Kandidatene sine navn: Carl Erik Hauge, Kenneth Pedersen og Kjetil Pedersen

Fag: Datateknikk

Tittelen på oppgåva (norsk): Rendezvous-biblioteket

Tittlene på oppgåva (engelsk): The Rendezvous Library

Oppgåvetekst:

This study will implement and evaluate a library that can be used for communication in distributed systems. The implementation will be based on the Rendezvous mechanism (which is used for communication in Ada programs), and will be implemented in C++.

A small distributed numerical application will be implemented using this library, and compared to versions of the same application that are using MPI, OOMPI and CORBA for communication. The average execution time of the four different versions will be measured using different numbers of processors in a cluster of workstations.

The study will be concluded with a suite of benchmarks measuring the performance and ease-of-use of the library compared to MPI, OOMPI and CORBA, in the context of distributed applications and from the point of view of the application developer.

Oppgåva gjeve: 21. januar 2002
Oppgåva leverast innan: 17. juni 2002
Oppgåva levert: 17. juni 2002
Utført ved: Institutt for datateknikk og informasjonsteknikk
Fagleg rettleiar: Roxana E. Diaconescu

Trondheim, 17. juni 2002

Reidar Conradi
Faglærar
Abstract

There exists several communication mechanisms for exchanging data between processes in a distributed environment. One of these, the rendezvous mechanism, is used by programming languages like ADA, Concurrent C and SR.

It has been our goal to implement an object oriented multithreaded communication library that we’ve called RL (the Rendezvous Library). The library uses rendezvous to communicate between the various tasks, and it is implemented in C++ on the GNU/Linux operating system using TCP/IP as the transport mechanism. By using threads we achieve overlapping between computation and communication, thus improving performance. It is our hypothesis that this library would perform well compared to MPI, CORBA and OOMPI, and that it would have usability better than CORBA.

In addition to the library itself, we implemented a compiler that transforms interface specifications written in RL IDL to C++ code used for marshaling and unmarshaling an interface’s operations. RL IDL is a subset of OMG IDL, the interface definition language used by CORBA. We also implemented a directory service that supports location independence of services through mapping of object references to human-readable names.

To validate our hypothesis we ran a test where RL is compared to CORBA, OOMPI and MPI with regards to execution time. Although the test is too small to provide conclusive evidence, it indicated that RL’s performance is significantly better than CORBA. The exact numbers vary greatly depending on the number of processors. The tests also indicate that RL has performance similar to MPI and OOMPI.

We have also compared the 3 approaches with respect to usability. We conclude that RL is easier to learn than CORBA, but it does not provide the wealth of features that CORBA supplies. It also offers a more natural style of programming in object-oriented languages than MPI or OOMPI, but it does not offer the same level of control over communication as MPI or OOMPI.
Contents

1. Introduction ................................................. 1
   1.1. Motivation ............................................... 1
   1.2. Structure of This Report .............................. 2

2. Distributed Systems Background ............................ 3
   2.1. Distributed Systems Concepts ........................ 4
   2.2. Process Management .................................. 6
   2.3. Communication Mechanisms .......................... 7
   2.4. Synchronization ...................................... 12
   2.5. The Rendezvous Concept ............................. 17
   2.6. Concurrent Programming Libraries .................. 31

3. Context .................................................. 52
   3.1. Motivation ............................................. 52
   3.2. Research Agenda ..................................... 55

4. Rendezvous Library Overview ............................... 61
   4.1. Overall Architecture ................................. 61
   4.2. Using the RL library ................................ 64
   4.3. RL Concepts And Semantics ......................... 66
   4.4. IDL Compiler .......................................... 69
   4.5. Directory Service .................................... 78
   4.6. Applications using RL ............................... 81
   4.7. Comparing RL to Other Approaches ................ 82

5. Implementation ............................................. 83
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.7</td>
<td>RL::MainThread Class Reference</td>
<td>168</td>
</tr>
<tr>
<td>C.8</td>
<td>RL::ObjectRef Class Reference</td>
<td>170</td>
</tr>
<tr>
<td>C.9</td>
<td>RL::ObjectServant Class Reference</td>
<td>172</td>
</tr>
<tr>
<td>C.10</td>
<td>RL::Runtime Class Reference</td>
<td>175</td>
</tr>
<tr>
<td>C.11</td>
<td>RL::Task Class Reference</td>
<td>177</td>
</tr>
<tr>
<td>C.12</td>
<td>RL::TCPLocation Class Reference</td>
<td>182</td>
</tr>
<tr>
<td>C.13</td>
<td>RL::TCPPool Class Reference</td>
<td>185</td>
</tr>
<tr>
<td>C.14</td>
<td>RL::TCPTransport Class Reference</td>
<td>188</td>
</tr>
<tr>
<td>C.15</td>
<td>RL::Thread Class Reference</td>
<td>193</td>
</tr>
</tbody>
</table>

D. IDL Syntax and Semantics

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1</td>
<td>Overview</td>
<td>196</td>
</tr>
<tr>
<td>D.2</td>
<td>Lexical Conventions</td>
<td>197</td>
</tr>
<tr>
<td>D.3</td>
<td>Preprocessing</td>
<td>201</td>
</tr>
<tr>
<td>D.4</td>
<td>RL IDL Grammar</td>
<td>201</td>
</tr>
<tr>
<td>D.5</td>
<td>RL IDL Specification</td>
<td>202</td>
</tr>
<tr>
<td>D.6</td>
<td>Module Declaration</td>
<td>202</td>
</tr>
<tr>
<td>D.7</td>
<td>Interface Declaration</td>
<td>202</td>
</tr>
<tr>
<td>D.8</td>
<td>Value Declaration</td>
<td>204</td>
</tr>
<tr>
<td>D.9</td>
<td>Constant declaration</td>
<td>205</td>
</tr>
<tr>
<td>D.10</td>
<td>Type Declaration</td>
<td>205</td>
</tr>
<tr>
<td>D.11</td>
<td>Exception Declaration</td>
<td>208</td>
</tr>
<tr>
<td>D.12</td>
<td>Operation Declaration</td>
<td>208</td>
</tr>
<tr>
<td>D.13</td>
<td>Attribute Declaration</td>
<td>210</td>
</tr>
<tr>
<td>D.14</td>
<td>CORBA Module</td>
<td>210</td>
</tr>
<tr>
<td>D.15</td>
<td>Names and Scoping</td>
<td>210</td>
</tr>
</tbody>
</table>

E. RL IDL C++ Language Mapping

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.1</td>
<td>C++ Compiler Requirements</td>
<td>212</td>
</tr>
<tr>
<td>E.2</td>
<td>The RL Namespace</td>
<td>213</td>
</tr>
<tr>
<td>E.3</td>
<td>Modules</td>
<td>213</td>
</tr>
<tr>
<td>E.4</td>
<td>Interfaces</td>
<td>213</td>
</tr>
</tbody>
</table>

212
vii
E.5. Valuatypes ................................................................. 213
E.6. Constants ................................................................. 213
E.7. Type Declaration ....................................................... 214
E.8. Exceptions ................................................................. 216
E.9. Operations ................................................................. 216
E.10. Attributes ................................................................. 216

F. RL IDL Compiler Invocation ............................................. 217

G. Installing RL ................................................................. 220
   G.1. Getting RL ........................................................... 220
   G.2. Compiling RL ........................................................ 221
   G.3. Installing RL ........................................................ 222

H. About the Source Code .................................................. 223
   H.1. Key Figures .......................................................... 223
   H.2. Directory Service Source Code ................................. 223

References ........................................................................... 230
Preface

This report is, together with a software library, the result of a master thesis for graduating students in computer sciences at the Norwegian University of Science and Technology in Trondheim during the spring 2002. The report has been written as a continuation of two system development projects from the autumn semester of 2001.

The projects has given us insight in the rigors of programming a multiprocess, multi-threaded library. Debugging this system has been difficult. Not only must the challenges of properly synchronizing several processes communicating via a network connection be considered, but in addition the many threads of a single process must be synchronized so they do not corrupt shared data structures. These two synchronization problems interact in ways that can be hard to understand.

We would like to thank Roxana Diaconescu, who once again has been very helpful during the project period. She has given us much valuable feedback and many suggestions on how to improve the quality of our work. We would also like to thank Zoran Constantines for giving us access to the cluster ClustIS\textsuperscript{1}.

More information about our work can be found on our website, http://www.faerun.dhs.org/diploma/.

Trondheim, 17th June 2002

Carl Erik Hauge Kenneth Pedersen Kjetil Pedersen

\textsuperscript{1}The Intelligent Systems Cluster at the Department of Computer Sciences, Intelligent Systems Division. See [27]
1. Introduction

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Structure of This Report</td>
<td>2</td>
</tr>
</tbody>
</table>

In this chapter we provide the motivation for our work, and describe how this report is structured.

1.1. Motivation

As our master thesis, we will design and implement an object oriented communication library for concurrent computing, comparable to the likes of CORBA and MPI. It is our goal to achieve good usability and efficiency by combining object-oriented concepts with low-level communication mechanisms. To improve performance we will use multiple threads, thus overlapping communication and computation. In order to validate our results with regards to performance, we will run a small test algorithm. The algorithm will be implemented using both RL, OOMPI, CORBA and MPI. We expect the RL library to perform better than CORBA, but not on par with MPI/OOMPI, with regards to execution time. With regards to usability we are interested in how easy it is for a user to write a program that utilizes the library and what kind of functionality the library provides.

Performance is important in computation intensive concurrent applications. A problem can usually be solved faster by running it concurrently on a distributed system. However, one is not guaranteed that 2 processors runs a program i.e. 75% faster than 1 processor. It might only run 50% faster. Therefore, how well the librarys performance scales with an increasing number of processors will also be an important consideration when evaluating the overall performance. When we have looked at usability we have been interested in the ease of which a user can write a program that utilizes the library, and what kind of functionality the library provides.
1.2. Structure of This Report

Chapter 1 Consists of the introduction and our motivation for writing this report.

Chapter 2 We present general background material on distributed systems. This includes basic concepts, communication mechanisms and synchronization mechanisms.

Chapter 3 We provide further motivation for our use of the rendezvous in the communication library. We present the research agenda, hypothesis and the metrics that we will use in our testing.

Chapter 4 We present an overview of the rendezvous library. We present the RL concepts, what kind of applications that can use RL, and the central parts of the library.

Chapter 5 We looks into the implementation of the rendezvous library and its 3 main components, the IDL compiler, the run-time library, and the directory service.

Chapter 6 Present two examples that we have used during the development and testing of RL. One is the producer/consumer interprocess communication problem, the other is an area computation problem.

Chapter 7 Contains our test results on performance and usability.

Chapter 8 We evaluate and validate the results. We also present suggested improvements.

Chapter 9 Concludes the report and suggests further work.
2. Distributed Systems Background

Contents

2.1. Distributed Systems Concepts ........................................ 4
   2.1.1. Distributed Applications ...................................... 4
   2.1.2. Processes ..................................................... 4
   2.1.3. Threads ...................................................... 5
2.2. Process Management .................................................... 6
   2.2.1. User Manages Processes Manually ............................. 6
   2.2.2. Language Manages Processes ................................ 6
   2.2.3. Library Software Manages Processes ......................... 7
   2.2.4. Operating System Manages Processes ......................... 7
2.3. Communication Mechanisms ............................................ 7
   2.3.1. Shared Memory ................................................ 8
   2.3.2. Pipes or Sockets ............................................. 8
   2.3.3. Message Passing ............................................. 9
   2.3.4. Remote Procedure Call .................................... 10
   2.3.5. Remote Method Call ........................................ 10
   2.3.6. Rendezvous ................................................ 11
2.4. Synchronization .......................................................... 12
   2.4.1. Synchronization Problems .................................... 12
   2.4.2. Synchronization Mechanisms ................................ 15
2.5. The Rendezvous Concept ............................................... 17
   2.5.1. Rendezvous in programming languages ....................... 17
2.6. Concurrent Programming Libraries .................................. 31
   2.6.1. TCP/IP Sockets .............................................. 31
   2.6.2. MPI/OMPI .................................................... 35
   2.6.3. CORBA ....................................................... 41
   2.6.4. Usability Comparison ....................................... 47
   2.6.5. Performance Comparison .................................... 50
A distributed system consists of two or more independent computers, working together as an integrated unit [2]. That the computers are independent means that architecturally, they are capable of operating as autonomous units. That they are integrated means that they cooperate to achieve a given goal. The computers have some means of communicating with each other (i.e. a computer network).

There are different kinds of distributed systems and it varies how tightly coupled the computers in the distributed systems are. Computers in a tightly-coupled distributed system are connected by a high-bandwidth, low-latency, reliable network, while nodes in a loosely-coupled distributed system are connected by a low-bandwidth, high-latency, unreliable network. Consequently, it is easier for computers in a tightly-coupled distributed system to share resources than it is for computers in a loosely-coupled distributed system. The degree of coupling in a given system affects how applications running in the distributed system must be designed.

The computers that form the distributed system is often called nodes.

2.1. Distributed Systems Concepts

Applications that run in distributed systems consists of a number of processes running in parallel. Each process can have one or more threads of execution. These concepts are discussed in this section.

2.1.1. Distributed Applications

An application is a computer program or a collection of programs that performs some given task (e.g. serving web pages, managing bank accounts, or simulating the weather).

A distributed application is simply an application that is designed to run in a distributed system. It can take advantage of the fact that multiple processors (CPU’s), memory banks, I/O devices (like harddisks) and network communication lines may be available, and takes into consideration that these resources may be located on different computers and that different parts of the application must communicate and synchronize using the network.

2.1.2. Processes

A process\(^1\) is just a running program. It has a program, input, output, and a state. The program is the series of instructions that the process will execute, while the state is the collection of data stored in the process’ region of memory. The input is data read either

\(^1\)Sometimes (like in [3]), a process is also referred to as a task, but we will use that term with a slightly different meaning.
from external units such as a hard disk, or from other processes. The output is data sent to external units or to other processes.

A process is also a unit of resource ownership. Each process has a virtual address space to hold the process image, and from time to time it may be allocated ownership of resources such as main memory, I/O channels, I/O devices, and files [3].

Since processes, in general, do not share memory or open files\(^2\), it is possible to execute different processes in a distributed application on different nodes in the distributed system.

The different between a process and a program is a subtle, but important one. An analogy that may help the reader is presented in [4], section 2.1.1.

2.1.3. Threads

Sometimes, threads are called *lightweight processes*, as opposed to the normal, *heavy-weight processes*. This is because creating a new thread is faster and requires less memory than creating a new process, and because context switches among threads are faster than among processes.

Traditionally, only a single path of execution could be active in a process at any given time. In other words, the process contained only a single thread of execution, and the unit of resource ownership (the process) was the same as the unit of scheduling/execution.

In modern operating systems, however, these two units are treated independently. When a process starts, a single thread of execution is created (just like in the traditional operating systems), but it is possible for the programmer to create multiple threads in the same process. Each thread can follow its own path of execution through the program, and all threads can execute simultaneously if the computer has enough processors available. If it has not, the operation system simulates simultaneous execution by timesharing the processor(s) between threads.

When running multiple threads or processes on a computer with only one CPU, the processor has to be timeshared between the threads or processes. The act of changing from one thread or process to another is called a *context switch*. A context switch for a thread involves saving the CPU state of the old task and reloading the CPU state of the new task, and possibly mapping in memory for the new task's stack and thread-local storage. For a process, the memory pages of the process must be mapped in as well, and this is what makes switching processes slower than switching threads.

Each thread shares the process' memory and open files, but has its own *execution context*. The execution context is the information required to keep track of where in the program

\(^2\)An operating system usually has provisions for making two processes share a memory area, but this must be explicitly enabled by the programmer. In addition, when a process creates a child process, that child usually inherits the open files of its parent. The child can choose to close any open files or open new ones without affecting the parent, though.
the thread is execution. It includes the program pointer, registers, and stack.

Since threads in the same process do share memory and open files, it is not possible to execute different threads of the same process on different nodes in the distributed system.

Figure 2.1, adapted from [4], shows what is shared between threads in a process, and what is private for each thread.

2.2. Process Management

A distributed application consists of multiple processes running on different nodes in the distributed system. Consequently, there is a question of who is responsible for creating the correct process on the correct node, and who is responsible for checking that the processes run and that they are terminated at the appropriate time.

2.2.1. User Manages Processes Manually

In some distributed applications, the user has to create all processes that constitute the application manually, either by logging into the different nodes and running each process (e.g. from the command line), or by creating a script that automates this (sometimes, the application programmer provides this script). The user is also responsible for monitoring the running processes and taking appropriate steps should a process terminate unexpectedly (for example, if the node running the process crashes).

2.2.2. Language Manages Processes

In some programming languages, processes or threads are supported by the language itself. The language compiler or interpreter, together with the language run-time-system, takes care of the details of process creation and management.

There are two kinds of support for processes in languages. In some languages, the concepts of processes or threads is built into the language, and it has explicit primitives for
defining, creating and managing processes or threads. Examples of languages that fall into this category are Ada, ORCA, and in part Java\textsuperscript{3}. Some other languages do not have concepts which correspond directly to processes or threads. Instead, threads or processes are created and used implicitly, in order to implement some other language feature. An example of this is true object-oriented languages, where objects are autonomous units which communicate by sending messages to each other. In these languages, each object could be given its own process or thread.

2.2.3. Library Software Manages Processes

Some communication libraries (e.g. MPI) or clustering software (e.g. Beowulf) has special software that can start the processes that constitute the application manually, by invoking one single command on one of the nodes. The user still has to list all processes in the application, and for each process, determine on which node to run the process.

The library or clustering software may also have provisions for monitoring process state and restarting a process or take other action if a process or an entire node fails.

2.2.4. Operating System Manages Processes

True distributed operating systems, like Amoeba, can transparently start the processes of an application on nodes in the distributed system. The application programmer programs his or her application has if it consisted of multiple processes running on one one physical computer, and the operating system determines what node to run the process on automatically — usually by monitoring the system load of the nodes and selecting the one with the lowest load.

2.3. Communication Mechanisms

If the processes in a distributed application are to cooperate meaningfully, they need to have some means of exchanging information with each other. In this section the most common communication mechanisms for distributed systems are covered.

Note that in this section we use the term “message passing” to mean explicit message passing where the programmer must assemble data into a message, send it using the send primitive, receive it using the receive primitive, and lastly unassemble it, and where message boundaries are preserved. Another possible interpretation of the term is that message passing is all kinds of information transfer from a sender to a receiver via a (physical or virtual) network. In that case all non-shared-memory communication mechanisms covered in this section fall into the message passing category.

\textsuperscript{3}In part because although the Thread class is part of the Java class library, and not the Java language itself, the language has support for thread synchronization through synchronized methods and variables.
2.3.1. Shared Memory

If processes in a distributed application can share a memory region, that memory region can be used for communication between the processes. This is an efficient and commonly used form of communication when the processes run on the same computer. (The computer could be a multiprocessor computer, capable of running more than one process on the same time.) However, in a distributed system, memory is not shared between the nodes in the system, and so this manner of communication is not available. Instead, the processes has to communicate through explicit software messages.

There are some libraries and languages that attempt to simulate shared memory in systems that do not have physically shared memory. These are called distributed shared memory environments, and map accesses to the virtually shared memory region into messages that are transmitted via the network. Examples of distributed shared memory environments are Linda [5] and OpenMP [6].

2.3.2. Pipes or Sockets

A pipe is a unidirectional communication channel that links two processes. Usually these two processes must be running on the same node\(^4\). One process acts as the writer and writes data to the pipe, and the other acts as the reader and reads data from the pipe, and the processes cannot suddenly reverse roles using the same pipe.

Data written to the pipe by one process can be read back by the other, and data always arrive in the same order as it was sent. Message boundaries are not preserved. In other words, if process A writes two chunks of 128 bytes and process B reads one chunk of 256 bytes, it will get the 256 bytes written by process A as a contiguous block of data.

A pipe is implemented as a circular buffer. When a process tries to write something into the buffer, the request is executed immediately if there is sufficient room; otherwise the process is blocked (i.e. stops running) until there is enough room. Similarly, when a process tries reading a number of bytes from the buffer, the request is executed immediately if there is at least that number of bytes of data available in the buffer; otherwise, the process blocks until enough data becomes available.

Sockets are similar to pipes\(^5\), but can be used for network communication. In addition, sockets are bidirectional, and both processes can read from and/or write to a socket at any given time (logically, you can think of a socket as a pair of pipes, one for each direction). Figure 2.1 illustrates pipes and sockets.

Pipes and sockets were first implemented in BSD Unix, but are now supported in most operating systems, including GNU/Linux, most variants of UNIX, and Microsoft Windows.

\(^4\)The "Named Pipes" implemented in Microsoft Windows NT can be used for communication between processes located at different nodes.

\(^5\)At least, stream-oriented sockets are like pipes. Datagram-oriented sockets are more like message-passing.
Using the sockets interface, it is possible to create applications that use a wide range of networking protocols; the most commonly-used of these are the internet protocols TCP and UDP.

### 2.3.3. Message Passing

Message passing is a method of interprocess communication that uses two primitives, `send` and `receive`. Typically these are implemented as library procedures or system calls and may look something like this:

```c
send(destination, &message)
```

```c
receive(source, &message)
```

The first call sends a `message` to a given `destination` and the latter one receives a message from a given `source` (or from `ANY`, if the receiver does not care). If no message is available, the receiver can block until one arrives, or return immediately with an error code.

Unlike pipes and sockets, with message passing, message boundaries are preserved. Two messages with 128 bytes of data each will be received as two messages with 128 bytes, and not as one message with 256 bytes. Like sockets and unlike pipes, message passing can be used for network communication (and frequently is).

Examples of message-passing libraries, widely used for scientific computing, are MPI [7] and PVM[8].
2.3.4. Remote Procedure Call

The message-passing model provides a simple way of communicating between processes, but it has one serious shortcoming: the basic model around which all communication is built is input/output. The procedures send and receive are fundamentally engaged in doing I/O, and for many purposes, this is an unsuitable programming model.

Remote Procedure Calls (RPC) present an attractive alternative to the input/output model. The idea behind RPC is that you can call procedures located on other nodes. Seen from the programmer's perspective, remote procedure calls work just like normal (local) procedure calls. The programmer calls a procedure, and gets back a result. But the actual processing happens on another node, and the calling process is suspended until the called process has finished executing the procedure.

The actual steps in carrying out an RPC is as follows: When a procedure call is made, a stub procedure on the client (calling computer) takes the parameters, packs them (a process known as marshaling), and sends them over the network. A server stub procedure (also called a skeleton procedure) receives the parameters, unpacks them, and calls the appropriate server procedure. When processing is complete, the return value is transferred in the opposite direction using the same method.

Two well-known RPC standards are ONC RPC, which is used in the NFS (network file system), and DCE RPC, which is used in Microsoft DCOM\textsuperscript{6}.

For object-oriented programming languages, such as C++ and Java, RPC implementations like ONC RPC and DCE RPC has the important limitation that only procedure calls are supported, not method calls. This means that the part of the program that uses these RPC libraries cannot be programmed in an object-oriented style, but has to be programmed using the old function-oriented style (alternatively, one has to make function-oriented wrappers around object-oriented code). To overcome this limitation, there exists variation of remote procedure calls, called remote method calls.

2.3.5. Remote Method Call

Remote method calls (RMC, also called Remote Object Call) are remote procedure calls in an object-oriented context. RMC adds the concept of remote objects to the basic RPC model. These objects has methods that can be called in the usual way. This makes the RMC model blend in more naturally in an object-oriented programming language.

To the programmer, remote method calls works just like calling local object methods, but behind the scenes, an actual method call consists of the following steps:

1. The client requests a reference to a remote object. This can be done using some kind of directory service, via another remote object, via hardcoded references, or

\textsuperscript{6}Microsoft DCOM (Distributed Common Object Model) is an implementation of the remote method call paradigm, described in the section 2.3.5.
Figure 2.2: This figure illustrates the remote object calling process. The numbers inside circles refer to the calling sequence described in the text.

some other means. As a result of this process, a local stub object is created on the client. This object then be used to call the remote object’s methods.

2. The client calls one of the stub object’s methods.

3. The stub object marshals (packs) the method call. This involves packing the method name, object reference and method arguments in a network message, and sending them to the server via the network.

4. At the other end, a corresponding server stub object unmarshals the method arguments, determines which method to call, and invokes the method call on the appropriate object using the arguments received.

5. The method call executes, just like a local method call, and returns a value.

6. The return value is marshaled by the server stub object and sent back to the client.

7. The client stub object receives the return value and returns it to the calling client.

This process is very similar to the RPC process described in section 2.3.4, but with the addition of object references.

Examples of remote method call standards are CORBA[9], DCOM[10] and Java RMI[11].

2.3.6. Rendezvous

In the rendezvous model, e.g. used by the language Ada, two processes (called tasks in Ada) communicate with each other by synchronizing as follows: one task calls an entry
Figure 2.3: This figure illustrates the primary difference between the RPC (left) and the rendezvous (right) model. In the rendezvous model, the called task can execute concurrently with the calling task before and after the accept.

on the other one, and the latter responds with an accept for that entry. The two task then exchanges information. A task that arrives earlier is required to wait for the task that it needs to communicate with.

Rendezvous is similar to RPC/RMC in that one task can call another task, but is more general. In the RPC model, the called task acts as a server, waiting for incoming calls and serving them as they arrive. With the rendezvous mechanism, the calling task can do any computation before and after the accept (see figure 2.3).

Since the rendezvous mechanism is central to our approach to communication, it will be covered in more depth in section 2.5.

2.4. Synchronization

In addition to communication, most distributed applications need some way of synchronizing the various processes in the application. Synchronization is a facility that enforces mutual exclusion and event ordering[3], thereby ensuring correct execution in concurrent environments. This is especially important if two or more of the processes share some common resource (some examples of such resources are memory and files, hardware devices, and software services).

2.4.1. Synchronization Problems

If one ignores the questions of synchronization, or incorrectly uses synchronization primitives, problems can occur, including race conditions, deadlock and starvation.
2.4.1.1. Race Conditions

When two or more processes interact, the final result of the computation may depend on who runs exactly when. Such a condition is called a race condition. The following example illustrates how a race condition can happen.

Suppose we have a system in which one or more processes (which we will call producers) continuously enter data into a shared storage (this could be a memory buffer or a file), and one or more processes (which we will call consumers) continuously remove data from the same storage\(^7\). Data is always entered or removed in fixed-size chunks (which we will call data items), and the storage has a fixed capacity and can only hold a limited number of items (see figure 2.4).

To ensure that a producer doesn’t try to exceed the capacity of the storage, it will check the number of items already stored before trying to enter a new one. If the storage is full, the producer will not enter a new item, but will wait a while before trying again. Similarly, a consumer will check that there is at least one data item in the storage before trying to remove an item.

Now, suppose that two producers more or less simultaneously try to enter a new data item into the storage. Most of the time, this will work fine. But in rare cases, what might happen is this. One producer reads the number of items in the storage, and finds out that there is still room for one more item. However, before this producer gets a chance to enter the new data item, the other producer reads the number of items in the storage. It, too, finds out that there is room for one more item.

Both processes now believe that there is room for the data item that they are going to enter, but the storage only has room for one of them. Depending on how the application is implemented, it might crash, behave unexpectedly, or bring down the whole computer system with an out-of-resources condition.

Debugging an application with these errors can be difficult, because the final result of the application is dependent on which process runs exactly when. The program might work fine most of the time, but can behave unexpectedly in rare conditions which are

\(^7\)This is called the Producer/Consumer problem and is one of the classic interprocess communication problems often described in the literature in order to illustrate IPC issues. See, for example, [3] or [4] for more information about interprocess communication.
2.4.1.2. Deadlock

To prevent race conditions, one can use one of the synchronization mechanisms which will be covered in section 2.4.2. When applying these mechanisms, great care must be taken to avoid deadlock, which is when two (or more) processes prevents each other from gaining access to a resource.

For example if you have two processes, PA and PB, that both need access to two resources, RA and RB. The operating system assigns PA to RA and PB to RB. After PA is finished with RA it holds the resources until RB is free, but since PB does not release RB until it can get PA both are stuck in their state (see figure 2.5).

2.4.1.3. Starvation

Another problem that may arise when using synchronization mechanisms is starvation, which is when a program runs forever without making any progress. To explain starvation, a classic interprocess communication problem is used.

In the dining philosophers problem, there are five philosophers sitting around a circular table with five forks and five plates of spaghetti. A philosopher has two states: eating or thinking. Since the spaghetti is quite slippery each need two forks to eat, one to his right and one to his left, aquired one at a time in either order. If aquired successfully he eats for a while and then puts the forks down to resume his thinking. The problem is to construct a program doing so without getting deadlocked.

Assume that in a particular algorithm everyone tries to get their left fork first, and then their right fork. If they cannot get both forks they put their left fork down. When this program starts every philosopher notice that his left fork is available and picks it up, thus when each philopher checks his right fork it is not available, and everyone put their left fork down. Since their left forks are now available each philospher picks up his left fork again and the program will continue like this indefinitely. Although the program runs without getting stuck no progress is made, and since no one is able to eat, we have starvation.
2.4.2. Synchronization Mechanisms

There are seven important mechanisms to achieve synchronization of processes. These are spin lock, semaphores, mutexes, condition variables, monitors, message passing and barriers. A process has three states, running, runnable and blocked, and another process can change the state of a process by the use of these primitives, ensuring that no two processes are in their critical region at the same time.

