. Diss. KTH, 2003.

[24] DeLillo, Nicholas J. A *First Course in Computer Science With Ada.* McGraw-Hill Higher Education, November 1992.

[25] Magee, Jeff, and Jeff Kramer. *Concurrency: State Models and Java Programs*, pages 209-230. Wiley, 2006.

[26] Farley, Jim. *Java distributed computing*, pages 1-74 . O'Reilly, 1 January 1998.

[27] *Language Design FAQ*. `http://golang.org/doc/faq`. 16 January 2010. Retrieved 27 January 2014.

[28] Phillip Stanley-Marbell. *Inferno Programming with Limbo*, pages 3-6. Chichester: John Wiley and Sons, 2003.

[29] Dijkstra, Edsger W. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the* ACM 18.8 (1975): 453-457.

[30] Burns, Alan, and Wellings, Andy. *Concurrency in Ada*, pages 31-54. Cambridge university Press, second edition, 1998.

[31] Armstrong, Joe, Virding, Robert, Wikstrom, Clases , and Williams, Mikes. *Concurrent Programming in ERLANG*, pages 67-71. Prentice Hall, second edition, June 2004.

[32] Peter Van Roy. *General Overview of Mozart/Oz.* Slides for a talk given at the Second International Mozart/Oz Conference (MOZ 2004).

[33] Grogono, Peter, and Brian Shearing. MEC Referance Manual. *Technical Report* TR E-06 Department of Computer Science and Software Engineering, Concordia University, January 2008.

[34] Hoare, Charles Antony Richard. *Communication Sequential Process.* Prentice Hall International, third edition, June 2004.

[35] Lee, Edward A. The Problem with Threads. *IEEE Computer,* 39(5):33-42, May 2006.

[36] Gropp, William D., Ewing L. Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface. Vol. 1.* the MIT Press, 1999.

[37] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 7 Edition. Oracle, 2013-02-28. Online available at `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`. Retrieved 2014-02-09.

[38] Cartwright, Robert “Corky”. Notes on Object-Oriented Program Design. Copyright 1999-2010. Online available at `http://www.cs.rice.edu/~cork/book/node96.html`. Retrieved 2014-02-09.