2.4.2.1. Shared Memory Synchronization

To achieve synchronization when dealing with processes accessing a shared memory one can use spin locks, semaphores, mutexes, condition variables or monitors as primitives. These primitives enable one to obtain mutual exclusion and make sure the events are done in the right order.

- **Spin lock** is a software primitive where you use a lock variable that locks the resource when a process enters its critical region, and when the process leaves its critical region the resource becomes unlocked. A lock has two states: unlocked or locked. A locked resource makes it impossible for other processors to reach its critical region. A spin lock can be used to implement locking, but it is not failproof, and it relies on busy-waiting, using CPU power when running in a tight loop checking if the resource is unlocked or not.

- **Semaphores** attempt to address the problems caused by spin locks. A semaphore has two operations and a value. The down operation decreases the value by one. If it becomes zero, the process blocks. The up operation increments the value by one. If this causes it to increase from zero to one, one of the processes blocking on the semaphore is unblocked. The semaphore is an operating system primitive, and the operating system makes sure that the processes block and unblock at the appropriate time, avoiding busy-waiting. A problem with semaphores is that it they are low-level and can be complicated to use.

- **Mutex** is just a simplified version of the semaphore and is only useful for managing mutual exclusion to a shared resource. A mutex is a variable that has two states, lock and unlocked. Only one process can hold a mutex lock at any time. If processes try to lock an already-locked mutex they are blocked until the mutex is unlocked.

- **Condition variables** - A condition (short for "condition variable") is a synchronization device that allows processes to suspend execution until some predicate on shared data is satisfied. The basic operations on conditions are: **SIGNAL** the condition (when the predicate becomes true), and **WAIT** for the condition, suspending the process execution until another process signals the condition. A condition variable must always be associated with a mutex, to avoid the race condition where
a process prepares to wait on a condition variable and another process signals the condition just before the first process actually waits on it.

- A Monitor is a programming language construct that provides abstract data types and mutually exclusive access to a set of constructs.

### 2.4.2.2. Message Passing, Pipes and Streams

With processes that interact there are two requirements that must be satisfied, synchronization and communication. Processes must be synchronized to enforce correct execution and cooperative processes must interact to exchange information. Message passing provides both.

The minimum set of operations required for a message passing system are *send* and *receive*. Obviously, both the sender and receiver must be aware of each other and know when to receive a message and when to send. Both the sender and receiver can be blocking or nonblocking. There are three common combinations:

- **Blocking send, blocking receive**: Both the sender and receiver is blocked until the message is delivered.

- **Nonblocking send, blocking receive**: The sender continue on, but the receiver is blocked until the requested message is received. Since the sender is not required to wait until the message is received it can carry on and do other useful tasks, which makes this a very useful combination.

- **Nonblocking send, nonblocking receive**: Neither is required to wait.

When it comes to pipes and streams the synchronization operates exactly as with message passing.

### 2.4.2.3. Barrier

This synchronization mechanism is meant for groups of processes. Some applications are divided into phases, where a process cannot start a new phase before all processes have finished the previous phase.

A typical example can be a large matrix, (like 1 million by 1 million) where processors work separately on different part of the matrix, calculating new matrix elements from old ones. By placing a *barrier* on the transition between phase \( n \) and phase \( n+1 \), one ensures that no process begins the new iteration before all processes have finished the old one. When the processes hit the barrier they simply block and wait until every process reaches it.
2.4.2.4. Remote Procedure Calls (RPC), Remote Method Calls (RMC),
Rendezvous

The basic synchronization mechanism in RPC, RMC and rendezvous communication is
the remote call itself. The calling process is blocked until the remote call has completed
and the remote process has sent a reply back to its caller. This is very similar to message
passing synchronization. The sender of a remote call request blocks until a reply has been
received, and the receiver of a remote call blocks until it actually receives the request.

2.5. The Rendezvous Concept

As stated in section 2.3.6, in the rendezvous model, two processes or tasks interact by first
synchronizing, then exchanging information and, finally, by continuing their individual
activities [12]. The rendezvous is both a synchronization and a communication primitive²,
and can be said to consist of two phases. First, tasks synchronize, then they communicate.

Figure 2.6 illustrates the steps involved in making a rendezvous between two tasks A
and B that are running independently. First, task A calls an operation of B⁹, and B accepts
the operation call. The task that calls or accepts first is required to wait
for its counterpart. When the call and the accept is complete, the two processes are
synchronized.

The last step of the rendezvous is that the two processes communicate with each other.
A rendezvous is called simple if information is only passed in one direction (from the
caller to the receiver), and extended if the information exchange is bidirectional. After
the rendezvous is complete, the two tasks can start running independently again.

2.5.1. Rendezvous in programming languages

Several programming languages incorporates rendezvous as a mechanism of synchronization
and communication. In this section, we will compare the implementation of ren-
dezvous in four languages: Ada, Concurrent C, SR and JR. For each language a simple
programming example will be given.

2.5.1.1. Terminology

In different implementations of the rendezvous model, the terminology is used differently.
Table 2.2 shows the difference between the terminology used in Ada, in Concurrent C,
in SR and JR, and in the Rendezvous Library that we will introduce in chapter 3.

²M. E. Conway was the first to recognize that communication and synchronization is two inseparable
activities. In [15], he defined coroutines, which was the first high-level synchronization mechanism.
⁹An operation is called an entry in Ada and a process in Concurrent C.
Figure 2.6: The steps involved in a rendezvous.

<table>
<thead>
<tr>
<th>Ada task</th>
<th>Concurrent C process</th>
<th>SRand JR process</th>
<th>Rendezvous Library task</th>
</tr>
</thead>
<tbody>
<tr>
<td>extended rendezvous</td>
<td>transaction</td>
<td>call/in (rendezvous)</td>
<td>rendezvous</td>
</tr>
<tr>
<td>entry declaration</td>
<td>transaction declaration</td>
<td>operation declaration</td>
<td>operation declaration</td>
</tr>
<tr>
<td>entry call</td>
<td>transaction call</td>
<td>operation call</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Terminology differences

2.5.1.2. Ada

The programming language for which the use of the rendezvous concept is most well known is Ada [13]. The use of rendezvous as a primitive for synchronization and communication was inspired by the concept of *distributed processes*, introduced by Hoare [15] and the concept of *communicating sequential processes* introduced by Brinch Hansen [16].

In order to show how the rendezvous concept is used in Ada, a concurrent programming example is given in figure 2.7. The problem is to implement a simple version of the producer-consumer problem. One task, the producer, reads characters from standard input (e.g. a computer terminal), and sends them to another process, the consumer, which prints the uppercase versions of these characters on standard output (e.g. a computer screen). The code example is shown in the style traditionally used for Ada programs, with keywords in **bold** and the rest of the program in a Roman font.

The program text begins by importing the routines for text input/output (line 1) and for converting a character to upper case (line 2). Then the program itself is declared as the procedure CONVERT_TO_UPPER_CASE (line 3). Inside the declarations section of the procedure, the tasks are defined.

A task in Ada consists of a task *declaration* and a task *body*. The task declaration defines the name of the task and lists the operations that can be called by other tasks (the operations are called *entries* in Ada). Only the names and parameters of the entries
with TEXT_IO; use TEXT_IO;
with TO_UPPER;
procedure CONVERT_TO_UPPER_CASE is
  task PRODUCER;
  task CONSUMER is
    entry SEND(C: in CHARACTER);
  end CONSUMER;
  task body PRODUCER is
    C: CHARACTER
  begin
    while not END_OF_FILE(STANDARD_INPUT) loop
      GET(C);
      CONSUMER.SEND(C);
    end loop;
    CONSUMER.SEND(ASCII.EOT);
  end PRODUCER;
  task body CONSUMER is
    X: CHARACTER;
  begin
    loop
      accept SEND(C: in CHARACTER) do
        X := C
      end SEND;
      if X = ASCII.EOT then
        exit;
      end if;
      PUT(TO_UPPER(X));
    end loop;
  end CONSUMER;
begin — — PRODUCER AND CONSUMER become active
null; — — subprogram body must have at least one statement
end CONVERT_TO_UPPER_CASE;

Figure 2.7.: Ada version of the producer-consumer problem (taken from [12]). The line numbers are not part of the program text, they are included just for reference.
are listed here; the statements that will be executed upon acceptance of the entries are defined within the task body itself.

The PRODUCER task is declared in line 4, and the CONSUMER task is declared in lines 5-7. In the CONSUMER task, an entry SEND is defined (line 6). This entry will be called by the PRODUCER in order to transfer a single character read from the input to the CONSUMER through a rendezvous (the entry takes a single parameter, C, that is the character to be transferred). The PRODUCER task doesn’t have any entries.

Lines 8-16 constitute the body of the PRODUCER task. The producer runs in a loop, reading one character from standard input, and sending it to the consumer, until the end of input is encountered. Lines 13 and 15 are examples of entry calls in Ada. In the first case, the character read is sent to the consumer, and in the second case an end-of-file marker is sent to indicate that the consumer should finish executing.

The body of the CONSUMER task is defined in lines 17 to 28. The consumer, like the producer, runs in a loop. First the entry SEND, which was called by the producer in lines 13 and 15, is accepted in line 21. In line 22, the character received (C) is stored in a local variable (X). The character is then checked to see if it is the end-of-file marker. If it is, the consumer exits, thus ending the execution of the program (the producer has already finished after sending the end-of-file marker to the consumer). If it is not, the character is converted to upper case.

As can be seen from the example, in Ada tasks are declared as part of a procedure. The tasks start running automatically when the procedure starts, and in parallel with the procedure’s own statements (between the begin and end statements; in the example these are on lines 30 to 32 and the procedure only consists of the null statement, which does nothing). The procedure does not exit until all tasks have finished executing.

When an entry (such as the SEND entry in the example) is declared in a task, it can be called by other tasks with a syntax equal to that of calling a procedure. When the calling task calls an entry, it is suspended until the receiving task has finished executing the accept; that is, executed all statements between accept ... do and end (in the example that is between the lines 21 and 23).

When the receiving task calls accept, it is suspended until a task has called the corresponding entry. When a task calls the entry, the arguments that the calling task passed as part of the call is made available to the receiving task (the variable C in the example). The task then executes the statements in the entry until the end statement. When the end statements is reached, the calling task is unblocked and the caller and the receiver start executing independently again.

Note that the naming of tasks is asymmetric; the called names the called task explicitly in the entry call, while the receiver does not name the caller in the accept. Ada tasks also cannot be parameterized, therefore, the names of tasks must also be hardcoded in the program text.10

---

10This is not strictly true. As stated, Ada tasks cannot directly have parameters, but the generic facilities
2.5.1.3. Concurrent C

Concurrent C\cite{18} is an upwards-compatible extension to the C language\footnote{There exists a version called Concurrent C++ that is an extension to C++ instead of C. For more information on that version, see \cite{19}.} that adds facilities for concurrent programming to the language. Like in Ada, the concurrent programming facilities in Concurrent C are based on the rendezvous model, but there are some differences in the implementation of that model.

To show the difference between the concurrent programming facilities in the two languages, we will show an example. In figure 2.8, the same problem used in section 2.5.1.2 (producer-consumer) is implemented in Concurrent C.

The first two lines include the header files that define functionality for text input and output and for converting characters to upper case. Then the process specification of the consumer process is declared in lines 4-7. Process specifications are like the declaration sections of Ada tasks; they list the transactions (entries) that this process has, in the case of the consumer process, this is the send transaction.

The process specification of the producer process is on line 8. As in the Ada version, this process has no transactions. Unlike the Ada version, this process takes a parameter; namely the consumer process with which it will rendezvous. Passing parameters to processes is similar passing parameters to constructors in C++. Any parameters will be available as variables in the body of the process.

In lines 9-19 the process body of the consumer process is defined. The algorithm executed is the same as in the Ada version; the process starts by first accepting a character from the consumer (line 13), copying it to a local variable (line 14), testing if it is an end-of-file marker and exiting if it is (lines 15-16), and finally printing the character converted to upper case (line 17).

The process body of the producer process is defined in lines 20 to 26. It reads characters from input (line 23) and sends them to the consumer by calling the consumer’s send transaction (line 24). When end of input is reached, an end-of-file marker is sent (line 25).

Finally, a small main function is included (lines 27-31). The main function begins executing when the program starts (like in regular C and C++). This small function first declares an instance of the consumer process (called q), and starts it (activates it) using the create operator. The producer process is also started, and the consumer process q is passed as a parameter.

The differences between Ada and Concurrent C that are evident from the program examples can be summarized like this:

- In Concurrent C, processes are top-level objects in the same way that functions are, while in Ada the tasks must be contained within a procedure.

\footnote{In the Ada language can be used to statically parameterize processes\cite{12}.}
```c
#include <stdio.h>
#include <ctype.h> /* contains islower(), toupper() */

process spec consumer()
{
  trans void send(int c);
}

process spec producer (process consumer cons);
process body consumer()
{
  int ch;
  for (;;) {
    accept send(c)
    { ch = c; }
    if (ch == EOF)
      break;
    islower(ch) ? putchar(toupper(ch)) : putchar(ch);
  }
}

process body producer(cons)
{
  int c;
  while ((c = getchar()) != EOF)
    cons.send(c);
  cons.send(EOF);
}

main()
{
  process consumer q;
  q = create consumer(); create producer(q);
}
```

Figure 2.8: Concurrent C version of the producer-consumer problem, taken from [12].

The line numbers are not part of the program text, they are just for reference.
• It is not possible to pass parameters to tasks in Ada, but this is possible in Concurrent C.

• In Ada, the name of the called task had to be specified directly in the program text when calling an entry, but in the Concurrent C version, a transaction can be called via a variable that references the called process. In the example, the producer stores such a reference to the consumer in the cons variable. This reference was passed to the producer by the main program when creating the producer (line 30). Using references to processes makes it possible for a process to call one of several possible processes, depending on e.g. a parameter passed to the process upon creation.

• In Concurrent C, processes have to be explicitly started using the create operator. This was not necessary in the Ada version, where the tasks began running when the containing procedure started.

In addition, Concurrent C offers more flexibility when it comes to determining which transaction call will be accepted if several are waiting to be served in a given accept statement. While Ada only accepts entries in FIFO order, in Concurrent C one can base the order of acceptance on the parameters passed in the transaction.

Concurrent C also gives more control over the priorities of the various processes in a program. In Ada, all tasks of the same type are of equal priority, but in Concurrent C, one can set the priority of specific processes.

Lastly, Concurrent C lacks explicit facilities for passing variable length arrays as transaction or process parameters, or passing output and input/output parameters. Ada does not suffer from these limitations.

2.5.1.4. SR

SR[20] supports many features useful for concurrent programming; however the design goals have been to keep the language simple and easy to use, while at the same time providing an efficient implementation. This is achieved by integrating common notions, both sequential and concurrent, into a few powerful mechanisms. For synchronization and communication, SR provides dynamic process creation, semaphores, message passing, remote procedure calls, and rendezvous. However, all these mechanisms are provided through a single mechanism: the operation.

The main idea is that operations can be invoked in two ways, synchronously (call) or asynchronously (send); and can be serviced in one of two ways, by proc or by input statements (in). With procs, a new process is created to service the operation, with an input statement the operation is served in-line with other statements in an existing process. This yields the four combinations listed in table 2.3.

In order to illustrate how concurrent programming with rendezvous in SR works, an SR version of the producer-consumer problem is given in figs. 2.9 and 2.10.
1 resource consumer
2  op SEND(c: int)
3
4 body consumer()
5 procedure toupper(ch: int) returns uch : int
6  if ch >= 97 and ch <= 122 ->
7    uch := ch - (97 - 65 )
8  [] else ->
9    uch := ch
10  fi
11 end
12
13 process do_cons
14  var ch : int
15  do true ->
16    in SEND(c) ->
17      ch := c
18    ni
19    if ch = EOF ->
20      return
21    fi
22    printf("%c",char(toupper(ch)))
23  od
24 end
25
26 resource producer
27  import consumer
28 body producer(cons: cap consumer)
29  var c: char
30
31 process do_pro
d
32  do int(c) := EOF ->
33    if scanf("%c",c) = EOF) ->
34      exit
35    fi
36    call cons.SEND(int(c))
37  od
38    call cons.SEND(EOF)
39 end
40 end
41
42

Figure 2.9: SR version of the producer-consumer problem (producer and consumer resources)
<table>
<thead>
<tr>
<th>Invoke</th>
<th>Service</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>proc</td>
<td>(possibly remote) procedure call</td>
</tr>
<tr>
<td>call</td>
<td>in</td>
<td>rendezvous</td>
</tr>
<tr>
<td>send</td>
<td>proc</td>
<td>dynamic process creation</td>
</tr>
<tr>
<td>send</td>
<td>in</td>
<td>asynchronous message passing</td>
</tr>
</tbody>
</table>

Table 2.3.: The four combinations of invocation and servicing of operations in SR

```plaintext
44 resource main()
45 import producer
46 import consumer
47 var prod : cap producer
48 var cons : cap consumer
49
50 cons := create consumer();
51 prod := create producer(cons);
52
53 final
54 destroy cons
55 destroy prod
56 end
57 end
```

Figure 2.10.: SR version of the producer-consumer problem (main resource)
In lines 1 - 26 the **consumer resource** is defined. An SR resource is a modular component that is used to group related functionality, much like procedures in Ada or classes in C++ (Unlike procedures, and like classes, resources must be instantiated before they are used). The resource contains two parts: a specification, and an implementation (or a body). The specification of a resource contains declarations of types, constants and operations that are made available to other resources through the use of an `import` statement. The body of a resource can contain additional declarations (that won’t be made available to other resources), and also contain code that, among other things, services operations.

The consumer resource defines one operation, the `send` operation that will be called by the producer in order to transfer characters from producer to consumer (the operation name is in upper case because `send` is a reserved word in SR). This operation takes one parameter, the character (an integer) that will be sent.

The `toupper` procedure, that converts a character to upper case, is defined in lines 5-11. It is included because SR doesn’t have this procedure built-in.

In lines 13-25 the `do_cons process` is defined. This process implements the consumer’s functionality. Like tasks in Ada, SR processes are started automatically when the enclosing resource is created (a process is a proc with an implicit `send` to it when the containing resource is created). In line 16, a character is received from the producer through the `in` statement. An `in` statement constitutes the callee’s side of the rendezvous, and works just like the `accept` keyword in Ada or Concurrent C. The character, converted to upper case, is printed to standard output.

The **producer resource** is defined in lines 28-42, including the `do_prod process` defined in lines 33-41. This process reads a character from standard input and sends it to the consumer with a `call` statement in line 38. This constitutes the caller’s side of the rendezvous, and works like transaction calls in Concurrent C and entry calls in Ada. In a call, the caller blocks until the callee completes servicing of the called operation.

In lines 44-57, the **main resource** is defined. This creates the other resources, so that their processes may start, and destroys the resources when the processes have finished executing.

Compared with Concurrent C and Ada, the following differences are evident:

- Rendezvous in SR is implemented with a small set of general primitives, and SR can support other styles of synchronization and communication, like remote procedure calls, or message passing, by combining these primitives in other ways. In Concurrent C and Ada, rendezvous is the primary primitive for synchronization and communication.

- In SR, processes are contained within a resource. This is like Ada, where tasks are contained in a procedure, and unlike Concurrent C, where processes are top-level objects.

- Unlike Concurrent C and Ada, operations are defined as part of a resource, and
not as part of a process/task, and they can be serviced by any process within the resource, or a new process may be spawned to service the operation.

- It is possible to pass parameters to resources, like in Concurrent C and unlike in Ada.

- An operation can be called via a variable that references the called resource, like in Concurrent C. In SR, this kind of variable is called a capability variable.

- In SR, resources have to be explicitly started using the `create` statement, but processes within the resource are created automatically. This can be said to be a hybrid of the approaches used by Concurrent C and Ada.

2.5.1.5. JR

SR has a rich concurrency model, but lacks many sequential programming features that are useful for complex programs. It is not object-oriented, and lacks the rich standard libraries of languages like Java and C++. JR[21] is an extension to Java that adds the concurrency primitives of SR to the language. It has been designed to integrate the SR concurrency model with Java in a manner that retains the “feel” of Java, while supporting all concurrency primitives of SR (including `send` and `call`, input statements, and processes). It is implemented as a translator that converts JR programs into standard Java, and a run-time system (written in Java) that supports execution of these programs. The run-time system uses RMI to transfer information between processes.

To illustrate the differences between JR and SR, the producer-consumer example given in figs. 2.9 and 2.10 has been ported to JR. The result is given in figs. 2.11-2.13. The concepts used and the basic structure of the program is the same, but there are some differences between the two languages, besides the differences in syntax and in the sequential programming features.

- In JR, Java classes take the place of SR resources, and Java methods are used instead of SR proc.

- A new object (which can be remote, i.e. located in another Java Virtual Machine) is created with the `new remote` operator. This corresponds to the SR `create` operator.

- References to remote objects have the type `remote classname`. In SR, they have the type `cap classname`.

- The input statement is called `inmi` in JR, and uses Java block syntax. For an operation that is to be serviced with the `inmi` statement, the complete operation signature must be given, not just the operation name and actual parameters.
public class consumer {
    public op void SEND(int c);

    public consumer() {
    }

    process do_cons {
        int ch = 0;
        while(true) {
            try {
                inni void SEND(int c) {
                    ch = c;
                }

                if(ch == prodcon.EOF) {
                    send prodcon.done();
                    return;
                }
            }
            catch(Exception e) {
                System.err.println("Exception in \'inni\' statement");
                e.printStackTrace();
            }

            System.out.println(Character.toUpperCase((char)ch));
        }
    }
}
public class producer {
  private remote consumer cons;
  private int c;

  public producer(remote consumer the_cons) {
    c = 0;
    cons = the_cons;
  }

  process do_proc {
    try {
      while( ( c = System.in.read() ) != prodcon.EOF ) {
        call cons.SEND(c);
      }
      call cons.SEND(prodcon.EOF);
      send prodcon.done();
    } catch(Exception e) {
      System.err.println("Exception in 'call' statement");
      e.printStackTrace();
    }
  }
};

Figure 2.12: JR version of the producer-consumer problem (producer.jr)
import edu.ucdavis.jr.JR;

public class prodcon {
    public static int EOF = -1;
    public static void done();

    public static void main(String[] args) {
        try {
            remote consumer cons = new remote consumer();
            remote producer prod = new remote producer(cons);

            inni void done() {}; // Wait for the first process to finish
            inni void done() {}; // Wait for the second process to finish
        }
        catch(Exception e) {
            System.err.println("Exception when creating/executing remote objects");
            e.printStackTrace();
        }
    }

    try {
        JR.exit(0);
    } catch(Exception e) {
        System.err.println("Exception when calling JR.exit()");
        e.printStackTrace();
    }
}

Figure 2.13: JR version of the producer-consumer problem (prodcon.jr)
- Input statements, remote object creation, \texttt{calls} and \texttt{sends} all throw exceptions if an error occurs, as does many of the other features added by JR. These must be caught using the Java \texttt{try \ldots catch} construct.

- The current implementation of JR doesn’t support \texttt{final} blocks. In SR, a resource’s \texttt{final} block is executed when all processes in the resource have finished executing. JR is unable to detect this condition, and this is why, in the JR version, the \texttt{do\_cons} and \texttt{do\_prod} processes send an explicit \texttt{done} message to the \texttt{prodcon} class to signal that they have finished.

For a more detailed feature comparison between SR and JR, see [22].

\section{Concurrent Programming Libraries}

In the previous section, several programming languages that implement the rendezvous mechanism have been covered. Other programming languages exist, that offers alternative concurrent programming primitives.

It is also possible to implement concurrent programming facilities as an add-on library to existing, sequential programming languages such as C, C++, FORTRAN, or Java. In this section we cover one low-level and two high-level libraries that can be used for concurrent programming. For each library, we will describe the mechanisms for process management, initialization and cleanup, communication, and synchronization.

Lastly, we will briefly compare the three libraries with respect to usability and performance (a definition of these terms can be found in section 3.2.1).

\subsection{TCP/IP Sockets}

The IP (Internet Protocol) family of networking protocols — TCP, UDP, ICMP, and others — has over the last two decades become the most widely used protocol family in the world, and is the foundation on which the Internet is built. TCP (Transport Control Protocol) is an connection-oriented protocol built on top of the datagram-oriented IP protocol. The data transferred over a TCP/IP connection is guaranteed to arrive in order and without error.

The \texttt{sockets} interface to TCP/IP programming[40] was first introduced with the 4.1c release of BSD UNIX in late 1982. Sockets are a generalized networking capability which enables communication between processes running on the same computer or on different computers. They are not limited to TCP/IP communication, or even to the IP family of protocols, and can be used with both connection-oriented and datagram-oriented protocols using the same set of function calls. Only connection-oriented communication using TCP will be covered in this section.
2.6.1.1. Process Management

The sockets library contains no provisions for process management. When creating an application that consists of a set of communicating processes that will be run on a distributed system, it is the developer’s responsibility to ensure that all processes start up on the desired nodes, using whatever operating-system provided method for running programs or creating processes that is available.

Creating processes on the local computer is possible using system calls such as (on UNIX) fork and exec. One common way of creating remote processes is to execute a remote shell command to the desired node using either the rsh utility or its secure alternative, ssh. Another way is to create a special server (or daemon) program on each node, that takes care of creating processes.

2.6.1.2. Initialization and cleanup

Before communication between two processes can be performed, each process must create an endpoint (or a socket) from which to communicate with the process. For a connection-oriented protocol such as TCP, it must also be determined which process will initiate the connection. The process initiating a connection is called the client, and the process accepting an incoming connection is called the server.

For server processes, the following initialization sequence must be performed.

1. Create the socket using the socket function.

2. Bind the socket to a local address using the bind function. This is done so that clients can know which address to contact the process at. After the socket is bound to an address, the process owns that address, and can receive incoming connections on that address.

3. Set the socket to a listening state using the listen function. When a socket is set to a listening state, it can accept (possibly multiple) incoming connections. An incoming connection is not accepted until the accept function in step 4.

4. Accept an incoming connection using the accept function. This function blocks until a connection is actually requested by a client. When this happens, the accept call returns a new socket that can be used for communication with the client. The original socket can be used to accept other incoming connections.

5. The new socket is now ready for use, and can be read from and written to.

For client processes, the initialization sequence is a bit simpler.

1. Create the socket using the socket function.
2. (Optional.) Bind the socket to a local address using the **bind** function. If this step is omitted, the operating system will pick a free local address.

3. Create a connection by using the **connect** function. As parameters, pass the address of the server (that the server binds to in step two of the server initialization procedure).

4. When the **connect** call returns, the socket is ready for use.

When the processes have finished using their sockets, they must free them by using the **close** system call.

---

### 2.6.1.3. Communication

The communication mechanism used for socket communication is that of a bi-directional data channel, as described in section 2.3.2. Data written to the socket at one end of the channel, can be read from the socket at the other end of the channel, and vice versa.

Data can be written to a socket by using the **send** function call \(^{12}\). The C-language prototype for send looks like this[23]:

\[
\text{int send(int s, const void *msg, size_t len, int flags);}\]

\(s\) is the file descriptor for the socket to send to and \(msg\) is a pointer to an array of bytes that holds the data to be sent. \(len\) is the number of bytes to send, and \(flags\) are optional flags that can be set to control how to transmit the data. For example, if to send the string “Hello World!” (and a null-terminating character) from a socket, one would issue the following call:

\[
\text{const char message[]} = \text{“Hello World!”};
\]

\[
\ldots
\text{send(s, (const void *)message, sizeof(message), 0);}\]

Here \(s\) is the socket to send to, and \(message\) is the buffer holding our data. The \(flags\) parameter is set to 0 to indicate that no special processing is to be done.

To read data from a socket, the **recv** function call can be used\(^{13}\). The C-language prototype for read looks like this[24]:

\[
\text{int recv(int s, void *buf, size_t len, int flags);}\]

\(^{12}\)Actually, since a socket is a normal file descriptor (at least in UNIX), one could use the **write** system call to send data. But it is often better to use **send** since it enables one to set options that control how the data is to be transmitted.

\(^{13}\)In operating systems such as UNIX where sockets are file descriptors, the standard **read** system call can also be used.
As an example, to read 13 bytes from a socket into a buffer, the following could be used:

```c
char message[13];
...
read(s, (void *)message, sizeof(message), 0);
```

Note that message boundaries are not preserved when sending or receiving messages to/from a connection-oriented socket. This means that if one wanted to read the "Hello World!" string sent in the `send` example, and didn't know how long that string was, one would have to either read character after character until the terminating null character was found, or include the length of the string in the message sent.

**2.6.1.4. Synchronization**

The synchronization mechanism available for TCP/IP socket programs is message-passing synchronization, as described in section 2.4.2.2. The `read` call blocks until the requested number of bytes are available for reading from the socket, or an error occurs.

Many of the socket programming calls can block, this is a partial list:

- `connect` blocks until the connection has been set up (that is, until the server accepts the connection).
- `accept` blocks until an incoming connection has been received.
- `send` blocks until the requested number of data bytes has been transferred to the operating system's network communication subsystem.
- `recv` blocks until the requested number of data bytes is available for reading from the given socket.
- `select` (not covered in this section) can be used to monitor up to three sets of sockets. It blocks until one or more sockets in the `read` set is available for reading, or one or more sockets in the `write` set is available for writing, or an exception has occurred on one or more sockets in the `except` set.

In addition, the blocking functions return when an error has occurred (for example, trying to read from a closed socket).

Note that there are no provisions for making the sender block until the receiver has finished processing the data; the sender only blocks until it has transferred the data. If one wish to use rendezvous-style communication, one must implement this e.g. by programming the receiver to send a reply message to indicate that the reciever has received and processed the data.

It is possible to instruct a socket to be non-blocking (e.g. by setting a flag with the `fcntl` system call). In that case, if a call would block, it returns an error code instead.
2.6.2. MPI/OOMPI

MPI — Message Passing Interface — is one of the two most popular high-level message passing systems (the other one being PVM[8]). It was standarized by the MPI Forum, and the standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. The preliminary draft proposal, MPI-1[7], was out first in 1992 and later revised in 1993.

The MPI Forum’s goal for the Message Passing Interface is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

OOMPI[25] (Object Oriented MPI) is an object oriented C++ class library to the Message Passing Interface. It is implemented as a lightweight layer on top of an existing MPI implementation, and provides all functionality defined by the MPI-1 standard. All standards-complying MPI implementations can usually be used with OOMPI.

2.6.2.1. Process Management

MPI includes software that can be used to start processes on desired nodes in a cluster. Each installation of MPI has a database (a text file) that lists the available nodes in the cluster and the addresses of each. When starting an MPI application, one uses the mpirun command, and specifies on the command line which nodes to start the application’s processes on. The mpirun command then takes care of starting processes on the desired nodes.

Some examples of mpirun command invocations are given below.

- Start *application* on nodes one and two.

  mpirun n1,2 application arguments...

- Start *application* on CPU’s one, three and five (each node can have multiple CPU’s).

  mpirun c1,3,5 application arguments...

- Start *application* on all available nodes in the cluster

  mpirun N application arguments...

This style of invocation is typically used when the application is an SPMD\(^{14}\)-type application, that is, when the same program is used for all processes in the application.

\(^{14}\)Single Program Multiple Data
The “arguments...” parameter is the command-line arguments that are passed the application processes. The application is the name of an executable program that uses MPI.

The mpirun command can also be invoked with the name of an application schema, like this:

```
mpirun appschema_name
```

An application schema is a text file that lists the names of the executable programs that the application consists of, and the nodes on which to start each program. This form of the mpirun command is usually used when the application is a MPMD\textsuperscript{15}-style application, that is, when different programs are used for different processes in the application.

For starting processes on remote nodes, the mpirun command either uses rsh or ssh, or a special MPI daemon that is started on each node. This varies with the MPI implementation.

OOMPI, being built on MPI, uses the same mechanism as MPI for process management.

2.6.2.2. Initialization and cleanup

Before using any other MPI functionality, an MPI program must call the \texttt{MPI_Init()} function call. This call processes any MPI-specific command-line arguments and initializes the MPI subsystem. The equivalent call for OOMPI is \texttt{OOMPI_COMM_WORLD.Init()}.

When initialization is complete, the MPI process can communicate with all other processes in the application. Each MPI process gets a unique rank (an integer number), and can address other MPI processes in the same application using their rank.

Before exiting, and MPI program must call \texttt{MPI_Finalize()}, which shuts down the MPI subsystem and does any necessary cleanup. For OOMPI, the equivalent call is \texttt{OOMPI_COMM_WORLD.Finalize()}.

2.6.2.3. Communication

The basic mechanism for communication in MPI and OOMPI is message passing, as described in section 2.3.3. The libraries support a wide range of communication primitives, both for blocking and non-blocking communication, and for peer-to-peer and collective communication.

Note that MPI and OOMPI also support many other calls for communication than those listed below; these are usually variations or combinations of the above calls.

\textsuperscript{15}Multiple Programs Multiple Data
Peer-to-Peer Communication  For peer-to-peer (one sender and one receiver) communication, it is possible to combine blocking or non-blocking sends with blocking or non-blocking receives in four different ways. The MPI calls for these four combinations are given below. The code examples send a message which is an array of icount floating-point-numbers, contained in sendbuf, from the process with rank 0 to the process with rank 1. The message will be received in recvbuf, which is an array with the same number of elements.

- **Blocking send and blocking receive:** For a blocking send, use MPI_Send. For a blocking receive, use MPI_Recv.

  ```c
  if (myrank==0) 
      rc = MPI_Send(sendbuf, icount, MPI_REAL8, 1, itag, 
                     MPI_COMM_WORLD);
  else if (myrank==1) 
      rc = MPI_Recv(recvbuf, icount, MPI_REAL8, 0, itag, 
                     MPI_COMM_WORLD, &status);
  ```

- **Non-blocking send and blocking receive:** For a non-blocking send, use MPI_Isend. The MPI_Isend call returns immediately after starting the transmission. It is possible to later wait until the transmission is complete by using the MPI_Wait function call. For blocking receives, use MPI_Recv as before.

  ```c
  if (myrank==0) {
      rc = MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag, 
                     MPI_COMM_WORLD, &ireq);
      rc = MPI_Wait(&ireq, &status);
  } else if (myrank==1) {
      rc = MPI_Recv(recvbuf, icount, MPI_REAL8, 0, itag, 
                     MPI_COMM_WORLD, &status);
  }
  ```

- **Blocking send and non-blocking receive:** For blocking sends, use MPI_Send as before. For non-blocking receives, use MPI_Irecv. The MPI_Irecv call returns immediately, without waiting for the data transmission to complete. It is possible to later wait until all data has been transmitted by using the MPI_Wait function call.

  ```c
  if (myrank==0) {
      rc = MPI_Send(sendbuf, icount, MPI_REAL8, 1, itag, 
                     MPI_COMM_WORLD);
  } else if (myrank==1) {
  ```
rc = MPI_Irecv(rcvbuf, icount, MPI_REAL8, 0, itag, 
    MPI_COMM_WORLD, ireq);  
rc = MPI_Wait(ireq, istatus, ierr); 
} 

- **Non-blocking send and non-blocking receive**: For non-blocking sends and non-blocking receives, use MPI_Isend and MPI_Irecv.

```c
if (myrank==0) {
    rc = MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag, 
        MPI_COMM_WORLD, ireq);
} else if (myrank==1) {
    rc = MPI_Irecv(rcvbuf, icount, MPI_REAL8, 0, itag, 
        MPI_COMM_WORLD, ireq);
    rc = MPI_Wait(ireq, istatus, ierr);
}
```

In OOMPI, one uses methods of the OOMPI_Port class to send and receive information. Examples that are equivalent of the MPI examples are given below. Note the much simpler syntax available when using OOMPI instead of MPI.

- **Blocking send and blocking receive**.

```c
if (myrank==0)
    OOMPI_COMM_WORLD[1].Send(sendbuf, icount);
else if (myrank==1)
    OOMPI_COMM_WORLD[0].Recv(rcvbuf, icount);
```

- **Non-blocking send and blocking receive**.

```c
if (myrank==0) {
    OOMPI_Request r = OOMPI_COMM_WORLD[1].Isend(sendbuf, icount);
    r.Wait();
} else if (myrank==1) {
    OOMPI_COMM_WORLD[0].Recv(rcvbuf, icount)
}
```

- **Blocking send and non-blocking receive**.

```c
if (myrank==0) {
    OOMPI_COMM_WORLD[1].Send(sendbuf, icount);
```
else if (myrank==1) {
    OOMPI_Request r = OOMPI_COMM_WORLD[0].IRcv(recvbuf, icount);
    r.Wait();
}

- Non-blocking send and non-blocking receive.

if (myrank==0) {
    OOMPI_Request r = OOMPI_COMM_WORLD[1].Isend(sendbuf, icount);
    r.Wait();
} else if (myrank==1) {
    OOMPI_Request r = OOMPI_COMM_WORLD[0].IRcv(recvbuf, icount);
    r.Wait();
}

**Collective Communication**

As previously mentioned, MPI and OOMPI also support **collective communication**, that is, communication involving a group of processes (instead of just two). The collective communication patterns supported by MPI and OOMPI are broadcast, scatter, gather, all-to-all broadcast, and all-to-all personalized exchange.

- **Broadcast** communication involves a single sender and multiple receivers. The same message block is transmitted to a number of receivers. Only one message is transmitted in a single communication. (This assumes that the nodes are on the same subnet on a bus network, such as Ethernet. If this condition is not true, one message per receiving node may be required.) In MPI, a message is broadcasted with the MPI_Bcast function, while the equivalent in OOMPI is OOMPI_Port::Bcast.

- **Scatter** communication involves a single sender and multiple receivers. Data is divided and distributed amongst a number of tasks. The total number of transmitted messages in a single communication is equal to the number of receivers. In MPI, a message is scattered with the MPI_Scatter function, while the equivalent in OOMPI is OOMPI_Port::Scatter.

- **Gather** communication involves multiple senders and a single receiver. Partial data is gathered from multiple tasks and reassembled into a coherent data set. The total number of transmitted messages in a single communication is equal to the number of senders. In MPI, a message is gathered with the MPI_Gather function, while the equivalent in OOMPI is OOMPI_Port::Gather.

- **All-to-all broadcast** involves that every tasks communicates a message block to every other task in the system. The total number of transmitted messages in a single communication is n, where n is the number of tasks. In MPI, an all-to-all broadcast message is sent with the MPI_AllGather function, while the equivalent in OOMPI is OOMPI_Intra_Comm::AllGather.
Figure 2.14: This figure illustrates the difference between the collective communication patterns. Note that the number of transmitted network messages is much higher for all-to-all personalized exchange than for broadcast communication.
• **All-to-all-personalized exchange**, or *complete exchange*, involves that every task communicates a distinct message block to every other task in the system. The total number of transmitted messages in a single communication is $n \cdot (n - 1)$, where $n$ is the number of tasks. This is generally the most demanding communication pattern. In MPI, an all-to-all personalized exchange is initiated with the `MPI_AllToAll` function, while the equivalent in OOMPI is `OMPI_Intra_Comm::AllToAll`.

### 2.6.2.4. Synchronization

The basic synchronization mechanism in MPI and OOMPI is blocking message sends and receives, as described in section 2.4.2.2. Unlike TCP/IP, message boundaries are preserved. This means that if you execute five `MPI_Send` calls, you need to call `MPI_Recv` five times to receive all the data.

Note that for blocking sends, the functions and methods mentioned above only block until the information has been transmitted from the source message buffer, and the buffer is available for reuse, just as with data transmission with TCP/IP. It is not guaranteed that the receiving process has completed, or even started, reception of the data.

MPI and OOMPI also support one other primitive for synchronization: the *barrier*. Sometimes, the execution of a distributed application is split into several phases, and one needs to ensure that all processes in the application have finished the previous phase before any processes begin the next one. One can achieve this by placing a *barrier* on the transition between one phase and the next. In MPI, this is achieved using the `MPI_Barrier` function call. When one process calls `MPI_Barrier`, it will not continue execution before all other processes in the application have also called `MPI_Barrier`. It is also possible to use barrier synchronization with only a subset of the application’s processes.

In OOMPI, the equivalent to `MPI_Barrier` is `OMPI_Intra_Comm::Barrier`.

### 2.6.3. CORBA

CORBA[1] (an acronym for Common Object Request Broker Architecture) is an open, vendor-independent architecture and infrastructure specification that applications can use to work together over networks. The entity behind the CORBA specification is OMG, the Object Management Group, which includes companies like 3Com Corporation, Hewlett-Packard, and Sun Microsystems. The OMG does not implement the specification itself, but many implementations exist. Three examples are omniORB, MICO, and VisiBroker.

CORBA is based on the remote method call paradigm described in section 2.3.5. One of the primary strengths of CORBA is interoperability. The interfaces between CORBA programs are defined in OMG IDL (Interface Definition Language), which is neutral to the implementation language used. For each programming language supported by
CORBA, a mapping from IDL to the language exists that transforms IDL concepts and notations to that of the programming language. Currently, language mappings exist for C, C++, Java, Ada, Fortran, COBOL, Python, Smalltalk, XML, and several other languages. OMG IDL and the language mappings are part of the CORBA specification. The specification also includes a standard protocol, IIOP (Internet Inter-ORB Protocol), that is used to transfer data from one CORBA program to another.

Then end result is that one CORBA program can interoperate with another CORBA program regardless of hardware platform, operating system, programming language, or CORBA implementation. The only requirement is that the programs know each other’s IDL-defined interfaces.

CORBA is useful in many situations. Because of the easy way that CORBA integrates machines from so many vendors, with sizes ranging from mainframes through minis and desktops to hand-holds and embedded systems, it is the middleware of choice for large (and even not-so-large) enterprises. One of its most important, as well most frequent, uses is in servers that must handle large number of clients, at high hit rates, with high reliability. Specializations for scalability and fault-tolerance exist for high-end systems which need to have high reliability, and for real-time systems and small embedded systems.

The entire CORBA architecture is based upon the Object Request Broker (ORB). The ORB can be perceived as a central object bus, which all other CORBA objects interact with whether they are local or remote. The ORB is responsible for finding a CORBA object’s implementation, preparing it to receive requests, communicate requests to it and act as the messenger between the object and client.

2.6.3.1. Process Management

The basic CORBA specification does not include any provisions for starting processes on remote nodes. This is because CORBA is primarily intended for a client/server environment, where the number of clients (and thus the total number of nodes in the distributed system) is unknown. A scheme with a central database of nodes, like in MPI, would not work.

To start processes in a CORBA-based distributed system one can use one of the methods listed in section 2.6.1.1 (TCP/IP Process Management).

An extension to CORBA exists that provides fault tolerance to CORBA applications (described in [1], chapter 25). When implemented in an application, this extension can replicate the various CORBA objects in the application to several nodes in order to provide redundancy, and monitor the execution of these CORBA objects.
2.6.3.2. Initialization and Cleanup

Initialization of a typical CORBA program can be complex, especially for CORBA servers.

For servers (that offers implementations of IDL-defined interfaces), initialization consists of the following sequence of steps:

1. Initialize CORBA ORB.
2. Get a reference to the object adapter.
3. Get a reference to the naming service.
4. Activate objects with the object adapter.
5. Register objects with the naming service.
6. Activate the object adapter, and running the ORB.

For clients (that uses the server’s interface implementations), initialization is a little simpler.

1. Initialize CORBA ORB.
2. Get a reference to the naming service.
3. Look up object references with the naming service.
4. Use the object references.

For both clients and servers, the ORB that was initialized in step 1 must be destroyed before the process exits.

**CORBA ORB** Before any CORBA functionality can be used, the CORBA ORB must be initialized. This is done, with a call to CORBA::ORB_init, like in the following example:

```cpp
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");
```

This initializes the ORB (passing it the arguments from the command line in argc and argv). The last string is vendor-specific, and must match the CORBA implementation that is used. A reference to the newly-initialized ORB is stored in the variable orb.

---

16 All code in the examples in this section are in C++, although CORBA supports multiple programming languages. The sequence of steps required are the same, but the actual code will be different for other programming languages.
Object Adapter  For servers (processes that will provide implementations of CORBA interfaces), an object adapter must then be created. An object adapter is an interface that object implementations can use for communicating with the ORB. It takes care of receiving incoming requests and routing them to the appropriate object implementation. This step can be skipped for clients, which have no object implementations.

The object adapter is created with the following method calls:

```cpp
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::narrow(obj);
```

This creates a reference to the object adapter (which is named “RootPOA”) via the ORB’s resolve_initial_references method. The resolve_initial_references method is a way to get object references that are “built into” the ORB. The returned reference is to a generic CORBA Object, and needs to be cast to the appropriate type using the _narrow method. Each CORBA interface has a _narrow method that converts generic object references to the interface’s type. The reference is then stored in the poa variable.

Naming Service  Processes that will call remote objects’ methods need to obtain references to the objects. This can be done in one of two ways. The first way is to use a stringified object reference. A server can convert a reference to one of its object implementations to a long string of characters. This string can then be passed to client processes via a command-line argument or some other way, and converted to a normal object reference there.

The other way is to use the CORBA Naming Service. The Naming Service is a database or a directory which stores mappings between human-readable names and object references. The names are organized hierarchically, just like the files on a file system. A complete name consists of a sequence of (id, kind) tuples, where id is a descriptive name for the object, and kind is a descriptive name for the object’s type, class or class. When representing names, a forward slash ( / ) is commonly used to separate elements in the sequence, and a dot ( . ) is commonly used to separate id and kind. The name “/MyObject.Obj” therefore represents the id “MyObject” with kind “Obj” located at the root ( “/” ) of the hierarchy.

Both servers and clients need to use the naming service. Servers need to store mappings between names and object references in the naming service (this is called a bind operation), and clients need to be able to convert a name to an object reference (this is called a resolve operation).

A reference to the naming service, or rather, to the root context, or root directory, of the naming service, is created in the following way:

```cpp
CORBA::Object_var obj = orb->resolve_initial_references("NameService");
```
CosNaming::NamingContext rootNaming = CosNaming::NamingContext::_narrow(obj);

Activating Objects Before an object implementation is made available to clients, it needs to be activated, or registered, with the object adapter. An example of object activation follows:

PortableServer::ObjectId_var objid = poa->activate_object(myObject);

Here myObject is a pointer to the object implementation that is to be activated, and poa is the object adapter reference that was created earlier. The object ID returned is a unique identifier that later can be used to refer to this activation of the object.

Registering Objects With The Naming Service As discussed earlier, a server must register its objects with the naming service so that clients can obtain references to the objects. The following example binds a reference to myObject to the name “/MyObject.Obj”:

CosNaming::Name objname;
objname.length(1);
objname[0].id = “MyObject’’;
objname[0].kind = “Obj’’;
CORBA::Object_var obj = myObject->_this();
rootNaming->bind(objname, myObject);
myObject->remove_ref();

First the name itself is created. Then, the object implementation is converted to a generic CORBA Object reference. This reference is then bound to the name in the root context of the naming service. Lastly, the generic Object reference is freed.

Looking up Object in The Naming Service A client can look up the reference bound earlier with the following piece of code (assuming that myObject implements the MyInterface interface):

CosNaming::Name objname;
objname.length(1);
objname[0].id = “MyObject’’;
objname[0].kind = “Obj’’;
CORBA::Object_var obj = rootNaming->resolve(objname);
MyInterface_var myObjectRef = MyInterface::_narrow(obj);
Activating The Object Adapter, and Running the ORB  Before the object adapter can start receiving method invocations from clients, it must be *activated*, like this:

```c
PortableServer::POAManager_var pman = poa->the_POAManager();
pman->activate();
```

The last step in initializing a server is *running* the ORB. Running the ORB involves giving up control of the program to the ORB so that it can start servicing incoming request. The orb is run with a call to its `run` method, like this:

```c
orb->run();
```

The `run` method does not return until the ORB is shut down (e.g. with a call to `orb->shutdown()` from one of the object implementations). If a server wished to do additional work besides servicing incoming operation requests, it must create additional threads of control before calling the `run` method.

While it is running, the ORB will direct incoming request to the object implementations that were activated earlier (via the object adapter).

Destroying the ORB  The ORB is destroyed by calling:

```c
orb->finalize();
```

This must be done before the process exits.

2.6.3.3. Communication

While initialization of a CORBA program is complex, communication is very simple. All communication between processes in CORBA is via the remote method call, as described in section 2.3.5. The following is a small example of a remote method call:

```c
myObjectRef->hello("World", 3);
```

This line invokes the `hello` operation on the `myObjectRef` object reference. The string argument “World” and the integer argument “3” is passed to the operation. The operation returns nothing (i.e. has *void* return type).

Assuming that the `hello` operation is defined in the `MyInterface` IDL interface, and that the `MyInterfaceImpl` class implements the `MyInterface` interface, the following could be a suitable implementation of the `hello` operation on the server side.
void MyInterfaceImpl::hello(CORBA::String what, CORBA::Long times)
{
    for(CORBA::Long i=0; i<times; i++)
    {
        cout << "Hello, " << what << "!" << endl;
    }
}

As can be seen, CORBA takes care of marshaling/unmarshaling arguments and transmitting the request. Arguments and return values to operations in CORBA can be almost anything: primitive types like integers and floating-point numbers, strings or sequences of other data types, arrays, or structures. The only requirement is that the type is either a primitive type, or defined in an IDL file. This means that one cannot transfer C or C++ structures or types without creating a matching IDL file for them. Pointers are also not allowed.

For languages, such as C++, that don’t have built-in garbage collection, there are a series of rules that describe who is responsible for allocating and freeing objects; either the developer, or the CORBA implementation itself. We will not cover those rules here.

2.6.3.4. Synchronization

The only synchronization mechanism that CORBA makes available is that of the remote method call. When a caller calls a remote method, the call does not return until the remote object has serviced the operation and any output arguments and return values are available to the caller.

Remote method calls in CORBA has at-most-once semantics. This means that an operation is carried out at most once as a response to an operation call. Usually, the operation is carried out exactly once, but if an error occurs, it is possible that the operation is not serviced at all, of that only part of it is carried out. CORBA guarantees that an operation will not be serviced twice from one operation call as a result of an error.

It is possible to specify that the caller should not wait until the operation has been serviced, in effect sending an asynchronous message. This is done by specifying the oneway attribute in the operation signature in the IDL file. With one-way operations, the operation call returns as soon as the operation request and arguments have been transmitted. One-way operations cannot have output arguments or return values.

2.6.4. Usability Comparison

Process Management The sockets library has no facilities for process management.

MPI (and consequently OOMPI) has a utility, mpirun, that starts processes on desired nodes in a cluster. The mpirun utility uses a central database of nodes in a cluster to
determine where to start the processes. This works great when the possible number of processes in a distributed application is pre-determined, but it is difficult to use when it is necessary to dynamically add or remove nodes to the system while the application is running.

CORBA also has no facilities for process management, although the specification does offer a fault-tolerance extension.

**Initialization and Cleanup** The sockets library has a fairly simple initialization phase — to establish a connection requires a sequence of simple steps both on client and server. However, if a process is to communicate with many other processes at the same time, it might be difficult to maintain all the open connections that are needed. The cleanup phase is fairly simple, too. All that is needed is to `close()` the connection on both sides.

MPI/OOMPI has a very simple initialization phase — all that is needed is to call `MPI_Init()` or `OMPI_COMM_WORLD.Init()`, respectively. The cleanup phase is also simple, only a call to `MPI_Finalize()` / `OMPI_COMM_WORLD.Finalize()` is needed.

CORBA has a very long-winded and complex initialization procedure, both for clients and servers. While much of this is boilerplate code, that can be repeated for each CORBA program, it still requires new users creating their first CORBA program to grasp a fairly large set of concepts.

**Communication** The sockets library offers a very low-level view of data transmission. The only communication facility that the library supports is transmission of a stream of contiguous, untyped data. It is the developer’s responsibility to split the data stream into individual messages, if needed (e.g. by inserting message delimiters into the stream or including a fixed-size header with message length information). Packing individual data fields into a message before sending, and unpacking them on the receiving side, is also the developer’s responsibility, as is any conversion of the data that is needed for transmission (e.g. from host byte order to network byte order).

MPI and OOMPI offers a wide range of communication primitives, both collective and peer-to-peer, in almost any conceivable variation (blocking, non-blocking, with message buffer copying, without message buffer copying, etc.). This means that if one needs to send a message a particular way, MPI and OOMPI probably has facilities for doing it. But it also means that it can be hard for new users to select among the wide array of possible functions and method calls. Also, the basic model used is input/output — the developer must assemble and disassemble his or her messages explicitly (although MPI and OOMPI has functionality that will help the developer to do so). For some applications, this is an appropriate model, but for many others, the remote method call model used by e.g. CORBA, or the rendezvous model used by e.g. Ada or Concurrent C++, will be better.

Communication in CORBA is very simple. CORBA uses the remote method call model,
and all that is required to send a message is to call an object reference’s method. One complication in communication, however, is that sometimes it is necessary to transfer native (e.g. C/C++) data types. This may require copying of the native data to a CORBA-defined IDL data type (this decreases the performance of the program), or casting the data type to a typeless chunk of data (thereby endangering the platform- and language-independent layer that is one of CORBA’s main strengths). Another complication in programming languages without garbage collection is that the rules for who owns (and is responsible for allocating and deallocating) a particular object must be learned, but this is more a restriction of the programming language itself than it is of CORBA.

**Synchronization** The basic synchronization primitive in all three libraries is synchronization on message passing, as described in section 2.4.2.2.

In the sockets library, the only synchronization mechanism available is blocking when reading from or monitoring a socket (with `recv()`, `accept()` or `select()` calls). It is up to the developer to build more sophisticated synchronization mechanisms on top of this.

MPI and OOMPI offers blocking versions of their functions for sending and receiving messages, and in addition, offers barrier synchronization (see section 2.4.2.3 for more information on barrier synchronization).

In CORBA, the caller of a remote method blocks until it has received any return values and output parameters (note that the caller still blocks until it receives notification from the server that it has completed the operation even if the operation has no return values or output parameters). The exception is one-way operations, where the caller continues execution as soon as the operation call and arguments has been transmitted.

**Summary** Table 2.4 rates each library with respect to usability in seven different categories. For each category, the following scale is used: poor — average — good.

Sockets and CORBA have poor process management, because they do not really support process management at all. MPI and OOMPI have average process management. The `mpirun` utility works great in a environment where the number of nodes is fixed, but is

<table>
<thead>
<tr>
<th>Process management</th>
<th>Sockets</th>
<th>MPI</th>
<th>OOMPI</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization and cleanup</td>
<td>Poor</td>
<td>Average</td>
<td>Average</td>
<td>Poor</td>
</tr>
<tr>
<td>Communication Primitives</td>
<td>Poor</td>
<td>Average</td>
<td>Average</td>
<td>Poor</td>
</tr>
<tr>
<td>Synchronization Primitives</td>
<td>Poor</td>
<td>Average</td>
<td>Average</td>
<td>Average</td>
</tr>
<tr>
<td>Appropriateness in a cluster env.</td>
<td>Poor</td>
<td>Good</td>
<td>Good</td>
<td>Average</td>
</tr>
<tr>
<td>Appropriateness in other envs.</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Good</td>
</tr>
<tr>
<td>Usability in object-oriented languages</td>
<td>Poor</td>
<td>Poor</td>
<td>Average</td>
<td>Good</td>
</tr>
</tbody>
</table>

Table 2.4: Usability ratings of TCP/IP sockets, MPI, OOMPI, and CORBA
unsuitable if the number of nodes varies dynamically.

MPI and OOPMI get a good rating for initialization and cleanup, because there is only one call required for each. The socket library gets an average rating since initialization is fairly simple and cleanup is very simple. CORBA gets a poor rating because of the extremely long and tedious initialization sequence.

The support for communication primitives in the sockets library is poor, because there is no support for anything besides transmission of an contiguous, untyped stream of data. MPI and OOMPI have average support for communication primitives. The range of different communication primitives is very large, giving a lot of flexibility, but also making the libraries harder to use. CORBA also gets an average rating. This is because while communication is easy (there being only one way of communication, the remote method call), it can be too inflexible in many cases.

For the support for synchronization primitives, the situation is similar. The sockets library offers only one synchronization mechanism (blocking read/select/accept calls). MPI and OOMPI offers a wide range of different primitives, but most are based on blocking message reads. The primary exception here is barrier synchronization. CORBA only has blocking on remote method calls. While this is adequate in most situations, it can be too inflexible for some cases.

MPI and OOMPI are the most appropriate choices in a cluster environment, where the number of nodes are fixed, but are almost unsuitable in other environments, because they depend on there being a central database of nodes. Using TCP/IP sockets directly is a poor choice in both cluster and non-cluster environments and should only be used if one needs the extra performance and flexibility that raw TCP/IP sockets programming offers. CORBA is an average choice in cluster environments (on the upside, it is simple to use once it is initialized, on the downside it does not have facilities for process management), and a good choice in other environments.

Both TCP/IP Sockets and MPI are ill-fitted to object-oriented programming languages, because they are procedure-oriented libraries. OOMPI is a little better, because it wraps MPI functionality in an object-oriented fashion and can take advantage of the somewhat terser and clearer syntax that C++ can offer over C. CORBA has the best usability in object-oriented languages, because it is built on top of object-oriented concepts and because passing message to remote objects are done in the same way as “passing messages” to local objects — with the method call.

2.6.5. Performance Comparison

Figure 2.15, adapted from [2], shows the average execution time for three distributed solutions to a simple computational problem — calculating the surface area of a three-dimensional mesh. The solutions were implemented using a CORBA implementation (omniORB), an MPI implementation (LAM) and the OOMPI library, respectively. The three solutions were run on a cluster in a three-, four-, five-, and six-processor configu-
Figure 2.15: CORBA, MPI, OOMPI execution time comparison for a simple computational problem

As can be seen, when the number of processors increases, the CORBA solution falls behind in terms of performance. MPI proved to have the best performance, closely followed by OOMPI.

The performance of a version based directly on TCP/IP were not tested, but we suspect that it would be faster than all of the three libraries above based on the following three observations:

- CORBA, MPI and OOMPI uses TCP/IP as the underlying transport mechanism.
- They need to be general enough to be able to transfer any kind of data.
- However, when one bases the implementation directly on TCP/IP, one can adapt data transfer to the specific problem at hand. This is more time consuming, but will probably result in better performance.

[26] suggests that the relatively lower performance of CORBA is due to presentation layer overhead (converting data to and from the host’s native format and the format used for network transmission), and unnecessary data copying, along with overhead when demultiplexing operation requests and managing memory.

It should also be noted that in the CORBA version of the area-computation problem, any variable-length arrays that were to be transferred needed to be copied from native C++ arrays to CORBA IDL sequences. This is because it is not possible to define variable-length arrays in the IDL.
3. Context

Contents

3.1. Motivation .................................................. 52
   3.1.1. Rendezvous in Languages .......................... 53
   3.1.2. Communication Libraries ......................... 53
   3.1.3. The Rendezvous Library ......................... 54
3.2. Research Agenda ........................................... 55
   3.2.1. Definition of terms .............................. 55
   3.2.2. Hypothesis ......................................... 56
   3.2.3. Test Description ................................ 56
   3.2.4. Usability Metrics .............................. 57
   3.2.5. Performance Metrics ........................... 58

We have now covered different languages and libraries used for concurrent programming. In this chapter, we provide the motivation for a new library, based on the rendezvous communication mechanism, and state our goals and research agenda for this library.

3.1. Motivation

In the preceding chapter, we covered the most common mechanisms for communication and synchronization. Of these, the remote method call and the rendezvous seems to be the most appropriate for object-oriented languages, because they are, or can be, built on top of object-oriented concepts (such as remote objects and interfaces). Both the remote method call and the rendezvous are initiated in the same way – with a method call to a remote object reference. The difference is in how these calls are serviced. While the remote method call mechanism implies a client-server architecture — where the callee waits for invocations from callers, rendezvous are serviced by program code that are in-line with other program code the callee might execute. This means that it is possible for the callee to do other work besides servicing method invocations from callers, something that is difficult with remote method calls. (On the other hand, it is easy to use a client/server architecture with rendezvous, all that is required is that the main program of the server executes a loop with only an accept statement, accepting all invocations from clients).
3.1.1. Rendezvous in Languages

We have briefly looked at four programming languages that implement the rendezvous mechanism; these are Ada, Concurrent C, SR, and JR. These are either languages with rendezvous built-in from the start, or extensions to existing sequential programming languages. Having the language implement a concurrent programming mechanism means that the mechanism can be integrated naturally with the rest of the language, with a syntax that blends in well. However, having the mechanism as a language feature also means that it is tied to that particular language. This can be an important drawback, because the general programming features of sequential languages, such as C++ or Java is usually much better than those of a special-purpose concurrent language; as is the availability of libraries, compilers, and debugging tools.

3.1.2. Communication Libraries

Another possible approach for implementing a concurrent programming mechanism is to implement it in a library, as a set of procedures, functions, or classes that can be used by an existing sequential language. This approach has been used successfully for message-passing libraries, such as MPI implementations, OOMPI, and PVM, and for remote method call libraries implementing e.g. CORBA, among others. However, we have not been able to find a pure library implementation of the rendezvous mechanism. This means that is is, to our knowledge, not possible to use the rendezvous mechanism in existing sequential programming languages.

While the rendezvous mechanism is not readily available in a library, there are libraries available that makes other mechanisms available for communication and synchronization. In section 2.6, we covered using the low-level sockets library for TCP/IP communication, using MPI and OOMPI for message passing, and using CORBA for remote method calls. We’ve conducted a brief comparison to these with respect to usability and performance. We concluded that

- Programming TCP/IP-based applications using the sockets library will yield the best performance if programmed as efficiently as possible, because one has full control of communication down to the transport level, but it is low-level and cumbersome. The added complexity increases development time and can increase the number of faults (or “bugs”) in an application.

- MPI and OOMPI has good performance and are well suited for use in a cluster environment, where the number of nodes are fixed while the application is running. They are, however, not suitable for a traditional client/server environment, where the number of clients are unknown. In addition, they are based on the input/output model of explicit message passing, and this may not the best model for object-oriented programs.
• CORBA is based on the remote method call mechanism. It is based on object-oriented concepts, and blends in well in object-oriented languages. Communication in CORBA (e.g. interacting with remote objects) is done in the same as interacting with local objects — with the method call. However, the performance of CORBA is lower than that of MPI and OOMPI for most applications.

3.1.3. The Rendezvous Library

With all this in mind, it seems that it can be useful to create a library that is implements the rendezvous mechanism in an object-oriented language, and that strikes a sensible middle ground between performance and ease of use. We will call this library the Rendezvous Library, or RL for short. It needs to be easy enough to use so that it is an attractive alternative to MPI, OOMPI, CORBA, and other libraries, while having a performance that is not considerably worse than any of the alternatives. It needs to be usable from a widely available object-oriented language.

In designing RL, we use the following design goals:

• RL should be as simple as possible. It is not our goal to add as many features as possible in order to anticipate the widest number of uses possible.

• It should be easier to use than CORBA and should map to the target language more naturally than CORBA. This means that we will stand the risk of loosing some of the portability between languages that is CORBA’s main strength, but it also means that we do not have to design to the lowest common denominator of languages.

• The performance should be better than that of CORBA for most cases, and not significantly worse than MPI or OOMPI. We plan to achieve this partly because the library is much simpler internally than CORBA, and partly because the communication protocol and language mappings will be designed with performance in mind.

• RL should have overlapping computation and communication to improve performance.

We have made the following implementation decisions regarding RL:

• We will implement the library in C++. Initially, the library will also only be usable from C++. Additional language bindings may be implemented in the future.

• We will use an interface definition language to define the interfaces between processes, and we will create a compiler that outputs stub and skeleton code from an interface specification. This enables us to use method call syntax and semantics for communication from C++. The interface definition language will be a subset of OMG IDL, the interface definition language used by CORBA.
• The initial implementation will be done on the GNU/Linux[28] operating system, but should be readily portable to UNIX operating systems and derivatives, and possibly other operating systems as well.

• The communications subsystem will use TCP/IP directly for data transport, in order to achieve acceptable performance. We will be using the sockets library for TCP/IP programming.

• In each process, we will use several threads in order to achieve overlapping communication and computation. This will increase performance in many cases. Threads will be implemented using the POSIX threads library.

• We will create a directory service that maps human-readable names to remote object references.

3.2. Research Agenda

In order to test the usability and performance of the Rendezvous Library, we will compare it to the existing communication libraries OOMPI and CORBA. We will do the test within the context of a small numerical problem.

In an earlier project [2] we compared the performance and usability of MPI, OOMPI and CORBA within the context of a small numerical problem, and we plan to use a similar testing procedure to compare these three libraries with RL.

3.2.1. Definition of terms

For library, such as RL, that is meant to be used by programmers, we define its usability as the productivity\(^1\) and satisfaction with which the programmer can use the library in an application to achieve his or her goals. The metrics by which we will determine usability are listed in section 3.2.4.

The performance of a given communication library is the efficiency with which the library performs the tasks requested of it by the application. Relevant performance metrics are listed in section 3.2.5.

Note that library usability reflects on the efficiency with which the programmer can do his or her work (i.e. a highly usable library can be used to develop applications quicker, more reliably, or with more functionality), and does not say anything about the runtime efficiency of the library itself. Performance metrics, on the other hand, do not

\(^1\)In the context of software usability, two distinct terms are used to describe the user's productivity: Effectiveness and efficiency (see e.g. ISO 9241). Effectiveness refers to the degree to which the user is able to get his or her work done using the software, while efficiency refers to the amount of resources (time, effort) that must be expended.
say anything about how easy it is to develop applications, but do give an indication on run-time efficiency.

3.2.2. Hypothesis

We believe that the RL library will have a performance that is significantly better than CORBA, and only slightly worse than OOMPI. We also believe that the RL library will have better usability than both CORBA and OOMPI.

There are three reasons why we believe that RL will be faster than CORBA. Firstly, CORBA is a very complex specification, and CORBA implementors have had to concentrate on implementing the complete standard rather than optimizing for speed. RL is much simpler and should be easier to optimize. Secondly, we are not constrained by the CORBA language mappings, which are equally complex and in many respects sub-optimal with respect to performance (and with respect to usability as well). Thirdly, RL is designed with performance as an important goal and the design has been chosen so that the performance is as good as possible. For example, the amount of data copying is kept to a minimum, and data are not transformed for transmission over the network if it is not necessary.

The reason why we believe that RL library is more usable than CORBA is that it is much simpler and requires less initialization, and less of its functionality must be learned before the library can be used. In designing RL, we have taken a pragmatic approach and created something that works well for the majority of cases, but may be unsuitable in some, instead of creating something that is general enough to be used in any context.

The reason why we believe that the RL library is more usable than OOMPI is that we think that the rendezvous model is fundamentally better suited to modern object-oriented applications than the the input/output model used by the message-passing libraries. Communication in RL works like normal method calls, and the programmer can to a higher degree create the application as if the different parts of the program run on the same computer. In OOMPI, on the other hand, the programmer must explicitly create, send and receive messages.

3.2.3. Test Description

Our primary work in this project has been to design and develop the Rendezvous Library. But developing a library is not enough, in order to have our hypothesis about RL confirmed we need to test it against other communication libraries. A large-scale scientific test is a project on its own, so we have to limit ourself to a simple test that it takes little time to create and run. The test results will of course not give conclusive evidence regarding performance or usability, but they will at least give us a good indication.

We have chosen to duplicate the test done in our previous project[2] since it is the quickest approach for us. In this test, we calculate the surface area of a three-dimensional object
defined by a three-dimensional mesh structure. By dividing up the calculation work among several processors the problem will be solved faster. How fast will be determined by the communication library that transfers the data. Implementation of this problem exists already for MPI, OOMPI and CORBA, and we will make a version of the problem that uses RL as its communication library. The program structure should be as identical as possible and of course use exactly the same calculation algorithm. In addition to these four versions a serial version exists, designed to run at 1 processor only. This version is needed to calculate how efficient the parallel version are compared to the serial version.

In order to make the test as accurate as possible we need to test it on a large number of processors. These were provided by using a cluster at NTNU, called ClustIS, which is a cluster of 51 processors in total[27]. We will not use all of these, only a selected number which are as similar to each other as possible. Test runs were also made to see how many processors each version could handle. This resulted in a choice of using a maximum of 25 processors, and a minimum of 3 processors. No dual-processor nodes were included. After experimenting with test runs we determined that our average execution time will be calculated from 5 runs. Making several runs reduces the probability that random variations in execution time due to factors outside our control affects our results.

In addition to measuring the execution time we are interested in the speed-up and efficiency of each program version as well. These are explained more thoroughly in 3.2.5. The area calculation implementation is described in 6.2.1.

### 3.2.4. Usability Metrics

Library usability metrics are measured from the point of view of the user of the library. In this case the user is a developer of an application (such as the OODFEM framework used in our tests). These are the usability metrics that will be used.

#### 3.2.4.1. Lines of code

The *lines of code* metric measures the number of code lines in an application, or the number of code lines needed to perform a given task (e.g. library initialization). This gives an indication on how much work it is to use the library to perform a given task, but does not say anything about how easy it is to use the library, and does not take the experience and knowledge of the programmer into consideration. The lines of code metric is an absolute metric that can be easily measured, and it can be used as a rough guide to how complex it is to develop an application using a given library.

For counting lines of code, we have used the SLOCcount program[38], which can count physical lines of code for a large number of different programming languages.
3.2.4.2. Numer of classes

The number of classes is a measurement of both the complexity and order of an implementation. Having to create a large number of classes in order to solve a problem is time-consuming work. Of course one need to strike a balance between having a few, large classes and having many, small classes. Few classes and many lines of codes per class makes it hard to navigate and make changes, while creating many classes compared to the number of code lines might mean much unneeded work. Number of classes is an absolute measurement.

3.2.4.3. Learnability

The learnability of a library gives an indication of how easy it is to learn the library, and how much time one must spend (on average) in order to learn what’s required to make an application using the library. Learnability is a subjective metric.

3.2.4.4. Flexibility and functionality

The flexibility of a library says something about of how easy it is to use the library in different settings, to do different things, using different programming styles and application patterns. A flexible library accommodates its users’ needs by not placing restrictions on the styles or patterns used, while an inflexible library forces the user to use one particular kind of programming style; or makes it hard to use application patterns other than that for which the library was designed (for example, if a communication library is designed for the client/server interaction pattern, it can be difficult, if not impossible, to use it in a peer-to-peer application). Like learnability, flexibility is a subjective metric.

The functionality of a library is the set of operations or tasks that can be performed by the library. A library with much functionality can perform a wider range of tasks than one with little functionality (for example, a library that supports broadcast communication has more functionality than one which doesn’t, assuming that the libraries are otherwise identical).

3.2.5. Performance Metrics

These are the metrics that will be used in the performance evaluation. All performance metrics are absolute, objective metrics.

3.2.5.1. Total Execution time

The total execution time $T(n)$ for a parallel program on $n$ processors, is the time it takes from the first process starts its execution, to the time the last process finishes. If the
processes start simultaneously, the execution time is equal to the execution time of the process that takes the longest time. Execution time can be divided into code execution time $C_i$ (for the $i$'th processor) and communication time $K$. $K$ is the sum of all the communication that is done throughout the execution. The communication time will obviously increase as the number of processes increase.

$$T(n) = K + \frac{\max_{i=0}^{n-1}(C_i)}{n}$$ (3.1)

### 3.2.5.2. Speed-up

It is known that it is unrealistic that the entire serial program can be divided and run on a distributed system. Only a percentage of the serial program will be affected by adding more processors to the system. This means there exist some threshold over which it is no longer cost efficient to add more processors. We can estimate the change of execution time from the serial to the parallel version of a program by using the following equation

$$R(p, n) = (1 - p) + \frac{p}{n}$$ (3.2)

Here $p$ is the percentage that can be parallelized, $n$ is the number of processors and $R$ represent the ratio of parallel execution time to the serial execution time. For example if 60% of the task is parallelizable, and we have 4 processors, we get:

$$(1 - 0.6) + \frac{0.6}{4} = 0.4 + 0.15 = 0.55$$ (3.3)

From equation 3.3 we can see that the parallel execution time is 55% of the serial execution time.

The equation is usable only for estimation and cannot be used to calculate any real results since it is based on a very simplified cost/time model. The communication time, for example, is not included. Still, in the cases where the code execution time greatly outweighs the communication time, it gives results that are close to reality. A difficult problem with the usage of this formula is to determine the percentage of the code that is parallelizable.

What we are interested in is a cost/time model that can give us an indication of when it is no longer economical to add more processors. Speed-up is the degree of performance difference between a serialized program and the same program run in a cluster with a given number of processors. The equation looks like this:

$$S(n) = \frac{t}{T(n)}$$ (3.4)

where $t$ is the serial execution time and $T(n)$ is the runtime of the parallel version. As you can see an ideal case is when $S(n) = R(1, n)$, which gives us $S(n) = n$. This means
that (near) 100% percent of the program can be parallelized and the communication time is negligible. By doubling the number of processors, the speed-up is doubled, which gives us a linear graph for \( S(n) \). For non-trivial problems, it’s usually impossible to parallelize the whole program, and the speed-up will be worse than the ideal case. However if the calculated number for speed-up is bigger than the number of processors used we get super-linear speed-up. This should be theoretical impossible for linear algorithms as this implies that the processors work faster than the processor running the serial version of the program. However it can be frequently observed when a program is running under optimal conditions. The reasons can be many, e.g. varying system or network performance.

3.2.5.3. Efficiency

Efficiency is defined as the speed-up divided by the number of processors:

\[
E(n) = \frac{S(n)}{n}
\]

This value gives an indication of how much the processors are kept occupied, or in other words, how good the speed-up result really is compared to the change in the number of processors. In most cases, one can easily see that the speed-up increases as the number of processors increases, but is it increasing enough? By plotting the efficiency in a graph for program executions with different number of processors we can find an efficiency maximum. This maximum is where the performance of the system compared to the number of processors is the highest. Beyond this maximum, total execution time will still decrease as more processors are added, but it will not decrease by as much.

The efficiency gives a good indication on when a parallel program works at its most efficient. Some parallel software is optimized for many processors, and performs poorly when run on only a few processors, while other software works great with a few processors, but performs poorly with many processors.
4. Rendezvous Library Overview

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Overall Architecture</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Using the RL library</td>
<td>64</td>
</tr>
<tr>
<td>4.2.1</td>
<td>User-supplied algorithm</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Stub procedures</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>RL Concepts And Semantics</td>
<td>66</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Process</td>
<td>66</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Thread</td>
<td>66</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Task</td>
<td>67</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Interfaces</td>
<td>68</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Reference objects</td>
<td>68</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Servant objects</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>IDL Compiler</td>
<td>69</td>
</tr>
<tr>
<td>4.4.1</td>
<td>IDL Example</td>
<td>71</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Invoking the IDL Compiler</td>
<td>72</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Using the Output From the Compiler</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Directory Service</td>
<td>78</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Interface</td>
<td>79</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Starting the Directory Service</td>
<td>81</td>
</tr>
<tr>
<td>4.6</td>
<td>Applications using RL</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparing RL to Other Approaches</td>
<td>82</td>
</tr>
</tbody>
</table>

As stated earlier, the Rendezvous Library is based on the rendezvous programming paradigm, which is similar to the remote object programming paradigm. In this chapter we will provide an overview of the overall architecture of RL, and describe briefly its major components.

4.1. Overall Architecture

Figure 4.1 shows the overall architecture for the rendezvous system.
Figure 4.1: Overall system architecture
A distributed application using RL consists of a number of tasks. Each task does some local computation (runs some pre-determined algorithm), and in addition it

- makes available services that other tasks can call upon
- uses the services that other tasks makes available.

The services that a task offers, which are called operations, can be accessed using operation calls.

Each task consists of three parts:

- A user-supplied algorithm. This is the task’s main, the algorithm that makes the task do useful work.
- A set of stubs that makes calling other tasks’ entries seem like normal method calls. A program called an IDL compiler generates these stubs.
- A set of skeletons that makes it possible to service other task’s operation calls with a normal method call. In other words, the developer need not be aware that the operation calls can come from a remote computer. The IDL compiler generates these skeletons.
- A run-time library that implements the task’s communication thread and related support services. This is the library part of the RL system.

The operations that each task makes available to others are listed in an interface specification. The interface specification is written in an abstract declarative language called the IDL (interface description language). Each interface consists of a set of operations that can be implemented by a task.

The IDL compiler transforms a set of interface specifications into a corresponding set of C++ stub and skeleton files. Stubs are used on the client (calling) side to make calling entries seem like normal method calls, and skeletons are used on the server (called) side as a basis for implementing the entries. The IDL compiler is described in section 4.4.

Tasks communicate with each other by using a communication protocol implemented by the run-time system. This works transparently to the user of the RL system.

To assist in finding what tasks are available on a distributed system, and where they are located (i.e. IP addresses, port numbers), a directory service can be used. The directory service maps tasks to human-readable names. Tasks register themselves with a name in the directory on start-up, and other tasks can query the directory for tasks that match the given name.

63
4.2. Using the RL library

From a users point of view it is necessary to think of everything in his problem domain as tasks. Each task will perform a specific piece of the problem. When a task is created, the user defines what this specific task will do in the task’s Run() method.

4.2.1. User-supplied algorithm

The algorithm will be implemented in a task’s Run() method. The user must create a class that subclasses the Task class, and then implement the Run() method as needed to solve his problem. An excerpt from a tiny example is shown below. First from the header file, and then from the source file.

```cpp
class TestTask : public RL::Task {
    public: TestTask();
    ~TestTask();
    void Run();
};

void TestTask::Run() {
    //Insert users algorithm here
}
```

4.2.2. Stub procedures

The interface specifications are transformed into skeleton and stub files. These files contain class declarations for skeleton classes and stub classes. Stubs lie on the client (calling) side, while skeletons lie on the server (called) side. It is possible for a task to operate both as a client and as a server. The task will then have both stubs and skeletons. Normally however, it is one or the other. The stub and skeleton classes are instantiated to reference objects and to servant objects, respectively. The superclass for a given Reference Object is the ObjectRef class, and the superclass for a given Servant Object is the ObjectServant class.

How to create the IDL file is briefly explained in section 4.4.

Run-time library The run-time library contains several classes, but from a user point of view only a selected few will be used.

The classes that the user will need to use are:

- Thread
• Task
• TCPLocation

The above classes are described briefly in the following text.
The classes that the user does not need to use are:

• CommThread
• ListenThread
• MainThread
• Location
• CommQueue
• CommItem
• TCPTransport

These are described in section 5.1.

Thread If the user wishes to create his own threads, he will have to subclass the Thread class. The new thread, which we’ll call TestThread, will only have to implement the virtual Run() method of the Thread class in order for it to work. An example is shown below. First the header file, and then the source file.

```
#include <RL/RL.h>
class TestThread : public Thread {
    public:
        TestThread();
        ~TestThread();
        void Run();
};

#include <testthread.h>
TestThread::TestThread() {
    //Empty constructor
}
TestThread::~TestThread() {
    //Empty destructor
}
```
void TestThread::Run() {
    //Do whatever the thread is supposed to do
}

Then, in the user-defined algorithm, he creates the object and starts its \textit{Run()} method. In addition, it is necessary to instruct the thread to wait until the thread has finished executing.

\begin{verbatim}
RL::Thread t;       //create object
    t.Start();      //start thread
    t.WaitFor();   //wait for thread to finish executing
\end{verbatim}

\textbf{Task} To create a new task one simply subclasses the existing task class. A code example is given above in section 4.2.1.

\textbf{TCPLocation} The user will only have to use this class in one case: If he should choose not to use the directory service. The user will then have to create a TCPLocation object to specify the address of a given task, or the address of a given object reference.

4.3. RL Concepts And Semantics

This section describes the main concepts used in the RL system, and their semantics.

4.3.1. Process

A process is a program being run. A process consists of one \textit{main} thread of execution (i.e. the thread that executes the \textit{main} function), and optionally one or more threads and/or tasks.

4.3.2. Thread

A thread of execution is a series of statements that are executed independently of and simultaneously\footnote{At least conceptually. In an actual implementation, if there are more threads than available CPU's, the operating system timeshares the available CPU's between the threads so that all threads \textit{appear} to execute simultaneously.} with other statements that are executed in a given process. A thread is an example of an \textit{active} object. An active object encapsulates data and behaviour, like a normal, \textit{passive} object, but in addition it also has \textit{control}, i.e. it executes statements independently of its creator.
4.3.2.1. Run operation

Each thread has a Run operation that contains the statements that is to be executed by this thread. The Run operation is not called directly by the user, rather, it is started by RL when the user invokes the Start operation, as described below.

4.3.2.2. Thread operations

The operations that can be performed on a thread are:

Start Starts the thread. When starting the thread, a new operating-system thread of execution is created, and the thread's Run method is executed in that thread of execution. The Start operation must be invoked by one thread only, and the WaitFor operation must be invoked before the Start operation can be invoked a second time. The Start operation returns when the thread has been successfully started; it does not wait for the thread to complete. To do that, use the WaitFor operation.

WaitFor Wait for the thread to complete. It is an error to have more than one thread waiting for a given thread to complete.

4.3.3. Task

A task represents a thread of execution\(^2\), that, in addition to executing a series of statements, can accept remote operation invocations from other tasks, and can all the operations of other tasks. It has a communications subsystem that is used to transfer messages to and from remote processes. Every task has a Run operation, and in addition supports operations for starting and monitoring the task. Tasks, like threads, are active objects.

4.3.3.1. Run operation

Each task has a Run operation that contains the statements that is to be executed by this task. The Run operation is not called directly by the user, rather, it is started by RL when the user invokes the Start operation, as described below.

4.3.3.2. Task operations

The operations that can be performed on a task is:

\(^2\)In the actual implementation, each task is represented by several threads of execution, but all but one of them are involved in communicating with other tasks, and are hidden from the user.
Start  Starts the task. When starting the task, the communication subsystem is initialized for that task, new operating-system threads of execution are created, and the task’s Run method is executed in one of these threads of execution. The rest of the threads are used for the communication subsystem. The Start operation must be invoked by one thread only, and the WaitFor operation must be invoked before the Start operation can be invoked a second time. The Start operation returns when the task has been successfully started; it does not wait for the task to complete. To do that, use the WaitFor operation.

WaitFor Wait for the task to complete. It is an error to have more than one thread waiting for a given task to complete.

4.3.3.3. Remote operation invocations

Inside a task, any number of reference objects and servant objects can be created. Reference objects are used to invoke a remote operation (i.e. an operation that is to be performed by another task in another process), while servant objects are used to accept remote operation invocations and service these.

Each reference object enables the task to call the operations of one remote interface, and each servant object enables the task to export one interface.

Both reference objects and servant objects use the task’s communication subsystem for transferring information between caller and callee. Both reference objects and servant objects are passive objects, i.e. calling or servicing a remote operation does not create a new thread of control.

4.3.4. Interfaces

An interface is the formal definition of a set of operations. For each operation, the name of the operation, and the names and types of any input and/or output arguments, and the type of the return value, are specified.

4.3.5. Reference objects

Reference objects contain the stub operations that are necessary in order to execute a remote operation. For each operation in an interface, the corresponding reference object contains one stub operation.

4.3.6. Servant objects

Servant objects (or skeletons) contain the skeleton operations that are necessary to accept an remote operation invocation and invoke the corresponding local operation. For each
operation in an interface, the corresponding servant object has two operations:

**Accept** _operation-name_ Wait for a request for the operation to arrive, and executes the corresponding operation. The _operation-name_ is defined by the user in the IDL specification. If the user has specified a method *delete*, the accept-operation will be named **Accept_delete**.

**TryAccept** _operation-name_ If a request for the operation is waiting, executes the corresponding operation. As above, the _operation-name_ is defined by the user.

In addition, servant objects support the following operations:

**Select** Select waits (either indefinitely, or a given amount of time) for an operation request to arrive.

**Accept** Accept waits for any request to arrive, and executes the corresponding operation.

**TryAccept** If any request is waiting, executes the corresponding operation.

### 4.3.6.1. Implementation object

Each servant object holds a reference to a local object that implements the interface’s operations. When calling any of the servant’s **Accept** or **TryAccept** operations, the servant invokes the corresponding operation of the implementation object.

### 4.4. IDL Compiler

The IDL (interface definition language) compiler generates *stub* and *skeleton* code from interface specifications listed in an IDL file. IDL is a purely declarative language; its syntax contains only declarations — of modules, interfaces, operations and other entities. The reason why interfaces are specified using a special language instead of using normal C++ classes, for example, is that the C++ syntax is not descriptive enough to be used for remote operation calling. For example, an argument to an operation can be an input argument (sending data to the operation), an output argument (returning data from the operation), or both; and it is not possible in C++ to specify which of the three types a particular method argument is. Furthermore, C++ allows constructs like *pointers* that only work if caller and callee is in the same address space, and that is *not* the case when calling a remote operation.

In addition to stub and skeleton code, the IDL compiler generates a C++ translation of the interface specification itself. Figure 4.2 illustrates the input and output files used by the IDL compiler.

---

In general, of course, IDL compilers can be used with any language — not just C++. For simplicity, and because that is what the RL library uses, we will concentrate on C++ here.
Figure 4.2: IDL compiler inputs and outputs
The purpose of the stub and skeleton code is to “bridge the gap” between local method calls and remote message passing:

- **stubs** are used on the client (caller) side. They marshal the operation call and any arguments into a network message, and sends the message to the server. When a reply arrives from the server, they unmarshal return values and output arguments and returns them to the calling task.

- **skeletons** are used on the server (callee) side. They unmarshal the operation call and arguments sent by the stub and invokes the appropriate local method to service the call. When the local method completes, the skeleton marshal return values and output arguments and sends them in a network message to the client.

Figure 4.3 illustrates the marshaling and unmarshaling process.

For each interface defined in the IDL file, the IDL compiler outputs the code for one **reference class**. The reference class contains the stub methods for that interface, and instances of the reference class are the **reference objects**, described in the previous section. In addition, for each interface, the IDL compiler also outputs the code for one **servant class**. The servant class contains the skeleton methods for the interface, and instances of the servant class are the **servant objects** described earlier.

The interface definition language is a subset of the OMG IDL[9, chapter 3], which is used by CORBA.

### 4.4.1. IDL Example

Figure 4.4 shows a small example of an IDL file.

This IDL file declares an **interface**, “hello”, with one **operation**, called “hi”. The “hi” operation takes one input argument — the string argument “a”.

When the IDL compiler is invoked on this IDL file (see the next section), the following files are produced:
interface hello {
  void hi(in string a);
};

Figure 4.4: Small IDL file

- **hello.h** — This file contains a C++ translation of the “hello” interface. It is a class with only pure virtual methods.

- **helloRef.h** — This file declares helloRef, the hello reference class. It is inherited from the hello class. As explained earlier, the reference class’ methods contain the functionality needed to marshal a call to one of the interface’s methods (in this case the “hi” method), and to unmarshal the reply that is sent back from the callee.

- **helloRef.cpp** — This file implements the helloRef class’ methods.

- **helloServant.h** — This file declares helloServant, the hello servant class. As previously mentioned, the servant class contains the functionality needed to accept a call to one of the interface’s methods from a remote process, unmarshal it, and invoke the corresponding local method. When the local method has finished, a reply is marshaled and sent back.

### 4.4.2. Invoking the IDL Compiler

The IDL compiler can be invoked simply as in the following example:

```
rlidl hello.idl
```

This compiles the hello.idl file and produces the files listed in the previous section. The files are output to the current directory.

### 4.4.2.1. Generating Parse Tree Graphics

The IDL compiler can also create a graphic of the parse tree produced during parsing. It does this by outputting a graph specification that can be read by the vcg [29] compiler-graph visualization tool. The vcg program can then draw the parse tree. To generate the graph specification, use the -t option to ridl, as in the following example:

```
rlidl -t hello.vcg hello.idl
```
This outputs the graph specification to the `hello.vcg` file. The resulting graph can then be visualized as follows:

```
  xvcg hello.vcg
```

### 4.4.2. Other options

The IDL compiler has some other options that can be useful. One can list all available options by giving the following command:

```
rildl --help
```

This will display a short help text including all available options.

### 4.4.3. Using the Output From the Compiler

#### 4.4.3.1. Calling Operations

If one wishes to call an interface’s operations one must do so by creating an instance of the interface’s reference class and calling the reference class’s methods, as in the following example:

```java
  helloRef h(this, "rltcp://localhost:8732/#2341");
  h.hi("World");
```

Here `this` is a pointer to the task (presumably the above code fragment would be placed in the `Run()` method of a task. The string “rltcp://localhost:8732/#2341” is an example of a *stringified object reference*, it is a string describing the location of the remote object. If one does not wish to concern oneself with these cryptic strings, one can use the *directory service* instead. The directory service maps symbolic names, such as “myHelloInstance” into stringified object references and vice versa. It is described in section 4.5.

In order to be able to use the reference class, its header file must be included in the source files of the program, and its implementation (.cpp) file must be compiled with the rest of the program.

#### 4.4.3.2. Serving Operation Calls

If one wishes to receive operation calls from remote processes, one must do so by creating an instance of the interface’s servant class, as follows:

```java
  helloImpl myHelloImpl;
  helloServant hs(this, &myHelloImpl);
```
<table>
<thead>
<tr>
<th>Calling</th>
<th>hello.h, helloRef.h</th>
<th>helloRef.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serving Calls</td>
<td>hello.h, helloServant.h</td>
<td>helloServant.cpp</td>
</tr>
<tr>
<td>Both</td>
<td>hello.h, helloRef.h, helloServant.h</td>
<td>helloRef.cpp, helloServant.cpp</td>
</tr>
</tbody>
</table>

Table 4.1.: IDL compiler output files needed.

Here this is a pointer to the task, and myHelloImpl is a pointer to an object that implements the hello interface, that is, it inherits from the interface class (“hello” in this example), and provides definitions for all its methods.

Individual operation calls can then be received by invoking one of the Accept or Try-Accept methods, as described in section 4.3. The servant object will then fetch the appropriate operation call request from the list of incoming requests, unmarshal it, and invoke the corresponding method of the implementation object (“myHelloImpl” in this example). When this method returns, a reply consisting of return values and output arguments are sent back to the caller.

```c
hs.Accept_Insert();    //waits until Insert has arrived
hs.TryAccept_Insert(); //accepts a waiting Insert
```

In order to be able to use the servant class, its header file must be included in the source files of the program, and its implementation (.cpp) file must be compiled with the rest of the program.

4.4.3.3. Both Calling and Serving Operations

It is possible to act both as a caller and a receiver of operation calls, in that case, the files for both the reference class and the servant class must be included in the program. Table 4.1 lists the files required for the different scenarios. If unsure, it is always safe to include all files generated by the IDL compiler.

4.4.3.4. IDL Types

In the present implementation, the IDL compiler supports the following type:

- Simple (or basic) types, as shown in table 4.2.
- Strings of 8-bit characters
- Single dimension, variable length arrays of simple types are supported as operation parameters. Arrays are not supported as return values, but one can use output parameters to achieve a similar effect. Multi-dimensional arrays, or arrays of strings, are not supported.
<table>
<thead>
<tr>
<th>RL IDL type</th>
<th>C++ type def.</th>
<th>C++ native type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>RL::Float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>RL::Double</td>
<td>double</td>
</tr>
<tr>
<td>long double</td>
<td>RL::LongDouble</td>
<td>long double</td>
</tr>
<tr>
<td>short</td>
<td>RL::Short</td>
<td>short int</td>
</tr>
<tr>
<td>long</td>
<td>RL::Long</td>
<td>long int</td>
</tr>
<tr>
<td>long long</td>
<td>RL::LongLong</td>
<td>long long</td>
</tr>
<tr>
<td>unsigned short</td>
<td>RL::UnsignedShort</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>unsigned long</td>
<td>RL::UnsignedLong</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>char</td>
<td>RL::Char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>boolean</td>
<td>RL::Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>octet</td>
<td>RL::Octet</td>
<td>unsigned char</td>
</tr>
</tbody>
</table>

Table 4.2: Mapping of basic types

Since the IDL compiler can parse the complete syntax specified by OMG IDL, it should be fairly easy to add support of other types in the future. One only needs to implement marshaling and unmarshaling of these new types. For more information, see section 5.2.

### 4.4.3.5. Mapping of IDL operations to C++ methods

The operations in an IDL interface is mapped to C++ methods according to the following rules:

- They have the same name as the corresponding IDL operation. It is not checked that this name does not clash with a C++ reserved word or is otherwise in conflict with an existing definition. It is the responsibility of the developer to ensure that name classes do not occur.

- The parameters are mapped in the following way:
  - Simple types are mapped according to table 4.2.
  - Strings are mapped to the `const char *` type.
  - Arrays are mapped to pointers, for example an array of long integers are mapped to `RL::Long *`. In addition, a parameter of the same name with " _size" appended, and with type `RL::Long contains the number of elements in the array.

- Input arguments are passed by value.

- Output and input/output arguments are passed by reference.
As an example, consider the following IDL:

```cpp
interface A {
    typedef short short_array[0];
    long myOp(in long p1, in short_array p2, out short_array p3, inout double p4);
};
```

The above IDL file would be mapped to the following C++ interface class:

```cpp
class A {
    public:
        typedef RL::Short *short_array;
        RL::Long myOp(RL::Long p1, 
                       RL::Short *p2, RL::Long p2_size, 
                       RL::Short *p3, RL::Long p3_size, 
                       RL::Double &p4) = 0;
};
```

### 4.4.3.6. Memory Management

C++ is a language without automatic garbage collection. This means that the programmer must explicitly delete objects in memory when they are no longer needed. This includes parameters passed as arguments to methods, and return values from methods, and it is also true for methods that are generated by the IDL compiler. To ensure consistent behavior, there has been defined a set of rules that governs who owns data objects passed to and from IDL-generated methods, and therefore are responsible for deleting them; the method itself, or the method’s caller. These rules are described below.

**Calling IDL-defined Operations** When passing arguments to operations calls, or receiving return or output arguments from them, the following rules apply:

**Simple types:** Simple types are passed by value as input parameters and return values and by reference as output or input/output arguments. In either case there is no allocation or deallocation involved.

**Strings:** As input parameters, strings are owned by their caller, and will not be modified by the IDL-generated method.

As output parameters or return values, the IDL-generated method will allocate memory for the string, and set the `const char` pointer passed as the parameter to point to this memory. The string should be deleted with `delete []` when it is no longer needed. When calling a method with an output string parameter, this parameter should be set to `NULL`. 

76
As input/output parameters, the IDL-generated method acquires ownership of the string passed in as parameter, and will delete it before allocating new memory for the output string, as described above.

**Arrays:** As input parameters, arrays are owned by their caller, and will not be modified by the IDL-generated method. This also includes the `array_size` parameter.

As output parameters, the IDL-generated method will allocate memory for the array, and set the parameter passed to point to this memory. It will also set the `array_size` parameter to the number of elements in the output array. The string should be deleted with `delete[]` when it is no longer needed. When calling a method with an output array parameter, this parameter should be set to `NULL`.

As input/output parameters, the IDL-generated method acquires ownership of the array passed in as parameter, and will delete it before allocating new memory for the output array, as described above. The `array_size` parameter is also updated to reflect the new element count of the array.

Arrays can not be return values, because there is no way to return the number of elements in the array.

4.4.3.7. Receiving Operation Calls

When receiving operation calls, the situation is opposite: the IDL-generated method calls a method implemented by the programmer. The following rules apply:

**Simple types:** As before, simple types are passed by value as input parameters and return values and by reference as output or input/output arguments. In either case there is no allocation or deallocation involved.

**Strings:** As input parameters, strings are owned by the IDL-generated method, which will delete it after the method call completes.

As output parameters or return values, the programmer must allocate memory for the string, and set the `const char` pointer passed as the parameter to point to this memory. This string is not deleted by the IDL-generated method, so the application must delete it at some later time if required. Not having the IDL-generated method delete the string can be an inconvenience, but it ensures that it is not necessary to copy the string before passing it to the IDL-generated method if it needs to be stored until later.

As input/output parameters, the programmer must allocate new memory for the string and set the `const char` pointer passed to point to this memory. It must not delete the old memory, because the IDL-generated method holds a reference to this memory and will delete it.
Arrays: As input parameters, arrays are owned by the IDL-generated method, which will delete it after the method call completes.

As output parameters, the programmer must allocate memory for the array, and set the parameter passed to point to this memory. The array is not deleted by the IDL-generated method, so the application must delete it at some later time if required. As with strings, this is to avoid imposing a requirement that the data be copied before passing it to the IDL-generated method.

As input/output parameters, the programmer must allocate new memory for the array and set the parameter passed to point to this memory. It must not delete the old memory, the IDL-generated method holds a reference to this memory and will delete it.

Arrays can not be used as return values, as previously mentioned.

4.5. Directory Service

In RL a process communicate with other processes by calling a remote operation, implemented in a remote task (i.e. a task residing on another computer). To be able to do this, one must know the location of each computer one communicates with. This information can be hard-coded in each program, but this makes it hard to move the program to another computer.

The directory service is a service running on a known location that allows processes to register the location of their object, mapping it to a human-readable name. Other processes can then find the location of objects they need to communicate with by querying the directory service. This service is comparable to the naming service of CORBA.

Only one directory service is needed on the network. Using only one directory service does create a single point of failure, because there are no backup solutions should this service fail. For systems that require high reliability, add-on solutions must be implemented by the developer. A possible solution is having multiple instances of the directory service running on several computers with data replication between the directory service instances.

Instead of implementing the directory service into the communication library directly it was implemented as a separate process, exactly like any other remote object. This was done deliberately in order to increase flexibility. As the directory service is a separate part it can be easily changed or replaced should the user want to make a more suitable version. Only minimum functionality has been implemented and it can be modified and expanded with ease. One need only to adhere to the interface specified in the directory service IDL file, to remain compatible with existing application. For more on how this version of the directory service was implemented, see section 5.3.
4.5.1. Interface

It is through the interface that developers are able to use the directory service. As previously explained, to make a set of remote operations accessible to other processes they must be defined in a IDL file. When this file is compiled the necessary layer that deals with the communication is created. The process of creating and compiling IDL-files is described in 4.4.

The Directory Service maintains a list of references to objects that are remotely accessible through the interface. The interface is used mainly for registering and searching for remote objects, but it is also possible to unbind (unregister) objects should this be desirable. Any attempt to bind an object to a name already in use will result in deleting the old reference and storing the new.

Interface Operations:

- **Bind(Object, Name)**: Binds the remote object (given as a *stringified object reference*) to the specified name. Unbind(Name) is called first, then it binds the object to the name given.

- **Unbind(Name)**: This method uses Resolve(Name) and removes the object reference if it is found.

- **Resolve(Name)**: This method returns the object reference the name refers to, or an empty string if the name was not found.

Let’s say a developer has created a IDL file defining a interface called “hello”. This interface contains a method called “sayhello(in string message)” and could look something like this:

```plaintext
interface hello{
    void sayhello(in string message);
}
```

By compiling the IDL file the classes “helloServant” and “helloRef” are created. The implementation of the interface is contained within a object the developer calls “hello_i”. The name of the object can be anything as long as it inherits from the “hello” interface class and implements the method “sayhello(string myname)” as defined in the interface.

For simplicity, the “hello_i” class also inherits from RL::Task, as described in section 4.2.1. This saves one from having to create a separate class as the task class. It also means that “hello_i” needs to contain another method called “Run()”, which contains the task’s algorithm. To allow remote objects to use the method “sayhello()” the developer has to bind the stringified object reference to a name in the directory service. The following three lines of code in “hello_i::Run()” describes how this is done.

79
hello_i::Run(){
    helloServant hs(this, hello_i);
    Directory::ServiceRef ds(this, DIRECTORY_SERVICE);
    ds.Bind(hs.ToString(), "HELLO");
    /* ... code to service operation calls here ... */
}

The first line creates the object “hs” that is the servant object for “hello_i”. This servant object will be the connection point on the server side when a remote call is made. The second object “ds” is an object reference to the directory service. “DIRECTORY_SERVICE” is a string that contains the location of the service, as a stringified object reference. By using the remote method call Bind(), described above, the path to “hello_i” in bound to the name “HELLO”. A stringified object reference to “hello_i” is returned when calling “hs.ToString()”. If the developer wants another object to a perform remote call to the “sayhello” operation he has to put the following three lines of code in that object’s Run() method (this object is called Test for the lack of a better name):

Test::Run(){
    Directory::ServiceRef ds(this, DIRECTORY_SERVICE);
    HelloRef hr(this, ds.Resolve("HELLO");
    hr.sayhello("Testing Directory Service");
}
The first line creates a reference to the directory service in the same way as in the previous example. The second line creates the HelloRef object that will contact the HelloServant object. By using Resolve() the “hello_i”’s reference can be found. The last line simply invokes the remote call by using the connection established between HelloRef and HelloServant.

From the example above one can see the only difference between making a remote call to the directory service and contacting the hello interface is in the second parameter. Both are a strings containing an object reference, but they are found in different ways. Resolve() returns the reference string by contacting the directory service. But it is obvious that you cannot contact the directory service to find the location of the directory service. This is why one need the string DIRECTORY_SERVICE that contains the object reference directly. A stringified object reference that the reference object use can look like this:

```
rltcp://localhost:1337/#1337
```

Here “localhost” is typically used instead of a IP address if the remote object is on the same machine. “1337” is the port number and “1337” is the object ID. This is the format the string that defines the path of the directory service must use.

### 4.5.2. Starting the Directory Service

The directory service must be started before trying to use it. This service is started with the following command line:

```
./rldirectory
```

As explained the directory service is a independent process that runs on a known location. In the previous examples the path to the directory service is contained within the “DIRECTORY_SERVICE” string.

### 4.6. Applications using RL

Since RL is a general communication library, it can be used in almost any type application. However, it has been designed to be a more appropriate alternative to remote object libraries for use in distributed numerical applications. Consequently, it is optimized for efficiency and ease-of-use over functionality and generality. Examples of kinds of application for which using RL is appropiate include:

- Distributed numerical applications
- Client-server software
- Component-oriented software

RL is probably not appropriate for applications requiring collective communications or using the advanced services of CORBA, for example.

4.7. Comparing RL to Other Approaches

The primary advantage gained by using the Rendezvous Library is that one uses the more natural object-oriented programming style. This style is also common in the remote object libraries. Being more natural to human cognition, it is easier to understand. Object-oriented programming also provides faster development, and (supposedly), higher quality.

We believe that the most significant advantages to using the Rendezvous Library instead of remote object libraries are the following:

- RL uses a separate thread of control for I/O operations. Thus achieving overlapped I/O and computation.
- The generated stub code is optimized for efficiency.
- The amount of data copying is kept to a minimum

In addition, remote object libraries are tied to the client-server programming style. This is not the case with RL applications. These are free to use any programming style they wish.

One disadvantage of RL compared to message-passing libraries is that RL does not support collective communication patterns like broadcast and scatter, only point-to-point communication. A disadvantage compared to remote method call libraries is that RL does not support the advanced features of some remote object libraries. On example is CORBA’s CORBA Services.
5. Implementation

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. Run-time library</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>5.1.1. Tasks</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>5.1.2. Implementation and usage of RL</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>5.1.3. Communication protocol</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>5.2. IDL Compiler</td>
<td></td>
<td>94</td>
</tr>
<tr>
<td>5.2.1. Internal Structure</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td>5.2.2. Scanner</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>5.2.3. Parser</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>5.2.4. Code Generator</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>5.3. Directory Service</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>5.3.1. The Master Class</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>5.3.2. Remote Object Reference List</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>5.3.3. Main Program</td>
<td></td>
<td>110</td>
</tr>
</tbody>
</table>

The RL system is implemented in C++ on the GNU/Linux operating system and uses TCP/IP for communication. It should be readily portable most versions of UNIX and UNIX derivatives. As stated earlier, the system consists of three major parts:

- The run-time library
- The IDL compiler
- The directory service

How these three parts interact is presented in chapter 4. In this chapter we will describe how the three parts are implemented.

5.1. Run-time library

The primary task of the library is to define the tasks that make up the system, and to set up communications between these tasks. In addition, the library implements support services.
As we can see from the overview (see section 4), communication between the tasks and the Directory Service is handled by the communication protocol. The protocol is implemented by the run-time library, and will be presented later in this section.

5.11. Tasks

A task is implemented as two or more threads in a process. The main thread will be used to run the main program of the task (that the user supplies), while the listening thread will listen for incoming connections. For each connection, a communication thread will be used to process entry request and call arguments that are received from the caller.

The listening thread waits for incoming connections from the network. If no connections are available, the listening thread blocks and does not consume CPU resources.

If a caller issues an entry call, a local stub procedure takes the entry request and arguments and sends them over the network. This happens in the main thread of the calling process. The stub procedure is generated by the IDL compiler, described in section 4.4.

The communication thread in the called process reads the request and any arguments from the network and stores them in an entry list.

When the task accepts an entry, if a matching entry call is available from the entry list, the entry arguments are fetched from the entry list by the main thread and the call and arguments are deleted from the list. Then the local method implementing the entry is called. On return from the local method, the return value, in/out and out parameters are stored in the return list. Note that all this happens in the main thread of the called process. If a matching entry call is not available from the entry list, the process blocks (using a condition variable, see section 2.4.2.1) until one becomes available. The main thread sends return values and output arguments over the network to the calling process, thus completing the entry call.

The program flow in the listen thread looks like this:

```c
if(new connection) {
    start communication thread
    add new communication thread to list of commthreads
} else {
    sleep;
}
```

And in the communication thread:

```c
if(entry data can be read from network) {
    store entry data in entry list;
} else {
    sleep;
}
```
(in reality the above would be implemented using a blocking OS call, such as the file descriptor select() function call)

This implementation approach means that:

- The algorithm in the main thread can execute the entry body immediately upon accept, if a matching entry call has already been processed by the communication thread.

- If no data is available for reading from the network, and no data is to be sent, the communication threads are blocked. Consequently, they do not consume CPU resources, and are not scheduled for execution by the operating system. Similarly, if no new connections are incoming, the listening thread is blocked.

- We have to synchronize access to two shared data structures, the entry list and the list of communication threads (referred to as commlist). Access is controlled by mutexes. Communication threads insert entries to the entry list, and the main thread removes entries from it. The listen thread inserts references to communication threads into the commlist and the communication threads remove them when they terminate.

By basing the implementation of the rendezvous mechanism directly on TCP/IP, one should be able to get an implementation with the same performance as, or only marginally slower than, message-passing libraries such as MPI. An alternative would be to implement the library on top of a higher-level library, e.g. MPI, but this would lower performance.

5.1.2. Implementation and usage of RL

Class overview The library is made up of a number of classes. They are briefly described in the following.

Thread - This is the base class for all threads. It contains 3 methods, Start(), Run() and WaitFor(). The Start() method executes Run(), and WaitFor() causes the thread to wait until it has finished executing before it returns. Run() is a virtual method that each thread must specify

MainThread - The MainThread has only got one method, Run(). It will execute the task’s Run() method, which is where the user has specified what is to be done.

ListenThread - This thread’s Run() method listens for incoming connections, and starts a new CommThread when a connection is accepted.
Figure 5.1: Abbreviated Class Diagram. A complete diagram can be found in the appendix.
CommThread - A CommThread handles one connection. It is initialized by the Listen-
Thread, and will exist as long as it receives data.

Location - A location is the "address" of a remote object. The Location class is the base
class of all other location classes. For each kind of network transport, a subclass
of Location can be defined. A location consists of two pieces of information: a
transport-dependent network address, and an object ID. Two virtual functions,
GetObjectID() and Clone() are explained below.

TCP Location - The TCP Location class is used for specifying TCP/IP addresses. Two
pieces of information are stored, a TCP/IP address, and an object ID. The GetObject-
ID() method returns the object that exists on the specified location. The Clone()
method clones the object (on this TCP location) and returns the clone.

Object - This is the base class for all object reference classes. The methods GetLocation()
and GetTask() return pointers to the object’s location object, and task object,
respectively.

ObjectServant - This is the base class for the servant objects classes. For an explanation
on servant objects, see section 4.3.

Protocol - This is not a class, but it defines the message header.

struct header {
    unsigned char proto_version;
    unsigned char encoding;
    unsigned char message_type;
    unsigned char reserved;
    int message_length;
    int object_id;
    int opname_length;
}

Task - The Task class is the base class of all RL tasks. Its main methods are Start() and
WaitFor(). In addition the Task class contains methods for locking and unlocking
shared resources.

CommItem - This is one of two helper classes for the Task class. It represents one item
in the communication queue and is used by the CommQueue class. A CommItem
stores the object’s ID, the name of the operation it holds, its data, the length of
the data, and also a pointer to the transport object over which this item was sent
(or is supposed to be sent). The class also stores pointers to the next and previous
CommItem in the CommQueue.

CommQueue - The other of the two helper classes. The class represents a communi-
cation queue that holds communication requests. A CommQueue stores pointers
to the head and tail of the queue, and has methods Add(), Remove(), Peek(),
Search(CommItem) and Remove(CommItem). Peek() returns the first item in
the queue, Search(CommItem) searches for the specified item, and Return(CommItem)
returns the specified item. Add() and Remove() adds an item to the end of
the queue, or removes an item from the head of the queue, respectively.

TCPTransport - TCP/IP transport mechanism. This class is a thin wrapper around
native TCP/IP system calls, like socket(), bind(), accept(), recv(), send(), and select(). In addition it contains methods for reading the connection state (unbound,
listening, connected), and for setting a socket to blocking or non-blocking commu-
nication.

How the Classes are Used  The task concept is central to the library. A complete task
process is made up of several of the classes explained above. The Task class is of course the
most important, with its helper classes CommItem and CommQueue that constructs the
queue where incoming requests are stored before they are processed. The MainThread,
ListenThread and CommThread define the 3 threads that are included in a task. In
addition, Task has a reference to a TCPLocation object, and to a TCPTransport object.
The TCPLocation specifies the IP address of the computer running the task, and the
object id of the task. The TCPTransport object provides the task with a communication
mechanism/subsystem.

The TCPTransport class is, as mentioned above, a thin wrapper around TCP/IP system
calls, and can be either unbound, listening or connected. A TCPTransport object that
has successfully called the listen() method changes its state to listening. When connect() is
executed, the state is set to connected. After connecting, the TCPTransport object
can use its read() and write() methods to receive and send data across the connection.

In the event diagram in figure 5.2, the various states of the task process can be seen.
Note however, that unlike a normal event diagram where the program is only in one state
at a time, this illustrates a process with 3 (or more) threads that execute at the same
time.

When a task is created, the two threads Main and Listen are created. The MainThread
starts running the main program (i.e. the task's Run() method), and the ListenThread
listens for incoming connections. Then, the tasks' main program returns and continues
do whatever the user has programmed it to do. The ListenThread will start a new
CommThread for each new incoming connection. Each new communication thread will be
added to a list of communication threads called the commlist. Note that the ListenThread
never changes state. It is always listening for new connections. For each new connection
a new TCPTransport object is also created. A pointer to this object is passed to the new
communication thread which then uses this connection when communicating.

If there are errors when reading from the connection, or the other end closed the con-
nection, the commthread will remove the transport object, close the connection, and
terminate itself.
A CommThread has references to the task that it belongs to, and to the ListenThread that created it. This is necessary because the ListenThread has got the list over all the communication threads that have been created. When the CommThread terminates it will have to shut itself down. If this is not done, memory and available file handles\(^1\) will be wasted. Unless we store a list over the existing communication threads, we will not be able to shut them down when they are done.

The CommThread will first listen for a operation request. When this arrives over the connection, it will try to place this request in the queue. If the queue is being accessed, the CommThread will block, and wait until the queue is available before placing the request in it. When this is done, the thread will wait for a new operation request. If there are errors when reading from the connection, or the other end closed the connection, the commthread will remove the transport object, close the connection, and terminate itself.

The MainThread accepts an operation call when the Accept() operation is called in the user-specified algorithm. It can be a specific operation call, e.g. named “insert”, or it can be any call that arrives. Then the thread will look in the request queue for this call. If no matching calls lie in the request queue, the thread will block until a matching call arrives. When an operation call is accepted, and afterwards found in the queue, the operation will be executed and the result sent back to the task that requested it.

The ObjectRef and ObjectServant classes are not involved directly in the implementation of the run-time library. These classes are the base classes of all reference objects and servant objects, and these are created by compiling the IDL files specified by the user.

The flowchart (figure 5.3) illustrates the flow of a task process.

**Communication mechanism** The different tasks in an RL application communicate using TCP/IP. The low-level details of TCP/IP communication are partly abstracted away in the classes TCPTransport and TCPLocation. For more information on TCP/IP programming, see section 2.6.1.

**Synchronization** The problems concerning synchronization, and the methods used to solve them are discussed in section 2.4.2.

The RL library uses blocking to synchronize access to the shared resources. The shared resource is blocked by the process or thread that uses it until the process is finished with the resource. The blocking is accomplished by using POSIX thread methods. The POSIX threads supports 3 different methods for synchronization by blocking.

- **Mutexes**.
- **Condition variables**

\(^1\)A Linux process can have a maximum of 1024 open files, and one connection counts as one file
Figure 5.2: Event diagram
Figure 5.3: Flowchart for a Task-process
• **Semaphores.**

In the RL library we employ mutexes in addition to condition variables. One example is shown below.

    pthread_mutex_lock(&task->request_queue_mutex);
    task->request_queue.Add(item);
    pthread_cond_broadcast(&task->request_queue_cond);
    pthread_mutex_unlock(&task->request_queue_mutex);

The request queue mutex is used to control access to the request queue. This is the queue that stores incoming requests from other tasks.

We have two functions, `pthread_mutex_lock` and `pthread_mutex_unlock` that locks and unlocks the mutex. After locking it, we proceed to perform an operation on the queue. We then call `pthread_cond_broadcast`. This sends a wake up call (a SIGNAL) to all threads that are waiting on the specified condition. Each awakened thread will then start to run as soon as the request queue mutex is unlocked.

5.1.3. **Communication protocol**

**Protocol** To provide communication between the various tasks in the system it is necessary to define a protocol. The protocol must enable us to accomplish our goals on efficiency.

Our protocol will be implemented on top of the TCP/IP protocol. We therefore add an extra header to each message that is transported across the network. The header contains information about (among other things) the message type, the object id of the object that sent/received the message, the operation name and the operation data.

The main feature of the RL protocol is that it adds the possibility to encapsulate data as messages, not just a stream of data. Note however, that the protocol does not add any extra security features, or features to ensure that messages are delivered. This was deemed unnecessary, since TCP guarantees delivery of data.

**The protocol header** We add a fixed-format 16-byte header on each message that is sent across the network. The fixed header is followed by 2 fields with varying length. The header has the following fields (ref: 5.4).

1. **Protocol version** - This field is 1 byte in size, and must, for the current version of the protocol, be zero. Future versions of the protocol may have another number in this field.

2. **Encoding** - This field is 1 byte in size. Bit 0 defines the byte order (0 for little-endian and 1 for big-endian). Bits 1-7 are reserved and must be zero.
3. **Message Type** - This field is 1 byte in size. It can specify either an error message, normal method call, normal method return, or an exception return.

4. **Reserved** - This field is 1 byte in size. It is reserved, and must be zero.

5. **Total Length of Message** - This field is 4 bytes in size. It specifies the total length of the message, including the header, in bytes.

6. **Object ID** - This field is 4 bytes in size. It contains the object identifier of the object that sent (or will receive) the message.

7. **Length of Operation Name** - This field is 4 bytes in size. It contains the length of the operation name, in bytes, including the terminating null character. For message types other than normal method calls, this field can be set to zero.

8. **Null Terminated Operation Name** - This field takes an unknown number of bytes, depending on the length of the name. For example, the operations named `insert` and `remove` are both six characters long. In these cases, the length of this field would be seven bytes.

9. **Operation Data** - This field contains the operation data. The length of the field can be found from the **Total Length of Message**-field. For a method call, the operation data field contains the arguments from the calling task. For a method return, this field will contain the return values and output arguments.

The parameters and return values in the operation data field is marshaled according to the following rules:
• Simple types are marshaled directly into the operation data field without any conversions. No information on the size and type of the data item is included, because this information can be deduced from the IDL specification for the operation.

• Strings are marshaled in the following way:
  
  – First an unsigned long (four-byte) field is added which contains the length of the string, in bytes, including the null-terminating character.
  – Then the data of the string itself is added.
  – Lastly the null-terminating character is added.

• Arrays are marshaled in the following way:

  – First an unsigned long (four-byte) field is added that contains the length of the array, in bytes. The number of elements in the array can be deduced by combining this information with the type information in the IDL specification for the operation.
  – Then the data of the array itself is inserted.

There is no padding between the data items in the operation data field, and no particular alignment of data is enforced. If the receiving process requires that the data is aligned in a particular way, it must move the data so that it is properly aligned after it has been read.

5.2. IDL Compiler

As stated in section 4.4, the IDL compiler takes an interface specification as input, and outputs a set of C++ source code files. The interface specification is given as an IDL file. The C++ source files describe the interface and implement the stub and skeleton methods required to marshal and unmarshal calls to the interface’s operations, as described in section 2.2.

The compiler program itself uses the pipes and filters architectural style, as described in [30]. The input (in this case the IDL file) is transformed in a series of steps to the output (the set of C++ code files). Each step (also called a filter) takes as input the output from the previous step. In the filter, the data can be transformed to a different representation. The sequence of steps is called a pipeline.

In the IDL compiler, the pipeline consists of three main steps (see figure 5.5):

• The lexical scanner takes as input the IDL file and outputs a stream of tokens. Each token corresponds to a primitive element in the language. One example of such primitives are keywords, that have special meanings in the language, such
Figure 5.5.: The IDL compiler pipeline
as “interface” or “void”. Another example is identifiers, that name user-defined elements.

- The parser takes the stream of tokens from the scanner and builds a parse tree. A parse tree is a hierarchical data structure that represents the grammatical structure of the input file. The parser contains a description of the language’s syntax, and as it recognizes a syntactical element, it adds it to the parse tree.

- The code generator generates the C++ code by traversing the parse tree. This involves
  - Building a table of the symbols encountered.
  - Performing semantic analysis to ensure that the symbols are used correctly (that, for example, a symbol denoting a type is not used where a symbol denoting a constant value is expected).
  - Outputting code corresponding to the syntactical elements found in the parse tree, looking up definitions in the symbol table as needed.

5.2.1. Internal Structure

The compiler consists of the following main components, as shown on figure 5.6.

- A main program that parses command-line arguments and controls the process of compiling.

- The ParseControl class that is used as the top-level data structure in the program. This class contains the compiler’s state, and pointers to instances of the IDLLexer, NodeState and CppCodeGenator classes, as well as to the root of the tree. In addition, it contains methods for selecting the current input file, and for error reporting.

- The IDLLexer class that implements the scanner. It inherits from yFlexLexer, and is described in section 5.2.2.

- The parser, implemented as the global function yyparse. It is described in section 5.2.3

- The set of classes for the different node types in the parse tree, one per node type. They are described in section 5.2.3.

- The NodeState class that is used for memory management and for storing line number information in the nodes of the parse tree. It inherits from the YNODESTATE class, as described in section 5.2.3.

---

2The naming is unfortunate. It should really be called e.g. CompilerControl instead, since it is used throughout the entire compiler.
Figure 5.6: Internal structure of the IDL compiler
• The CppCodeGenGenerator class that contains most of the functionality required for outputting C++ code from the parse tree. It is described in section 5.2.4.

• The SymbolTable class, that is used for the symbol table. It is described in section 5.2.4.

• The ParameterTable class, that is used for generating a table of the parameters to an operation, complete with the concrete types for these parameters. The information in this table is used as a basis for generating the code for marshaling and demarshaling. It is described further in section 5.2.4.

5.2.2. Scanner

As described earlier, the lexical scanner transforms an input file into a stream of tokens, with each token corresponding to a primitive element of the language (see figure 5.7). The elements can be divided into the following categories:

• **Keywords**, like “module”, “string”, “typedef”, are words that have a special meaning in the language.

• **Identifiers**, like “myInterface”, “Insert”, “hello”, are strings that are used to name entities that are defined in input file. Identifiers can be chosen freely by the user, subject to a set of constraints (for example, that they must begin with a letter). Keywords cannot be used as identifiers, but they can be used as part of an identifier.

• **Literal values**, like “2”, “3.14”, “4.53e13”, “a”, can be used in expressions, for example to set a constant to a given value.

• **Punctuation** have, like keywords, a special meaning in the language, but consists entirely of non-alphanumeric characters. Most operators are punctuation characters. Examples of punctuation include “{”, “}”, and “;”.

• **Comments** can be used to document the interface and help explain how it is to be used. Comments are discarded by the scanner, and do not form part of the output stream of tokens.

5.2.2.1. Generating a Scanner Using Flex

The scanner used in the IDL compiler is generated using the flex scanner generator[31]. Flex reads an input file containing a description of the scanner, and outputs a C or C++ source code file that implements the scanner. The scanner description file consists of three kinds of elements:

• **Definitions** of names, which helps simplify writing the scanner.
Figure 5.7: The scanner

definitions section
  
rules section

user code

Figure 5.8: Layout of a flex source file

- Rules, that tells the scanner generator how to recognize tokens, and what to do when a token is recognized.
- User code that is pasted verbatim into the generated scanner.

The layout of a flex source file is shown in figure 5.8. It consists of three sections. First, the definition section, that contains the definitions, and options which control how the scanner is generated, then the rules section, and lastly the user code section.

A small excerpt from the flex source file is shown in figure 5.9. The first five lines are examples of definitions, in this case its definitions of what constitutes an identifier, a literal number (in octal, hexadecimal and decimal), and whitespace. The rest of the lines, after the "%%" marker, are the rules. A rule consists of two things:

- A pattern that tells the scanner how to recognize a token.
- An action that is C/C++ code that is executed whenever this pattern is recognized in the input stream.

Patterns and definitions are expressed with so-called regular expressions. Regular expressions enables one to write patterns that can be quite complex, using a terse, straightforward syntax. We will not cover regular expressions here, but many references for writing regular expressions exists, see e.g. [33].
Figure 5.9: Excerpt from the flex source file for the IDL compiler (idl_scanner.l)

Actions are normal C or C++ code and can do anything one wants, but most often a token code is returned to the parser to indicate what kind of token was just read. For example, when the sequence of characters “abstract” is encountered, the scanner returns the token code K_ABSTRACT to the parser.

5.2.2.2. The Generated Scanner

The flex program generates a C++ class called yyFlexLexer that implements the scanner. This class has the following important methods:

- **yyflex():** This is the actual scanner implementation. The parser calls this method repeatedly; each method call returns one token from the input stream.
- **YYText():** This method returns the text (i.e. value) of the most recently matched token.
- **lineno():** Returns the current line number in the input file.
- **switch_streams():** Switches the file streams used for input\(^3\). Used for specifying which file will be scanned.

The yyFlexLexer class is subclassed by the IDL lexer class, which is used as the actual scanner class in the IDL compiler. In addition to the functionality provided by yyFlexLexer, IDL lexer has a method for error reporting, and a pointer to an instance of the ParseControl class.

\(^3\) And for output, but this is not used in the IDL compiler.
5.2.3. Parser

The parser generates a parse tree from the stream of tokens output from the scanner, as shown in figure 5.10. It does this by recognizing that a particular sequence of tokens form a particular grammatical construct in the language. For example, the keyword “interface” followed by an identifier constitutes an interface header, and an interface header followed by an interface body constitutes an interface declaration. Whenever a construct is recognized, it is inserted as a new node into the parse tree.

The parser knows how to recognize grammatical constructs because it has built-in knowledge of the language’s syntax. One way of describing the syntax of a language is as a set of rules that describes how to form the language’s constructs from its constituent parts. A language description that is organized in this way is called a context-free grammar. A context-free grammar has four parts [34]:

1. A set of tokens, or terminals; these are the atomic symbols in the language, and form the leaves of the parse tree. They correspond to the tokens found by the lexical scanner.
2. A set of nonterminals; these are the variables representing constructs in the language.
3. A set of rules called productions that identifies the components of a construct.
4. A nonterminal chosen as the starting nonterminal; it represents the main construct of the language. The starting nonterminal becomes the root of the parse tree.

For humans, the most common formal system for expressing context-free grammars is the “Backus-Naur Form”, or “BNF”, which was developed in order to specify the language Algol 60 [35, 32].

A simplified grammar for an interface declaration in IDL can be written in BNF as follows:

\[
\begin{align*}
\langle \text{interface declaration} \rangle & \quad ::= \quad \langle \text{interface header} \rangle \{ \langle \text{interface body} \rangle \} \\
\langle \text{interface header} \rangle & \quad ::= \quad \text{interface identifier} \\
\langle \text{interface body} \rangle & \quad ::= \quad \langle \text{empty} \rangle \mid \langle \text{operation list} \rangle
\end{align*}
\]

The symbols inside \textit{<angular brackets>} are nonterminals, and the symbols in a \textbf{bold sans-serif font} are terminals.

From the first rule it follows that an interface declaration consists of an interface header, followed by the token \{, followed by an interface body, followed by the token \}. The second rule states that an interface header is an interface token followed by an identifier token, and the third rule says that an interface body is either the empty list, or an operation list. (The definition of an operation list would presumably be in another part of the language's grammar).

A formal grammar such as BNF selects tokens only by their classifications: for example, if a rule mentions the terminal symbol identifier, it means that any identifier is grammatically valid in that position. The precise name of the identifier is irrelevant when parsing the input: if ‘interface hello’ is grammatical then ‘interface bye_bye’ or ‘interface guten_tag’ are equally grammatical. The terminal’s semantic value holds the rest of the information, for identifiers, it can be the identifier’s name (‘hello’, ‘bye_bye’, or ‘guten_tag’), and for integer constants, it can be the integer’s value (like 4, 32, or 2931). A nonterminal can also have a semantic value; this value is usually a tree structure that describes the components of the nonterminal.

There are two different ways a parser could construct the parse tree. A top-down parser works from the root (top) of a parse tree towards the leaves. A bottom-up parser works from the leaves (bottom) of a parse tree towards the root.
Figure 5.12: Bottom-up parser.

```
interface_dcl : interface_header '{' interface_body '}'
  { $$ = NEW(interface_dcl)($1, $3); }
  | interface_header '{' '}'
  { $$ = NEW(interface_dcl)($1, LIST(export_list)); }
interface_header: K_INTERFACE IDENTIFIER interface_inheritance_spec
  { $$ = NEW(interface_header)
    (false, false, NEWID($2), $3); }
interface_body : /* empty */
  | operation_list { $$ = $1 }
```

Figure 5.13: Bison source for interface declarations (from idl_parser.yy, slightly modified)

As an example, consider the following token stream:

```
interface identifier { }
```

A top-down parser would start with an *interface declaration*, and divide that construct into its constituent parts until the leaves of the tree consists of only terminals, as shown on figure 5.11. A bottom-up parser would start with the terminals and built the tree from the bottom up until an *interface declaration* could be formed, as shown on figure 5.12.

5.2.3.1. Generating a Parser Using Bison

The IDL compiler’s parser is generated using the *bison* parser generator program [32]. The input to the bison program is a file describing the language’s grammar in what is essentially machine-readable BNF, and containing a set of *actions* that are performed whenever one of the rules (productions) in the grammar is matched (similar to the way actions are executed by the scanner).

Each action contains the C++ code necessary to generate a node in the parse tree that corresponds to the grammatical construct recognized. Figure 5.13 shows the part of the bison source file that contains the rules and actions for an interface declaration. The
%node interface_dcl interface = {
    interface_header  *header;
    operation_list    *body;
}

Figure 5.14: Treeccc source file for the interface declarations node type (from idl_tree.tc, slightly modified).

actions for each rule is between the curly brackets ( { ... } ). The code inside the actions is normal C++ code, but in addition one can use the special symbols $$ and $1 ... $n. $$ is the semantic value for the syntactic construct being created, while $1 ... $n is the semantic value of each of the components that the construct consists of. Most of the work done in the actions consists of creating new nodes and assigning them to $$, thereby building up the parse tree.

The output from the bison program is a C source code file containing the definition of the yyparse function, which implements the parser.

5.2.3.2. Parse Tree Nodes

The number of different node types in a parser for a complex language like IDL can be very large. Each node type is typically implemented as a C++ class. While it is possible to implement all these classes manually, this leads to much repetitive work.

For the RL IDL compiler, the treeccc program [36, 37] was used to generate the node classes from a specification of the node types. The specification contains a listing of all possible node types, and for each node type, the data that must be stored in the node; for example, what children are possible in a node, and the semantic values of any terminals.

Figure 5.14 shows the node type for interface declarations. As can be seen, the interface_dcl node type inherits from the (abstract) node type interface, and contains the following data members: a pointer to an interface_header node called header, and a pointer to an operation_list node called body. The trecc program automatically generates the corresponding node class, complete with an appropriate constructor.

It is also possible to define operations that are to be performed on the nodes. These operations can implement, for example, semantic analysis, symbol table lookups, optimizations, or code generation. An operation can be declared for a set of related node types, and separate definitions of the operation can be made for each node type, or a set of node types can share a definition. The definitions of a node type contain normal C++ program code. One example is the gen_code_cpp operation that is defined for all node types in the IDL compiler. This operation generates output C++ code for a given node, and is described further in the next subsection.

When an operation is declared for a given set of node types, the treeccc program checks
that the programmer does not forget to include definitions of the operation to cover all
the node types. This helps reduce the number of possible bugs in the compiler.

The node classes uses an instance of the `NodeState` class that acts as the node memory
manager and node creation facility, and as an interface for getting line number informa-
tion from the scanner. Storing line number information in the nodes themselves means
that it is possible to print out helpful error messages that includes the location of the
error, even if the error is discovered after the scanning and parsing steps are complete.

5.2.4. **Code Generator**

The code generator, as stated earlier, traverses the parse tree and outputs C++ code for
the definitions found in the tree (see figure 5.15).
All nodes in the parse tree have a `gen_code_cpp` method. When invoked, this method generates code as appropriate for that node. However, the bulk of the functionality for code generation is implemented in the `CppCodeGenerator` class. The `gen_code_cpp` method simply acts as a controller and invokes appropriate methods in the `CppCodeGenerator` class. In addition, it may need to define symbols in the symbol table. It does that by invoking the appropriate methods in the `SymbolTable` class.

### 5.2.4.1. `CppCodeGenerator` Class

The `CppCodeGenerator` has methods for

- Opening and closing output files and/or directories (modules are placed in subdirectories).
- Defining classes for interfaces, references and servants.
- Opening and closing scopes. A scope is a namespace — modules and interfaces, for example, form scopes.
- Outputting code for marshaling and unmarshaling operations. This is further described in the following text.
- Outputting code for type declarations. (Types are declared with the typedef keyword in IDL, in the same way as in C or C++).
- Looking up scoped names from the symbol table, as described below.

### 5.2.4.2. `SymbolTable` Class

The `SymbolTable` class has methods for

- Adding symbols to the symbol table
- Looking up symbols in the symbol table

Symbol tables are hierarchical. Each scope has its own symbol table, and each symbol in the table can contain a symbol table of its own. The global, file-level namespace (outside all modules and interfaces) is called the root symbol table.

When looking up names, the following algorithm is used:

1. If the name can be found in the current scope, then it has been resolved.
2. If not, try to resolve it in the parent of the current scope.
3. Repeat 1 and 2 until the name has been resolved, or the root scope is reached.

In a given scope, when resolving a name that consist of a single identifier, the algorithm
only needs to check if that symbol can be found in the scope's symbol table. When
resolving a name that consists of multiple components, the following algorithm is used:

1. Check if the first component of the name can be found in the current scope. If yes,
then proceed, if not, the name could not be found.

2. Check if the symbol found in step one has a child symbol table (i.e. it is a scope).
If yes, then proceed, if not, the name could not be found.

3. Check if the second component of the name can be found in that child symbol
table. If yes, then proceed, if not, the name could not be found.

4. All components of the name has been checked, then the name has been resolved, if
not, then proceed.

5. This process of looking up name components in child symbol tables continues until
either:

- A name component could not be found in the symbol table where it was
  expected to be. The name could not be found.
- A symbol that was expected to have a child symbol table did not have one.
  The name could not be found
- All components of the name has been checked. The name was found.

Generating Code for Operation Parameters  For an interface's operations, code for
marshaling and unmarshaling the operation's parameters need to be generated. To be
able to do this, one needs to collect the following kinds of information:

- The name of the parameter
- If it is an input, output or in/out parameter
- The type of the parameter

The name of the parameter, and if it is an input output or in/out parameter, can be
found relatively easy from the parameter declaration nodes in the parse tree. However,
finding the type of the parameter can be more complex. This is because the user can
define types with the typedef keyword, and the compiler needs to know not only the
type’s name (as defined by the user), but also the actual, underlying type. Note that the
typedef keyword in the RL IDL can only be used with the types that RL supports. This
information must be looked up in the definition of the type, which is located in another part of the parse tree.

This functionality is implemented in the ParameterTable class. The ParameterTable’s InsertParameter method takes a parameter declaration node as input and inserts the required information into a table. If the type of the parameter is a primitive type (such as an integer, a floating-point number, or a string), the required type information can be found directly from the parameter declaration node itself. If, however, the type is a user-defined type, the the type’s definition must be looked up. This is done by first looking up the type’s name in the symbol table, using the algorithm described in the previous subsection, and verifying that it is indeed a symbol for a defined type. A pointer to the type declaration node was recorded in the symbol’s symbol table entry when the symbol was first defined, and the required type information can then be found in the type declaration node and its subtree.

An example can help clarify this process. If, for example, we have the following IDL:

```idl
module A {
    typedef double double_array[0];
    interface B {
        void myOp(in long param1, double_array param2);
    };
};
```

The first parameter (“param1”) of the myOp operation is not a problem. It is of the primitive type “long”. However, the type of the second parameter (“param2”) is not readily apparent. All that is known is that it is named “double_array”. The InsertParameter method must therefore look up the symbol in the symbol table. It will first try to look up the symbol in the “B” interface’s scope, but this will fail. Then the symbol will be looked up in the “A” module’s scope, and this time the name resolution will succeed. From the type declaration node it can be deduced that the symbol “double_array” denotes a (variable-length) array of double-precision floating-point numbers. The compiler now has the information it needs to generate the marshaling and unmarshaling code for the operation.

This process must also work if a name is a redefinition of another name; if for example the following IDL is given:

```idl
typedef double double_array[0];
typedef vertex_coordinates double_array;
```

Then the type resolution algorithm needs to be able to resolve the name “vertex_coordinates” to the type definition “array of doubles”.

---

4 A list, actually.
5.3. Directory Service

The directory service is very simple, and is nothing more than a task exporting a remote interface. This interface is defined in a IDL file and looks like this:

```java
module Directory{
    interface Service{
        void Bind(in string objectpath, in string reference);
        void Unbind(in string reference);
        string Resolve(in string reference);
    }
}
```

Compiling this IDL file creates two classes: Directory::ServiceRef and Directory::ServiceServant. These two classes are used by a developer when using the interface as described in 4.5. More on how to create IDL files and the result of compiling them can be found in 4.4.

The three methods defined in the directory service IDL file is implemented in a class called Master. A brief overview is given below while the entire code is included in the appendix H.

5.3.1. The Master Class

The Master class contain five methods, whose main purpose is to maintain the list of remote object references. This object inherits from the RL::Task class as explained in section 4.2.1, and redefines the pure virtual method Run(). In the Run() method, a Directory::ServiceServant object is created and then set to accept incoming calls in a loop. The other functions Master contains are Bind(), Unbind() and Resolve(). These are the methods accessible through the interface, and maintain the list of references. They are explained in 4.5. Since all three methods need to traverse the list to find the correct item a Traverse() method exists. This method searches through the list using the reference string. It is not accessible through the interface.

5.3.2. Remote Object Reference List

The list is a linked list, where each link is a data structure called Item. This data structure contains the registered remote object, its reference name and a pointer to the next item. When a list is first created it only contains two pointers, a pointer to the empty head and a position pointer. Each item in the list has a next pointer that points to the item that was created next, e.g. head's next points to the oldest item in the list while the last item is the newest. The last item's next is a NULL pointer.
5.3.3. Main Program

The main program creates a Master object and invokes \textit{Start()} and \textit{WaitFor()}. These two are inherited from \textit{RL::Task} and described in 4.3.
6. Programming Examples

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.</td>
<td>Producer/Consumer Problem</td>
<td>111</td>
</tr>
<tr>
<td>6.1.1.</td>
<td>Implementation Overview</td>
<td>112</td>
</tr>
<tr>
<td>6.1.2.</td>
<td>Process Management</td>
<td>115</td>
</tr>
<tr>
<td>6.1.3.</td>
<td>Initialization</td>
<td>115</td>
</tr>
<tr>
<td>6.1.4.</td>
<td>Communication</td>
<td>117</td>
</tr>
<tr>
<td>6.1.5.</td>
<td>Synchronization</td>
<td>119</td>
</tr>
<tr>
<td>6.2.</td>
<td>Area Computation Problem</td>
<td>120</td>
</tr>
<tr>
<td>6.2.1.</td>
<td>Problem description</td>
<td>120</td>
</tr>
<tr>
<td>6.2.2.</td>
<td>Common structure and shared elements</td>
<td>121</td>
</tr>
<tr>
<td>6.2.3.</td>
<td>Shared elements</td>
<td>124</td>
</tr>
<tr>
<td>6.2.4.</td>
<td>MPI</td>
<td>124</td>
</tr>
<tr>
<td>6.2.5.</td>
<td>OOMPI</td>
<td>125</td>
</tr>
<tr>
<td>6.2.6.</td>
<td>CORBA</td>
<td>125</td>
</tr>
<tr>
<td>6.2.7.</td>
<td>RL</td>
<td>126</td>
</tr>
</tbody>
</table>

In this chapter, two programming examples are presented. Each problem was implemented in a multitude of communication libraries so that the different libraries could be compared.

The first problem is a classical interprocess communication problem called the producer/consumer problem. It was implemented using TCP/IP directly, using CORBA, and using the Rendezvous Library, and is described in section 6.1.

The second problem is a small numerical problem that was used to test the performance of the different libraries. It was implemented using MPI, OOMPI, CORBA and the Rendezvous Library, and is described in section 6.2.

6.1. Producer/Consumer Problem

When developers use interprocess communication there will be synchronization problems they must be prepared to solve. IPC problems are a set of problems meant to challenge
developers by pointing at relevant synchronization problems that arise when processes communicate. Interprocess synchronization is described in detail in section 2.4.2, where we pointed out the danger of race conditions, which may result in starvation and deadlocks if not solved properly.

The producer-consumer problem consists of a producer process and a consumer process that share a common, fixed-size buffer. The producer puts items into the buffer, and the consumer takes them out. The buffer processes the items in a FIFO\(^1\) order and the producer and consumer run normally until one of two conditions occur:

- **The buffer is full:** This will cause the producer to block. If the producer didn’t block, it would try to insert items into a buffer that is already full.

- **The buffer is empty:** This will cause the consumer to block. If the consumer didn’t block, it would try to remove non-existing items.

To avoid a deadlock where both the producer and consumer blocks, it is necessary to unblock the consumer when the buffer is no longer empty, and to unblock the producer when the buffer is no longer full.

We implemented the problem in TCP/IP, CORBA and RL. TCP/IP was selected because the Rendezvous Library is implemented in TCP/IP, thus making it relevant to test on a smaller scale. We used CORBA because it is object-oriented and shares many concepts with the Rendezvous Library.

### 6.1.1. Implementation Overview

All implementations use three processes to simulate the problem description: a producer, a consumer and a buffer. The producer’s only job is to generate the items, which are random integers, and pass them on to the buffer as long as it is not full. The consumer receives items from the buffer and prints them out one by one.

As implied earlier the producer process and consumer process has two states: running and blocked. The reason for this is to prevent the producer process from sending data to the buffer process when the storage buffer is full, and to prevent the consumer process from requesting data when the storage buffer is empty. One can summarize the possible state transitions like this:

- If the producer process tries to produce an item, and the storage buffer is full, the producer process will block.

- If the producer process is blocking, and the storage buffer is no longer full, the producer process should be unblocked.

\(^1\)First In First Out
- If the consumer process tries to consume an item and the buffer is empty, the consumer should block.

- If the consumer process is blocking, and the buffer is no longer empty, the consumer should be unblocked.

The buffer process passively waits for operation requests from the producer and consumer processes. As a response to a request, the buffer process can perform one of two operations:

**Produce** Receives an item from the producer process and inserts it in the buffer’s data storage.

**Consume** Removes an item from the process’ data storage and returns it to the consumer process.

In figure 6.1 the state of the producer is shown. The producer starts in the unblocked state and inserts items into the buffer until it is full. At this point the producer blocks until the buffer is non-full, upon which it resumes its normal operation of producing items and inserting them into the buffer.

Figure 6.2 shows the state diagram of the consumer. As we can see it is almost identical to the one for the producer with regard to states and transitions. The consumer is unblocked and removes items from the buffer until it is empty. As soon as the buffer is empty, the consumer will be blocked, and it will remain in this state until the producer has inserted a new item, and the buffer is no longer empty.

The states for the buffer is shown in figure 6.3. When starting up the buffer is empty, and will wait for “insert” requests from the producer. The consumer will not be able to
Begin, buffer empty

Waiting for "Insert"

Inserting

Insert

Buffer non-full

Remove

Removing

Buffer non-empty

Waiting for "Insert" and "Remove" requests

Buffer full

Waiting for "Remove" requests

Figure 6.2: Consumer State Diagram

Figure 6.3: Buffer State Diagram
send since it is blocked (ref: 6.2) if the buffer is empty. In the next state (after an item has been inserted), the buffer will begin to wait for both “remove” and “insert” requests. These requests will be processed by the buffer as long as one of the conditions mentioned earlier does not occur. If the buffer runs out of items, only “insert” requests will be received, and if the buffer is filled up, only “remove” requests will arrive.

6.1.2. Process Management

In all three version the processes are started and handled manually. The producer, consumer, and buffer processes are part of the same program, and one selects if a particular process is to act as a producer, consumer, or a buffer by specifying an argument on the command line when starting the program. An example is given below for the CORBA version.

For the CORBA version, the naming service must be started before running any of the three processes. The location of the naming service is given on the command line, as in the following example:

./prodcon -ORBitRef NameService=corbaname::localhost:8088 p

This command line above shows the producer/consumer program, “prodcon”, executed as a producer (p) with the naming service on the local machine. Since the producer and the consumer depends on there being a buffer, one has to start the buffer process first. The location of the naming service is the only location that has to be hard-coded, the processes can find the locations of other processes by querying the naming service.

In the TCP/IP and RL versions of the program, the locations of the processes are hard coded into the program. Although RL has a directory service that performs a similar function to CORBA’s naming service, it was not used in this example².

6.1.3. Initialization

6.1.3.1. TCP/IP

In the TCP/IP version, the buffer process acts as a server, and starts by expecting two connections: one from the producer, and one from the consumer. It therefore calls accept() twice on startup.

The producer and consumer processes acts as clients, and establishes one connection each to the buffer using connect(). For more detailed information on the TCP/IP initialization sequence, see section 2.6.1.

²The directory service was not used because the RL version of the producer-consumer problem was implemented using a prototype of RL, in which the directory service were not implemented.
The buffer process can accept the producer and consumer connections in any order. When the producer and consumer processes have established an connection, they send an identifier string, identifying themselves as either a producer or a consumer.

6.1.3.2. CORBA

In the CORBA version, all three processes (producer, consumer, and buffer) acts as both clients and servers. The buffer needs to act as a server in order to receive requests to insert or remove data items. The producer and consumer processes needs to acts as servers because the buffer needs to be able to control their state depending on the state of the buffer itself, for example, when the buffer is full, the producer should be prevented from trying to insert additional items into the buffer.

The initialization sequence therefore becomes a hybrid of the initialization sequences listed for clients and servers in section 2.6.3. The following is an example for the buffer, the initialization sequence for the other processes is similar:

1. Initialize CORBA ORB.
2. Get a reference to the object adapter.
3. Get a reference to the naming service.
4. Activate the buffer implementation objects with the object adapter.
5. Register the buffer implementation object with the naming service.
6. Look up a reference to the producer implementation in the naming service.
7. Look up a reference to the consumer implementation in the naming service.
8. Activate the object adapter, and run the ORB.

This sequence will not be covered in detail here.

6.1.3.3. RL

In the RL version, the main() function of the producer, consumer, and buffer processes create one task each. These three tasks together implements the solution to the producer-consumer problem. After the task has been created, the main() function waits until the task has finished executing. The following example shows how the buffer task is created, the other tasks are created in a similar way:

```cpp
ConsumerTask ct; // Create the task
ct.Start();       // Start it running
ct.WaitFor();     // Wait for it to complete
```
The producer and consumer tasks each creates a reference to the buffer task, like in the following example:

```cpp
int ProducerTask::Run()
{
    TCPLocation loc("localhost", 3333, 2132);
    BufferRef Buf(this, loc);
    /* ... */
}
```

For the buffer tasks, all that is needed is to create a servant object and *activate* it:

```cpp
int BufferTask::Run() {
    /* bi is an object that implements the Buffer interface */
    BufferServant bs(this, &bi); // Create the servant
    bs.Activate(2132); // Activate it
}
```

### 6.1.4. Communication

#### 6.1.4.1. TCP/IP

In the TCP/IP version, it was necessary to invent a new “communication protocol” on top of TCP/IP. This communication protocol divides the data stream into messages and provides the means for the buffer to control the stream of data and requests from the producer and consumer processes. This protocol works as follows:

The producer process sends a data item to the buffer process using the connection established earlier, and then waits for an reply. This reply can be one of two things:

- *Ready.* The data was accepted, and the buffer process is ready to accept new items
- *Wait.* The data was accepted, but the buffer process cannot accept new items

In the latter case, the consumer will wait for a new *Ready* reply before proceeding.

The consumer process sends a *request* for a new data item to the buffer process, and waits for a reply. The reply contains the data item.

The buffer process receives new data items from the producer and new requests from the consumer and services these requests. It does not send a *Ready* reply to the producer before there is space available in the storage buffer, and it does not send a reply to the consumer until there is at least one item to retrieve in the storage buffer.
6.1.4.2. CORBA

In the CORBA version the producer, consumer and buffer processes are all servicing incoming remote calls. The producer and consumer processes also does additional work (i.e. producing and consuming data items), in addition to servicing incoming remote operations — something that is not straightforward in the CORBA client/server architecture, because the ORB run() method blocks, and does not return control to its caller. This problem is solved by spawning a new thread of control and executing the ORB run() method in that thread.

The producer runs its Produce() method in a loop as long as the process is awake, as shown below. The same goes for the consumer with its Consume() method. This means there are five processes in total, three run() processes and two processes that either produce items or remove items.

This is a code fragment from the main program of the producer:

```java
while(1){
    if(producer.awake == 1) // Spin lock
        producer.Produce(); // Produce a new item
}
```

The Produce() method just generates a random integer and transfers it to the buffer process by making a remote operation call, while Consume() merely requests from the buffer process that a integer should be removed.

As stated above the buffer have no need of running in a loop since it should be passive and only handle incoming remote requests through its run() process. However some sort of control must exist to make the appropriate action when the buffer is either full or empty. Any remove or insert request triggers a check to see if the consumer process or producer process needs to be blocked or unblocked. This is done by making a remote operation call to the producer or consumer’s SetStatus(int status), with the status as parameter. The status can be one of:

- **stop running** - The process should stop running. It will start looping in the spin lock shown in the code fragment above.
- **start running** - The process should start running, and will be released if it is currently looping in the spin lock.

6.1.4.3. RL

In the RL version, the producer and consumer processes runs in a loop, the producer process invoking the Insert operation, and the consumer process invoking the Remove
operation of the Buffer reference created in the initialization step. Actual code for the producer is shown below, the consumer is similar.

```c
while(1) {
    long int random_number = random();
    Buf.Insert(random_number);
}
```

The buffer process also runs in a loop, accepting operation calls from the producer and consumer.

```c
while(1) {
    if(bi.CanInsert() && bi.CanRemove()) {
        bs.Select("Insert:Remove");
        bs.Accept();
    } else if(bi.CanInsert()) {
        bs.Accept("Insert");
    } else {
        bs.Accept("Remove");
    }
}
```

Note that the operation calls are accepted selectively, the Insert operation is only accepted if the buffer has space for new data items, and the Remove operation is only accepted if the buffer has any data items.

### 6.1.5. Synchronization

In a correct solution to the producer-consumer problem, overflow (adding a new data item when the buffer storage is already full), and underflow (trying to remove a data item when the buffer storage is already empty) must be avoided. The three versions uses different mechanisms for avoiding underflow and overflow.

#### 6.1.5.1. TCP/IP

The TCP/IP version uses blocking read (i.e. `recv()`) calls for synchronization. Because the buffer process does not send a `Ready` reply to the producer before it is ready to accept a new item, and it does not send a reply with a data item to the consumer before it has any data items to send, it ensures that the producer and consumer processes block at the correct times.
6.1.5.2. CORBA

As explained earlier the synchronization is handled by having the buffer process set the status of the consumer and producer. The sender must wait until the entire remote call has been processed until it can proceed, which is why the spin lock that is used works, at least when dealing with only one producer and one consumer.

6.1.5.3. RL

The RL version uses rendezvous synchronization. As can be seen from the code example, the buffer process does not accept an operation if it cannot execute it (i.e., because the buffer’s data storage is full or empty), and the calling process simply waits until its request can be executed. This ensures that failures because of overflow and underflow of the buffer’s data storage are avoided.

6.2. Area Computation Problem

The producer/consumer problem described in the previous section is a simple test we used mainly to test the functionality of some interprocess communication methods. We have chosen another problem for our final test of the Rendezvous Library. This problem is not appropriate for giving comprehensive and accurate performance measurements, but it is enough to give us an indication on how RL compares to other communication libraries. Our main reason for choosing this test is that we have worked with problem earlier and setting up the test requires little work.

6.2.1. Problem description

The problem we have selected is the following:

For a three-dimensional mesh, consisting of elements, faces, and vertices, calculate the total surface area of the mesh. The total surface area is the sum of the areas of all faces that form the external boundary of the mesh. The faces containing internal boundaries are not counted in the total.

Figure 6.4 illustrates area calculation of a cube. The black edges line the faces that are included in the calculation, while the light gray edges line faces that are internal to the mesh and are not calculated towards the total.

The area of an individual triangular\textsuperscript{3} face can be calculated using the cross product. See figure 6.5. If we call the vertices that make up a triangular face \(v_1, v_2\) and \(v_3\), and we call the vector from \(v_1\) to \(v_2\) \(\vec{v_1v_2}\), and the vector from \(v_1\) to \(v_3\) \(\vec{v_1v_3}\), we can

\textsuperscript{3}Theoretically, the faces could be of any shape, but we have particularized the solution to triangular faces, since the mesh data that we had available contained triangular faces.
calculate the cross product $\overrightarrow{v_1} \times \overrightarrow{v_2}$. We then know that that vector's length is equal to the area of the parallelogram formed by the vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$. Since the area of the parallelogram is twice the area of the triangular face, the area of the face can be calculated by the following equation:

$$A = \frac{|\overrightarrow{v_1} \times \overrightarrow{v_2}|}{2}$$

The total surface area of a mesh can be calculated as follows: for all elements and all faces, if a face of an element does not have another element as a neighbour, its area is counted towards the total surface area of the mesh. The neighbour test ensures that the areas of faces that are internal to the mesh are not added to the total.

The pseudocode C++ algorithm below shows how the surface area can be calculated in a serial program.

```cpp
double area = 0.0;
for(int i=0; i<number_of_elements; i++) {
  for(int j=0; j<number_of_faces_per_elem; j++) {
    if(!elements[i].faces[j].hasNeighbour()) {
      area += elements[i].faces[j].area();
    }
  }
}
```

### 6.2.2. Common structure and shared elements

Regardless of which method of communication is used, the general structure in the program remains the same. All implementations use a master/worker implementation.
paradigm, in which there is one master process and several worker processes. The master process controls the calculation process while the worker processes carry out the actual calculations. Figure 6.6 shows a general flowchart for both the master and the worker processes.

As can be seen from the figure, the master process is responsible for

- reading the data,
- partitioning the data,
- sending data to worker processes,
- collecting results, and
- printing the final result.

The worker process

- receives data from the master
- calculates the surface area of all its elements’ faces, and
- sends this result to the master.
Figure 6.6: Flowchart of the Master and Worker processes. Dotted lines indicate activities that must happen simultaneously (i.e. sends and receives which correspond to each other).
6.2.3. Shared elements

Some program code could be shared between the implementations. This includes the procedures that

- read in the mesh data from file
- partitions the domain (i.e. the mesh) into subdomains
- prepares the data that will be sent to each processor
- calculates the area of a given subdomain

A procedure that prepares the data that will be sent to each processor is implemented. This procedure picks elements from the total data based on information from the domain partitioning procedure, and packs the element, face and vertex data into arrays, taking care not to duplicate face or vertex data (in case the same face or vertex is referenced two or more elements or faces).

6.2.4. MPI

MPI has been discussed in detail in section 2.6.2. Here we will only list the most important functions used in the MPI implementation of the area computation problem.

First of all we have the environmental procedures that needs to be called. These are MPI_Init(), which initializes the library, MPI_Comm_size(), which returns the number of processors and MPI_Comm_rank(), which returns the rank of the actual processor. This rank is the ID of the process. Each worker gets its own rank (master has rank 0), and the number of workers is equal to the number of processes that will run minus one (the master process). Last in the procedure, MPI_Finalize() is used to end the MPI communication session. No further MPI calls can be made after this call.

The master distributes the domain data to several workers that calculate a partial result and sends this back to the master. When sending and receiving we use the basic MPI_Send() and MPI_Recv() calls. The master task also need to wait for all worker tasks to finish receiving data before it can start collecting results. MPI has a call that does exactly this, MPI_Barrier().

One of the more troublesome issues with MPI was the packing of data to be sent. The elements, faces and vertices needs to be packed into arrays, and in the receiving end, the worker unpack the data much in the same way it was packed. The MPI calls that were used for this are MPI_Pack() and MPI_Unpack().

MPI includes all the functionality needed for communication in our program, and the library was generally easy to use. However, most MPI functions need to be called with many parameters that will be the same for all function calls in the program, and this

124
cluttered the program with unnecessary detail. In OOMPI, these parameters are given defaults and need not be specified.

6.2.5. OOMPI

Using OOMPI for communication is very similar to using MPI. The function calls are basically the same. This means that it is very easy to convert a program which uses MPI to use OOMPI.

The environmental functions in OOMPI belong to the `OOMPI_Intra_comm` class, initiated as `OOMPI_COMM_WORLD` in our program. These function include `Init()`, `Size()`, `Rank()` and `Set_error_action()`. The latter is specific for OOMPI and specifies how errors should be handled. The `OOMPI_COMM_WORLD` object also holds an array of `OOMPI_Port` objects for all processes in the cluster. The `OOMPI_Port` class has `Send()` and `Recv()` methods which are used for communication.

In OOMPI packed data messages are created a bit differently from MPI; a separate class is used for this, `OOMPI_Packed`. The method `Pack()` adds elements to the package to be sent. Sending in packages is an advantage since only one big transfer is needed. The package can be sent using a normal `Send()` call. The unpacking is done correspondingly with the `Unpack()` method.

6.2.6. CORBA

As previously stated, (see section 2.6.3), CORBA uses a client-server architecture, and for our problem, this architecture could fairly easily be mapped to the master/worker architecture that we were using. It was natural for the master to function as the server and the workers to function as clients. The server implements the following IDL interface:

```idl
interface master {
    typedef sequence<long> long_array;
    typedef sequence<double> double_array;
    void getSubDomain(in long pid,
        out long no elems, out long no faces, out long no verts,
        out long_array elesms, out long_array faces,
        out double_array vertices);
    void reportResult(in long pid, in double area);
};
```

All operations take the process ID of the worker task as the first argument (worker process ID’s range from 1 to total number of tasks minus one).

The workers call the `getSubDomain` operation to get the subdomain data. The data is returned as sequences of long integers and doubles. When the workers have completed
their calculation, they call the `reportResult` operation with the total area as an argument. When all workers have reported their result, the master prints out the total area and exits.

A separate class (called `master_i`) was created which implements the interface. In the master `main()` method an instance of this class is created and activated.

The simple mapping between the client/server and the master/worker architectures is only possible if communication only occurs between the master task and one worker task at a time. If the workers need to communicate with each other, or the master needs to communicate with several worker tasks simultaneously, it can be difficult to get the strict CORBA client/server architecture to fit in.

### 6.2.7. RL

Overall, the RL version is very similar to the CORBA version. But there are some differences.

In the RL version, the definitions of `long_array` and `double_array` in the IDL file was modified somewhat compared to the CORBA version:

```c
typedef long long_array[0];
typedef double double_array[0];
```

The two types are defined as variable-length arrays instead of as sequences, as in the CORBA version. This has two reasons:

- The current implementation of RL doesn’t support sequences
- Using “real” arrays are more efficient than the approach used by CORBA, which is to implement sequences as a class with an overloaded `[ ]` operator.

Two task classes, `WorkerTask` and `MasterTask`, were created. The program code that was in the `main()` function in the master program was moved to the `Run()` method of `WorkerTask`, and the program code that was in the `main()` function for the worker program was moved to the `Run()` method of `MasterTask`. The `main()` functions are left almost empty, the only thing they do is to instantiate a task object, start it, and wait for it to finish.

The interaction between worker task and master task is identical to that of the CORBA version. The `getSubDomain` operation is called by a worker task to transfer subdomain data from master to worker, and the `reportResult` operation is called by a worker to report the result of the computation back to the master.

For the master task, the main difference is that it is explicitly defined which operation calls should be accepted, with the following code fragment:
for(int i=1; i<numprocs; i++) {
    ms.Accept_getSubDomain();
}
for(int i=1; i<numprocs; i++) {
    ms.Accept_reportResult();
}

Here, ms is the servant object for the master interface (i.e. it is an object of the masterServant class). As can be seen, the getSubDomain operation is accepted as many times as there are available worker processes, and after that, the reportResult operation is accepted an equal number of times. In CORBA, one has no control over what operations are accepted.
7. Test Results

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1. Usability</td>
<td>128</td>
</tr>
<tr>
<td>7.2. Performance</td>
<td>129</td>
</tr>
<tr>
<td>7.2.1. About the Cluster</td>
<td>129</td>
</tr>
<tr>
<td>7.2.2. About the Software</td>
<td>129</td>
</tr>
<tr>
<td>7.2.3. Test files</td>
<td>130</td>
</tr>
<tr>
<td>7.2.4. Results</td>
<td>130</td>
</tr>
</tbody>
</table>

In this chapter an overview of the results collected with respect to usability and performance of the Rendezvous Library, relative to MPI, OOMPI and CORBA, is presented. Detailed results can be found in appendix A.

7.1. Usability

Learnability It is our opinion that CORBA has the lowest learnability of the four, while MPI and OOMPI has the highest. The Rendezvous Library is placed somewhere in between. We estimate that about 7 days are required to learn a sufficient amount of CORBA, approximately 5 days to learn RL, while about 2-3 days are required to be able to write a small program in MPI/OOMPI.

Functionality With regards to functionality, RL does not have the advanced functions that CORBA provide. However, unlike CORBA, RL is not tied to the client-server programming style. In RL, any programming style is applicable. Thus it is comparable to

<table>
<thead>
<tr>
<th></th>
<th>Lines of Code</th>
<th>Number of Classes</th>
<th>LOC/NOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOMPI</td>
<td>726</td>
<td>2</td>
<td>363</td>
</tr>
<tr>
<td>MPI</td>
<td>749</td>
<td>2</td>
<td>375</td>
</tr>
<tr>
<td>CORBA</td>
<td>944</td>
<td>3</td>
<td>315</td>
</tr>
<tr>
<td>RL</td>
<td>847</td>
<td>5</td>
<td>170</td>
</tr>
</tbody>
</table>

Table 7.1.: Usability results. Lines of code was generated using SLOCcount/38/ by David A. Wheeler.
MPI and OOMPI. MPI and OOMPI can be used in any setting, and with any programming style, and they also support for collective communication patterns. We therefore see MPI and OOMPI as the benchmark against which we measure. One problem with RL is the lack of support for collective communication patterns. This reduces the number of different settings that it can be used in. Thus we state that RL offers least functionality.

7.2. Performance

7.2.1. About the Cluster

ClustIS[27] is the computational cluster of the Division for Intelligent Systems (DIS) from the Computer and Information Science department, NTNU, Trondheim. ClustIS consists of 1 master node and 43 computational nodes with a total of 51 processors. The nodes are on a switched private network with 100MBit/sec between the nodes and the switch and a 1GBit/sec link between the master node and the switch. ClustIS is running Source Mage GNU/Linux, a source based GNU Linux distribution.

- Master: AMD Athlon XP 1700+ (1.46 GHz), 2GB RAM, 2*80GB IDE HD, 1*Gigabit Ethernet, 1*100MBit Ethernet
- Node type 1: (16 nodes: 1..16) AMD Athlon XP 1700+ (1.46 GHz), 2GB RAM, 1*40GB IDE HD, 1*100MBit Ethernet
- Node type 2: (12 nodes: 17..28) AMD Athlon XP 1700+ (1.46 GHz), 1GB RAM, 1*40GB IDE HD, 1*100MBit Ethernet
- Node type 3: (8 nodes: 29..36) AMD Athlon MP 1600+ (1.4 GHz), 1GB RAM, 1*18GB SCSI HD, 2*100MBit Ethernet
- Node type 4: (7 nodes: 37..43) AMD Dual Athlon MP 1600+ (1.4 GHz), 1GB RAM, 3*18GB SCSI HD, 2*100MBit Ethernet

We used the nodes 1-25 in our test, they are all AMD Athlon XP 1700+ (1.46 GHz) processors. Also we tried to monitor the test to ensure that as few other processes as possible other than our own was running on these nodes.

7.2.2. About the Software

Table 7.2 lists the software we used. In addition to the listed software, bash shell scripts and Perl scripts were used to perform the actual test runs automatically and to collect and compile results. Some of these scripts are listed in appendix B. GNU make was used to build the software.
Table 7.2.: Software used for performance evaluation

<table>
<thead>
<tr>
<th>Software type</th>
<th>Name</th>
<th>Version</th>
<th>Authors/vendors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System Kernel</td>
<td>Linux</td>
<td>2.4.17</td>
<td>The Linux Kernel Team</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC</td>
<td>2.95-3</td>
<td>The GNU Project</td>
</tr>
<tr>
<td>C library</td>
<td>glibc</td>
<td>2.2.5</td>
<td>The GNU Project</td>
</tr>
<tr>
<td>CORBA Implementation</td>
<td>omniORB</td>
<td>3.0.4</td>
<td>AT&amp;T Laboratories, Cambridge</td>
</tr>
<tr>
<td>MPI Implementation</td>
<td>MPICH</td>
<td>1.2.4</td>
<td>Mathematical and Computer Science Division, Argonne National Laboratory</td>
</tr>
<tr>
<td>OOMPI</td>
<td>OOMPI</td>
<td>1.0.3</td>
<td>Open Systems Laboratory, Indiana University</td>
</tr>
<tr>
<td>Partitioning library</td>
<td>METIS</td>
<td>4.0</td>
<td>George Karypis, University of Minnesota</td>
</tr>
<tr>
<td>RL Implementation</td>
<td>RL</td>
<td>0.4</td>
<td>K. Pedersen, K. Pedersen, C. E. Hauge, IDI, NTNU</td>
</tr>
</tbody>
</table>

Table 7.3.: Test files used for input

<table>
<thead>
<tr>
<th>Filename</th>
<th>No. verts</th>
<th>No. faces</th>
<th>No. elms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tiny.dat</td>
<td>1331</td>
<td>12600</td>
<td>6000</td>
<td>A cube</td>
</tr>
<tr>
<td>small.dat</td>
<td>2933</td>
<td>31818</td>
<td>15581</td>
<td>A sphere</td>
</tr>
</tbody>
</table>

7.2.3. Test files

Two different files were used as input to the area-calculating algorithm. Table 7.3 lists details on these files. These two files are text files describing a three-dimensional figure.

7.2.4. Results

In this section, results from the performance testing are presented in the form of bar charts. Tables containing the exact results can be found in the appendix. We have measured the average execution time from calculating the average of 5 runs on each number of processors. Speed-up and efficiency is calculated by using the formulae described in 3.2.5. The results will merely be presented here and discussed in the next chapter.

7.2.4.1. Average execution time

Figures 7.1 and 7.2 show the average execution time for test runs with the two input files tiny.dat and small.dat. For each figure, the average execution time for the serial version of the algorithm, run on one processor, is also shown.
Figure 7.1: Average execution time for the tiny.dat file. Average serial execution time was 4.79 seconds.

Figure 7.2: Average execution time for the small.dat file. Average serial execution time was 13.69 seconds.
7.2.4.2. Speed-up

Figures 7.3 and 7.4 show the speed-up (defined in section 3.2.5.2) for the tiny.dat and small.dat files.

7.2.4.3. Efficiency

Figures 7.5 and 7.6 show the efficiency (defined in section 3.2.5.3) for the tiny.dat and small.dat files.
Figure 7.4: Speed-up for the small.dat file

Figure 7.5: Efficiency for the tiny.dat file
Figure 7.6: Efficiency for the small.dat file
8. Evaluation

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1. Usability</td>
<td>135</td>
</tr>
<tr>
<td>8.2. Performance</td>
<td>136</td>
</tr>
<tr>
<td>8.2.1. Average Execution Time</td>
<td>136</td>
</tr>
<tr>
<td>8.2.2. Speed-Up and Efficiency</td>
<td>138</td>
</tr>
<tr>
<td>8.3. Validation</td>
<td>139</td>
</tr>
</tbody>
</table>

In this chapter, the usability and performance results are evaluated. Then the validity of the results is discussed.

8.1. Usability

We look at usability from the user’s perspective, i.e. from the perspective of the person or persons who have a particular problem that needs to be solved by using a distributed system. The typical user doesn’t want to worry about the inner workings of a given library or framework. What is important for the user is that it does its job with as little work as possible required on the user’s part, while at the same time providing satisfactory performance, functionality and accurate results.

In table 7.1 we have tried to assess the learning time of the different libraries. As one can see, MPI and OOMPI have the shortest learning time, followed by RL, and finally CORBA. We have based these approximations on the following facts:

- The MPI and OOMPI libraries have a lot of functions available, but only four or five is required in order to get started.
- OOMPI offers an object oriented programming style and provides default arguments for many method calls. This makes the code more readable.
- CORBA is a comprehensive specification, and in order to write applications that uses it the user is required to learn much of it.
- The CORBA programming style, while object-oriented, can be very complex. The header and code files created by the IDL compiler are usually practically unreadable.
• Implementing CORBA remote objects on the server side can be difficult for a user with no object oriented middleware experience.

• RL is simpler than CORBA. For example, to bind an object to a name in CORBA, approximately 20 lines of code are required. In RL this is done with one line of code. This also makes RL code somewhat easier to read than CORBA code.

OOMPI, closely followed by MPI comes out on top when we look at the number of lines of code required to solve the area computation problem. RL requires approximately 850 lines of code, while CORBA is the worst with about 950 lines of code.

RL has a larger number of classes than CORBA, MPI and OOMPI. This has to do with the architecture where everything is defined as tasks. Thus it is necessary, i.e., to specify a worker_task class and a master_task class in RL, unlike in the other 3 libraries.

With regards to functionality it is difficult to accurately assess how “good” or “bad” each library is. This will depend on the user, and what he wants to do. For example, CORBA offers a larger set of features, but if the user feels that it is frustrating to learn and use, the users perceived functionality of CORBA will be low. On the other hand, if the user needs the extra features, he will perceive CORBA’s functionality to be high.

8.2. Performance

Performance is determined by the performance metrics described in 3.2.5. The three metrics are: average execution time, speed-up and efficiency. The two latter will be calculated from the average execution time in relation to the execution time of the serial version, and the number of processors.

As will be explained later in the chapter an abnormal result of speed-up and efficiency will make these less important in our evaluation. This is unfortunate, but if forces us to make average execution time our main comparison factor when determining performance.

8.2.1. Average Execution Time

The average execution time is calculated from the average of the 5 runs we did on each number of processors. The execution time is our primary test result when it comes to performance. The other factors like speed-up and efficiency is calculated from this number. As we will notice later, the execution time of the serial version will seem long compared to the execution times of the distributed versions.

Noticable results are that RL has clearly the best performance in every case and that CORBA has little consistent results. The result from MPI and OOMPI are extremely similar as we expected, except that OOMPI is doing better than MPI sometimes. For all versions, the results when running on 3 processors seemed poor compared to the other
runs. This may be caused by a problem with METIS for that particular number of processors, but we are not sure of this.

8.2.1.1. Tiny.dat

When analysing the data, CORBA seems to have the slowest execution time until the run with 15 processors. From there on the execution time improves and it ends up as second best on the run that uses 25 processors. This is a very surprising result when compared to the bad execution time it has at 15-25 processors when using small.dat. At 10 processors the execution time increases compared to 7, but then it is decreasing again at 15. The result we get with 10 processors is very strange since it is the only time that the execution time increases. It is hard to say why the result is as it is, especially since all the 5 runs at 10 processors are virtually the same. Another strange result is when running with 5 processors. The reason for this is that 2 of the 5 runs have an extreme deviation.

MPI is doing better overall than CORBA, but the average execution time starts to increase from the test runs with 15 processors. As we expected, the OOMPI and MPI versions have very similar execution times. The differences are so small that we can not draw any conclusions as to which is faster.

RL has the best average execution time at every run, and it does not increase until 25 processors are used. In fact the result from 15 to 25 processors are almost the same. It is hard to determine how much faster RL is than the other libraries, but e.g. on 25 processors one can see a big difference between RL and the other libraries. At 5 processors the differences between the four libraries are the smallest. After that the differences just get bigger as the number of processors increase.

8.2.1.2. Small.dat

Small.dat is a much bigger file than tiny.dat. A bigger file means that more time will be spent on calculating results and, since more data must be transfered, communication will take longer. The ratio between increase time spent on communicating and time spent calculating should be approximately constant. Therefore the ratio in execution time between the libraries should be about the same.

Overall CORBA delivers the worst performance. However, the results are clearer than with tiny.dat. CORBA runs the slowest on 5-10 processors, but the difference is not that alarming. At 10 processors the execution time increases dramatically, and is the worst by far. The difference between 10 and 15 processors is alarming.

Overall, MPI is doing far better than CORBA, and average execution time does not increase until 20 processors are used. OOMPI is doing slightly better than MPI, but also in this test they are very similar.

RL is the best by far in this test. The difference between CORBA and RL is really
8.2.2. Speed-Up and Efficiency

As will be discussed in this section, we believe that the speed-up and efficiency numbers cannot be absolutely compared, because the serial version has a longer execution time than one should expect. We think this is caused by a deficiency in the algorithm which makes it inappropriate for absolute comparisons. The relative differences between the versions can still be used, though.

Also, as pointed out earlier, for small.dat we had some very strange results from all libraries when running at 3 processors. Since it happens in all versions, the fault lies in the common structure of the program. We have still included the results in the test results, but ignored it in the discussion.

8.2.2.1. Tiny.dat

RL has very good efficiency with any number of processors except for 25 processors. While the others have a good efficiency up to about 7 processors. After that the efficiency drop steadily. The estimate for “good” is purely a subjective metric, based on the fact that all libraries tended to peak at about 0.50-0.60.

8.2.2.2. Small.dat

The most noticeable result here is that RL performs better than theoretically possible. By “theoretically possible” we mean that the highest possible speed-up is equal to the number of processors it is running on. In other words, a parallel program running on 5 processors should have a maxium speed-up of 5. At 5 processors this means that 100% of the serial program can be parallelized. This does not include the communication overhead or any extra latency created from running a parallel version. If the number is higher this will result in an efficiency above 1.00, i.e. a program with a parallelization percentage of over 100%, and this is impossible.

We have not been able to find out why this occurs. The result from the area calculation is correct and the correct number of processors are used. Furthermore the structure of the program is almost identical to CORBA, and of course the algorithm is the same. This implies that the average execution time of the parallel version is not at fault. It is rather the serial version that has much higher running time that it should.

By re-testing both the parallel version as well as the serial version we observe that everything executes with basically the same results. The processors used are identical.
and with minimum load. A fact that lead us to believe that the serial version runs at a much higher execution time than it should is that it runs faster when running with more than 1 process. By running the small.dat on the serial version with 15 processes at one processor we got an execution time of about one third of the execution time for 1 process. We believe that this is close to what the serial version should run at, but have not found any evidence of why it behaves like this.

Since our algorithm has linear running time this is not the cause, as our first thought was. A non-linear running time would reduce the running time not only by dividing the work and running it in parallel, but also by simply dividing the work. The result of all this is that the average execution time of the parallel versions are valid and certainly comparable, but the execution time of the serial version is too low. This causes the speed-up and efficiency number we get to be too high. From a relative point of view they are still usable when comparing the libraries against each other.

We can only assume that this is true for the tiny.dat file as well, but since it is smaller this is not so noticeable. This is another point that make us believe it is the calculation itself that is the problem and not the communication.

When we compare the speed-up and efficiency results we can see that CORBA performs OK until 10 processors, but at 15 processors the efficiency drops by a third compared to 10 processors. It is the slowest before 15 processors, but the sudden drop is strange.

MPI is doing really well up to 7 processors, and then drops steadily from 0.92 at 7 processors to 0.26 at 25 processors. OOMPI is doing slightly better compared to MPI, which is surprising. What is expected, however, is the steady drop that is similar to MPI. These two libraries have almost the same results at the corresponding number of processors, and at some even the exact same efficiency.

RL has, as explained, an efficiency above 1.00 on some runs. This is an invalid result from a theoretical point of view. But since we believe that this is caused by a too high running time for the serial version, which affects the speed-up and efficiency of all versions equally, the results for the different libraries can be compared relative to each other. RL has a quite stable efficiency up until 15 processors, and then it drops slightly at 20 processors. At 25 processors the drop is more significant, but this may be due to a strange result from one of the 5 runs. While 4 of the 5 runs was at about 0.70, the fifth ended up at 1.38, which dragged the average execution time up to 0.79, thereby reducing the average efficiency.

8.3. Validation

Our tests support our claim that RL has a significantly better performance than CORBA. Somewhat surprisingly, our tests also indicated the the Rendezvous Library has better performance than MPI and OOMPI. As we stated in 3.2.2, we believed that the simplicity of RL was what would enabled us to provide superior performance. Why RL provided
better performance than MPI/OMPI is unknown to us at this time.

The results we got for speed-up and efficiency will be invalid since they compare the parallel execution time against the serial execution time. And since we believe the long serial execution time to be caused by a deficiency in the algorithm, the results are invalid, at least from an absolute point of view. As we first observed this from the test of small.dat our reasons can be found when discussing those results. We are still able to compare the results of the communication libraries, but the information we gained from these numbers are not much more than we got from simply evaluating the average execution time.

With regard to usability and learning time it is our opinion that RL would require less time to learn than CORBA. When designing the library it was intended that RL would not have the extensive functionality of MPI and CORBA. This, we believe, would result in a library that is easier to use.

We must however state that our results are not adequate as more than an indication of performance and learning time. As we state in section 8.2, the performance results have serious errors and lackings, but the relative results should be correct. With regards to usability we should ideally have run a large number of tests using volunteers, but this was outside the scope of our project due to time constraints.
9. Conclusion

Contents

<table>
<thead>
<tr>
<th>9.1. Conclusion</th>
<th>141</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2. Further Work</td>
<td>142</td>
</tr>
</tbody>
</table>

This chapter concludes the main part of the report. First our conclusion is presented, then suggestions on further work are given.

9.1. Conclusion

In our hypothesis we stated that we believed that RL will have a significantly better performance than CORBA, and based on our test this is true. The test had to be a small one, because most of our effort went into creating the library. The results are good enough to support our claim, even though they do not give reliable evidence. We also believed that RL would perform worse than OOMPI and MPI. Our tests indicate that this assumption was wrong. RL consistently performed better than both libraries.

The indications from our tests confirm our belief that RL is faster than CORBA. CORBA is a very complex specification and the developers have had to concentrate on implementing the complete standard rather than optimizing for speed. RL is much simpler and easier to optimize and not constrained by the CORBA language mappings, which are equally complex (and in many respects sub-optimal with respect to performance). Also RL is designed with performance as an important goal and the design has been chosen so that the performance is as good as possible. For example, the amount of data copying is kept to a minimum, and data are not transformed for transmission over the network if it is not necessary. From the test it is hard to judge why it actually was faster than CORBA, but we maintain these reasons as an explanation. Why it was significantly faster than OOMPI and MPI is unknown and we have nothing conclusive on this.

RL is more usable than CORBA because it is much simpler. Less initialization is required and less of its functionality must be learned before the library can be used. For our little test this is true, but as mentioned the test is too limited to be much more than a subjective statement about the usability in general.

We think RL is more usable than OOMPI because the rendezvous model is fundamentally better suited to modern object-oriented applications than the the input/output model.
used by the message-passing libraries. Therefore, the communication mechanism in RL offers a more natural style of programming than MPI or OOMPI, although it doesn’t offer as good control over the communication. For example, MPI offers collective communication and a choice between blocking and non-blocking communication, while RL only offers the rendezvous.

In order to get more conclusive results concerning performance and usability, a bigger more suited evaluation test must be created.

9.2. Further Work

We suggest the following for future work:

- The tests that we’ve run indicate that the RL has got very good performance. However, our tests are not extensive enough. We propose an expanded test suite to confirm our hypothesis. Examples of tests commonly used for benchmarking communication libraries are ring tests and ping-pong tests.

- Optimizing the Rendezvous Library. Further optimizations can be done with regards to performance. For example by reducing the amount of overhead sent. A suggestion might be to implement an adaptive header. Presently, the header is 16 bytes long. For short messages this can be reduced. An adaptive header could dynamically change the size of the header as needed.

- Expanding RL by adding more functionality. E.g. we can add support for transmitting some of the additional data types supported by CORBA. Examples are:
  - Structures
  - Unions
  - Sequences
  - Value types
  - Exceptions

- Expanding the Rendezvous Library by adding more error checking and error handling. In its current version RL contains a minimal amount of error checking and handling.
A. Results

In this chapter, complete results from the performance testing is presented. First, the average execution times, speed up and efficiency for the five runs on each number of processors is presented. After that, the raw results of the execution time measurements are included.

All execution times are in seconds.

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4.79</td>
<td>3.59</td>
<td>2.81</td>
<td>2.77</td>
<td>2.61</td>
</tr>
<tr>
<td>5</td>
<td>4.79</td>
<td>1.78</td>
<td>1.67</td>
<td>1.67</td>
<td>1.41</td>
</tr>
<tr>
<td>7</td>
<td>4.79</td>
<td>1.57</td>
<td>1.35</td>
<td>1.36</td>
<td>1.00</td>
</tr>
<tr>
<td>10</td>
<td>4.79</td>
<td>1.70</td>
<td>1.23</td>
<td>1.24</td>
<td>0.73</td>
</tr>
<tr>
<td>15</td>
<td>4.79</td>
<td>1.49</td>
<td>1.30</td>
<td>1.31</td>
<td>0.51</td>
</tr>
<tr>
<td>25</td>
<td>4.79</td>
<td>1.36</td>
<td>1.86</td>
<td>1.84</td>
<td>0.54</td>
</tr>
</tbody>
</table>

Table A.1.: Average execution time for tiny.dat.
<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>13.69</td>
<td>15.07</td>
<td>14.28</td>
<td>14.37</td>
<td>13.97</td>
</tr>
<tr>
<td>5</td>
<td>13.69</td>
<td>3.52</td>
<td>2.81</td>
<td>2.81</td>
<td>2.50</td>
</tr>
<tr>
<td>7</td>
<td>13.69</td>
<td>2.76</td>
<td>2.13</td>
<td>2.08</td>
<td>1.75</td>
</tr>
<tr>
<td>10</td>
<td>13.69</td>
<td>2.36</td>
<td>1.82</td>
<td>1.81</td>
<td>1.25</td>
</tr>
<tr>
<td>15</td>
<td>13.69</td>
<td>4.78</td>
<td>1.65</td>
<td>1.68</td>
<td>0.83</td>
</tr>
<tr>
<td>25</td>
<td>13.69</td>
<td>4.55</td>
<td>2.14</td>
<td>2.05</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table A.2: Average execution time for small.dat.

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>1.33</td>
<td>1.71</td>
<td>1.73</td>
<td>1.84</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>2.70</td>
<td>2.87</td>
<td>2.88</td>
<td>3.40</td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>3.05</td>
<td>3.55</td>
<td>3.54</td>
<td>4.81</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>2.81</td>
<td>3.88</td>
<td>3.86</td>
<td>6.60</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>3.22</td>
<td>3.68</td>
<td>3.67</td>
<td>9.44</td>
</tr>
<tr>
<td>25</td>
<td>1.00</td>
<td>3.52</td>
<td>2.58</td>
<td>2.60</td>
<td>8.88</td>
</tr>
</tbody>
</table>

Table A.3: Average speed-up for tiny.dat.

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.91</td>
<td>0.96</td>
<td>0.95</td>
<td>0.98</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>3.89</td>
<td>4.87</td>
<td>4.87</td>
<td>5.47</td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>4.96</td>
<td>6.42</td>
<td>6.57</td>
<td>7.82</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>5.79</td>
<td>7.51</td>
<td>7.57</td>
<td>10.95</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>2.86</td>
<td>8.30</td>
<td>8.13</td>
<td>16.54</td>
</tr>
<tr>
<td>25</td>
<td>1.00</td>
<td>3.01</td>
<td>6.39</td>
<td>6.68</td>
<td>17.29</td>
</tr>
</tbody>
</table>

Table A.4: Average speed-up for small.dat.

<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.44</td>
<td>0.57</td>
<td>0.58</td>
<td>0.61</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.54</td>
<td>0.57</td>
<td>0.58</td>
<td>0.68</td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>0.44</td>
<td>0.51</td>
<td>0.51</td>
<td>0.69</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>0.28</td>
<td>0.39</td>
<td>0.39</td>
<td>0.66</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>0.21</td>
<td>0.25</td>
<td>0.24</td>
<td>0.63</td>
</tr>
<tr>
<td>25</td>
<td>1.00</td>
<td>0.14</td>
<td>0.10</td>
<td>0.10</td>
<td>0.36</td>
</tr>
</tbody>
</table>

Table A.5: Average efficiency for tiny.dat.
<table>
<thead>
<tr>
<th>Version</th>
<th>serial</th>
<th>CORBA</th>
<th>MPI</th>
<th>OOMPI</th>
<th>RL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
<td>0.30</td>
<td>0.32</td>
<td>0.32</td>
<td>0.33</td>
</tr>
<tr>
<td>5</td>
<td>1.00</td>
<td>0.78</td>
<td>0.97</td>
<td>0.97</td>
<td>1.09</td>
</tr>
<tr>
<td>7</td>
<td>1.00</td>
<td>0.71</td>
<td>0.92</td>
<td>0.94</td>
<td>1.12</td>
</tr>
<tr>
<td>10</td>
<td>1.00</td>
<td>0.58</td>
<td>0.75</td>
<td>0.76</td>
<td>1.10</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>0.19</td>
<td>0.55</td>
<td>0.54</td>
<td>1.10</td>
</tr>
<tr>
<td>25</td>
<td>1.00</td>
<td>0.12</td>
<td>0.26</td>
<td>0.27</td>
<td>0.69</td>
</tr>
</tbody>
</table>

Table A.6: Average efficiency for small.dat.

<table>
<thead>
<tr>
<th>Version</th>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>4.81</td>
<td>4.85</td>
<td>4.71</td>
<td>4.89</td>
<td>4.71</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>3.58</td>
<td>3.59</td>
<td>3.59</td>
<td>3.59</td>
<td>3.62</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>2.37</td>
<td>1.38</td>
<td>1.39</td>
<td>1.39</td>
<td>2.36</td>
</tr>
<tr>
<td>CORBA</td>
<td>7</td>
<td>0.97</td>
<td>1.97</td>
<td>1.95</td>
<td>1.02</td>
<td>1.96</td>
</tr>
<tr>
<td>CORBA</td>
<td>10</td>
<td>1.74</td>
<td>1.72</td>
<td>1.67</td>
<td>1.70</td>
<td>1.69</td>
</tr>
<tr>
<td>CORBA</td>
<td>15</td>
<td>1.49</td>
<td>1.47</td>
<td>1.51</td>
<td>1.51</td>
<td>1.47</td>
</tr>
<tr>
<td>CORBA</td>
<td>20</td>
<td>1.39</td>
<td>1.42</td>
<td>1.39</td>
<td>1.41</td>
<td>1.39</td>
</tr>
<tr>
<td>CORBA</td>
<td>25</td>
<td>1.37</td>
<td>1.36</td>
<td>1.37</td>
<td>1.35</td>
<td>1.35</td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>2.88</td>
<td>2.75</td>
<td>2.84</td>
<td>2.77</td>
<td>2.81</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>1.68</td>
<td>1.65</td>
<td>1.66</td>
<td>1.67</td>
<td>1.70</td>
</tr>
<tr>
<td>MPI</td>
<td>7</td>
<td>1.34</td>
<td>1.34</td>
<td>1.37</td>
<td>1.35</td>
<td>1.35</td>
</tr>
<tr>
<td>MPI</td>
<td>10</td>
<td>1.21</td>
<td>1.31</td>
<td>1.25</td>
<td>1.19</td>
<td>1.21</td>
</tr>
<tr>
<td>MPI</td>
<td>15</td>
<td>1.33</td>
<td>1.28</td>
<td>1.29</td>
<td>1.42</td>
<td>1.19</td>
</tr>
<tr>
<td>MPI</td>
<td>20</td>
<td>1.51</td>
<td>1.47</td>
<td>1.49</td>
<td>1.45</td>
<td>1.47</td>
</tr>
<tr>
<td>MPI</td>
<td>25</td>
<td>1.89</td>
<td>1.79</td>
<td>1.82</td>
<td>1.83</td>
<td>1.96</td>
</tr>
<tr>
<td>OOMPI</td>
<td>3</td>
<td>2.80</td>
<td>2.78</td>
<td>2.75</td>
<td>2.79</td>
<td>2.74</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>1.71</td>
<td>1.61</td>
<td>1.64</td>
<td>1.70</td>
<td>1.67</td>
</tr>
<tr>
<td>OOMPI</td>
<td>7</td>
<td>1.37</td>
<td>1.34</td>
<td>1.37</td>
<td>1.38</td>
<td>1.32</td>
</tr>
<tr>
<td>OOMPI</td>
<td>10</td>
<td>1.31</td>
<td>1.19</td>
<td>1.13</td>
<td>1.27</td>
<td>1.31</td>
</tr>
<tr>
<td>OOMPI</td>
<td>15</td>
<td>1.33</td>
<td>1.23</td>
<td>1.31</td>
<td>1.24</td>
<td>1.43</td>
</tr>
<tr>
<td>OOMPI</td>
<td>20</td>
<td>1.52</td>
<td>1.33</td>
<td>1.41</td>
<td>1.45</td>
<td>1.45</td>
</tr>
<tr>
<td>OOMPI</td>
<td>25</td>
<td>1.82</td>
<td>1.91</td>
<td>1.87</td>
<td>1.83</td>
<td>1.79</td>
</tr>
<tr>
<td>RL</td>
<td>3</td>
<td>2.65</td>
<td>2.61</td>
<td>2.63</td>
<td>2.59</td>
<td>2.57</td>
</tr>
<tr>
<td>RL</td>
<td>5</td>
<td>1.42</td>
<td>1.37</td>
<td>1.40</td>
<td>1.43</td>
<td>1.42</td>
</tr>
<tr>
<td>RL</td>
<td>7</td>
<td>1.01</td>
<td>0.98</td>
<td>1.00</td>
<td>0.98</td>
<td>1.01</td>
</tr>
<tr>
<td>RL</td>
<td>10</td>
<td>0.78</td>
<td>0.73</td>
<td>0.70</td>
<td>0.71</td>
<td>0.71</td>
</tr>
<tr>
<td>RL</td>
<td>15</td>
<td>0.53</td>
<td>0.50</td>
<td>0.49</td>
<td>0.50</td>
<td>0.52</td>
</tr>
<tr>
<td>RL</td>
<td>20</td>
<td>0.50</td>
<td>0.46</td>
<td>0.47</td>
<td>0.52</td>
<td>0.46</td>
</tr>
<tr>
<td>RL</td>
<td>25</td>
<td>0.55</td>
<td>0.50</td>
<td>0.51</td>
<td>0.55</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table A.7: Raw execution time results for tiny.dat
<table>
<thead>
<tr>
<th>Version</th>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1</td>
<td>13.70</td>
<td>13.69</td>
<td>13.69</td>
<td>13.69</td>
<td>13.69</td>
</tr>
<tr>
<td>CORBA</td>
<td>3</td>
<td>15.14</td>
<td>15.17</td>
<td>14.95</td>
<td>15.13</td>
<td>14.98</td>
</tr>
<tr>
<td>CORBA</td>
<td>5</td>
<td>3.53</td>
<td>3.55</td>
<td>3.48</td>
<td>3.51</td>
<td>3.54</td>
</tr>
<tr>
<td>CORBA</td>
<td>7</td>
<td>2.75</td>
<td>2.77</td>
<td>2.75</td>
<td>2.77</td>
<td>2.76</td>
</tr>
<tr>
<td>CORBA</td>
<td>10</td>
<td>2.38</td>
<td>2.37</td>
<td>2.35</td>
<td>2.32</td>
<td>2.40</td>
</tr>
<tr>
<td>CORBA</td>
<td>15</td>
<td>4.79</td>
<td>4.84</td>
<td>4.75</td>
<td>4.74</td>
<td>4.78</td>
</tr>
<tr>
<td>CORBA</td>
<td>20</td>
<td>4.67</td>
<td>4.62</td>
<td>4.62</td>
<td>4.67</td>
<td>4.66</td>
</tr>
<tr>
<td>CORBA</td>
<td>25</td>
<td>4.45</td>
<td>4.58</td>
<td>4.55</td>
<td>4.59</td>
<td>4.57</td>
</tr>
<tr>
<td>MPI</td>
<td>3</td>
<td>14.28</td>
<td>14.29</td>
<td>14.29</td>
<td>14.27</td>
<td>14.27</td>
</tr>
<tr>
<td>MPI</td>
<td>5</td>
<td>2.82</td>
<td>2.85</td>
<td>2.83</td>
<td>2.77</td>
<td>2.79</td>
</tr>
<tr>
<td>MPI</td>
<td>7</td>
<td>2.13</td>
<td>2.20</td>
<td>2.15</td>
<td>2.11</td>
<td>2.07</td>
</tr>
<tr>
<td>MPI</td>
<td>10</td>
<td>1.84</td>
<td>1.83</td>
<td>1.84</td>
<td>1.76</td>
<td>1.85</td>
</tr>
<tr>
<td>MPI</td>
<td>15</td>
<td>1.71</td>
<td>1.60</td>
<td>1.64</td>
<td>1.57</td>
<td>1.73</td>
</tr>
<tr>
<td>MPI</td>
<td>20</td>
<td>1.79</td>
<td>1.79</td>
<td>1.69</td>
<td>1.76</td>
<td>1.80</td>
</tr>
<tr>
<td>MPI</td>
<td>25</td>
<td>2.18</td>
<td>2.15</td>
<td>2.15</td>
<td>2.15</td>
<td>2.09</td>
</tr>
<tr>
<td>OOMPI</td>
<td>5</td>
<td>2.84</td>
<td>2.83</td>
<td>2.78</td>
<td>2.79</td>
<td>2.81</td>
</tr>
<tr>
<td>OOMPI</td>
<td>7</td>
<td>2.14</td>
<td>2.11</td>
<td>2.06</td>
<td>2.05</td>
<td>2.06</td>
</tr>
<tr>
<td>OOMPI</td>
<td>10</td>
<td>1.77</td>
<td>1.84</td>
<td>1.76</td>
<td>1.85</td>
<td>1.82</td>
</tr>
<tr>
<td>OOMPI</td>
<td>15</td>
<td>1.72</td>
<td>1.67</td>
<td>1.67</td>
<td>1.63</td>
<td>1.73</td>
</tr>
<tr>
<td>OOMPI</td>
<td>20</td>
<td>1.79</td>
<td>1.68</td>
<td>1.78</td>
<td>1.75</td>
<td>1.76</td>
</tr>
<tr>
<td>OOMPI</td>
<td>25</td>
<td>2.07</td>
<td>2.18</td>
<td>1.97</td>
<td>2.05</td>
<td>1.98</td>
</tr>
<tr>
<td>RL</td>
<td>3</td>
<td>14.06</td>
<td>14.01</td>
<td>13.95</td>
<td>13.89</td>
<td>13.93</td>
</tr>
<tr>
<td>RL</td>
<td>5</td>
<td>2.55</td>
<td>2.45</td>
<td>2.47</td>
<td>2.51</td>
<td>2.53</td>
</tr>
<tr>
<td>RL</td>
<td>7</td>
<td>1.81</td>
<td>1.73</td>
<td>1.71</td>
<td>1.77</td>
<td>1.73</td>
</tr>
<tr>
<td>RL</td>
<td>10</td>
<td>1.31</td>
<td>1.20</td>
<td>1.25</td>
<td>1.23</td>
<td>1.26</td>
</tr>
<tr>
<td>RL</td>
<td>15</td>
<td>0.89</td>
<td>0.81</td>
<td>0.80</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>RL</td>
<td>20</td>
<td>0.74</td>
<td>0.68</td>
<td>0.66</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>RL</td>
<td>25</td>
<td>0.65</td>
<td>1.38</td>
<td>0.71</td>
<td>0.61</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table A.8.: Raw execution time results for small.dat
B. Scripts

Contents

B.1. test_execution_time.sh ........................................ 147
B.2. repeat.sh ......................................................... 149
  B.2.1. start/CORBA/small.10 ................................... 149
  B.2.2. start/RL/small.10 ........................................ 149
  B.2.3. collect_results.pl ....................................... 150

The following scripts were the most important scripts used in the performance measurements. These, and other scripts, are included on the CD which accompanies this report.

B.1. test_execution_time.sh

This is the main script used for the test run.

COMMAND="scripts/repeat.sh 5 time --portability"

NPLIST="3 5 7 10 15 20 25"

DATFILES="tiny small"

function proclist
{
  *
    echo -np $1
  ;;
  esac
}

function try
{
    echo $0
$0
}

for VERSION in CORBA RL; do
  echo "$VERSION version"
  for NAME in $DATFILES; do
    for PROC in $NPLIST; do
      echo $NAME - $PROC processors
      if [ "$VERSION" == "RL" ]; then
        rsh node01 rldirectory &
        sleep 2
        fi
      try $COMMAND sh start/$VERSION/$NAME.$PROC 2>results/exec_${VERSION}_${NAME}_${PROC}.sh
      sh scripts/cleanall.sh 2>/dev/null >/dev/null
      if [ "$VERSION" == "RL" ]; then
        rsh node01 killall rldirectory
        sleep 2
        fi
    done
  done
done

for VERSION in MPI OOMPI Framework; do
  echo "$VERSION version"
  for NAME in $DATFILES; do
    for PROC in $NPLIST; do
      echo $NAME - $PROC processors
      try rsh node04 $COMMAND /opt/mpich/bin/mpirun 'proclist $PROC' src/area_$VERSION $COMMAND
      sh scripts/cleanall.sh 2>/dev/null >/dev/null
    done
  done
  done

# Serial version
echo "Serial version"

for NAME in $DATFILES; do
  echo $NAME.dat
  try $COMMAND rsh node04 src/area_serial/task src/data/$NAME.dat 2>results/exec_serial
done

148
B.2. repeat.sh

A small script used to repeat a given command a number of times

#!/bin/sh

NUMTIMES=$1
shift
COUNTER=0

while [ $COUNTER -lt $NUMTIMES ]; do
  $$
  COUNTER='expr $COUNTER + 1'
done

B.2.1. start/CORBA/small.10

A number of different scripts were used to start all required processes in CORBA version of the area computation problem, one for each number of processors. As an example, the script for a run with 10 processors is presented.

#!/bin/sh
rsh node04 src/area_RL/master_task src/data/small.dat 10 &
rsh node05 src/area_RL/worker_task 1 &
rsh node06 src/area_RL/worker_task 2 &
rsh node07 src/area_RL/worker_task 3 &
rsh node08 src/area_RL/worker_task 4 &
rsh node09 src/area_RL/worker_task 5 &
rsh node10 src/area_RL/worker_task 6 &
rsh node11 src/area_RL/worker_task 7 &
rsh node12 src/area_RL/worker_task 8 &
rsh node13 src/area_RL/worker_task 9 &
wait

B.2.2. start/RL/small.10

As with the CORBA version, a number of scripts were used to start required processes. The script for 10 processors is included below.
#!/bin/bash
rsh node04 src/area_RL/master_task src/data/small.dat 10 &
rsh node05 src/area_RL/worker_task 1 &
rsh node06 src/area_RL/worker_task 2 &
rsh node07 src/area_RL/worker_task 3 &
rsh node08 src/area_RL/worker_task 4 &
rsh node09 src/area_RL/worker_task 5 &
rsh node10 src/area_RL/worker_task 6 &
rsh node11 src/area_RL/worker_task 7 &
rsh node12 src/area_RL/worker_task 8 &
rsh node13 src/area_RL/worker_task 9 &
wait

B.2.3. collect_results.pl

A small Perl script used to collect results from the results files produced by test_execution_time.sh and format them as a semicolon separated file, suitable for input into a spreadsheet or for further processing.

#!/usr/bin/perl

#This script produces a nice semicolon-separated file from the results data #files

use strict;

my $RESULTS_DIR = '/home/kjetip/results';

my @VERSIONS = ('serial', 'CORBA', 'MPI', 'OMPI', 'Framework', 'RL');

my @DATFILES = ('tiny', 'small');

my $PROCARRAY = [ 3, 5, 7, 10, 15, 20, 25 ];

my $PROCS = {
    'serial' => [ 1 ],
    'CORBA' => $PROCARRAY,
    'MPI' => $PROCARRAY,
    'OMPI' => $PROCARRAY,
    'Framework' => $PROCARRAY,
    'RL' => $PROCARRAY,
};
my $NUMRUNS = 5;
my ($datfile, $version, $proc, $errflag);

foreach $datfile (@DATFILES) {
    my $file = "$datfile.csv";
die "Error opening OUTPUT file $file" unless open(OUTPUT, ">$file");
print OUTPUT "Version:NumProcs:";

print OUTPUT join(?,?,1..$NUMRUNS),"\n";
foreach $version (@VERSIONS) {
    foreach @{$PROCS}->{$version} {
        print OUTPUT "$version;proc:";
        my $suffix;
        $suffix = "_proc" if $proc>1;

        my $file = "$RESULTS_DIR/exec_"."$version_"."$datfile.$suffix;
my @$results;
die "Error opening $file" unless open(FILE, $file);
$errflag = 0;
while(<FILE>) {
    my ($parameter,$value) = split;
    if($parameter eq 'real') {
        if( $errflag == 0 ) {
            push @$results,$value;
        } else {
            $errflag = 0;
            push @$results, '-';
        }
    } elsif($parameter eq 'user' || $parameter eq 'sys' || $parameter eq 'Command') {
        # do nothing
    } else {
        if($errflag == 0) {
            print STDERR "Warning: unexpected input in file $file - ignoring next result!\n";
            $errflag = 1;
        }
        print STDERR "Warning: offending input line: \"_\";
    }
}
close(FILE);
if(scalar @$results > $NUMRUNS) {
    print STDERR "Warning: Too many results, got ".(scalar @$results).", expected $NUMRUNS results;";
    splice @$results, $NUMRUNS;
}
}
if(scalar @results < $NUMRUNS) {
    print STDERR "Warning: Too few results, got ".(scalar @results).", expected 
    while(scalar @results < $NUMRUNS) {
        push @results, '-';
    }

    print OUTPUT join(';',@results),"\n";
}
}
}
C. Library API Reference

Contents

C.1. RL::CommItem Class Reference .................................. 156
    C.1.1. Detailed Description ................................... 156
    C.1.2. Constructor & Destructor Documentation .................. 156
    C.1.3. Member Data Documentation ............................ 156
C.2. RL::CommQueue Class Reference ................................. 158
    C.2.1. Detailed Description ................................... 158
    C.2.2. Member Function Documentation .......................... 158
C.3. RL::CommThread Class Reference ............................... 160
    C.3.1. Detailed Description ................................... 160
    C.3.2. Constructor & Destructor Documentation .................. 160
    C.3.3. Member Function Documentation .......................... 161
    C.3.4. Member Data Documentation ............................ 161
C.4. RL::Protocol::header Struct Reference ...................... 162
    C.4.1. Detailed Description ................................... 162
    C.4.2. Member Function Documentation .......................... 162
C.5. RL::ListenThread Class Reference ............................. 164
    C.5.1. Detailed Description ................................... 164
    C.5.2. Constructor & Destructor Documentation .................. 164
    C.5.3. Member Function Documentation .......................... 165
    C.5.4. Member Data Documentation ............................ 165
C.6. RL::Location Class Reference ................................. 166
    C.6.1. Detailed Description ................................... 166
    C.6.2. Member Function Documentation .......................... 167
C.7. RL::MainThread Class Reference ............................... 168
    C.7.1. Detailed Description ................................... 168
    C.7.2. Constructor & Destructor Documentation .................. 168
    C.7.3. Member Function Documentation .......................... 168
C.8. RL::ObjectRef Class Reference ................................ 170
    C.8.1. Detailed Description ................................... 170
    C.8.2. Constructor & Destructor Documentation .................. 170
This chapter contains reference information for the RL API. All classes of the runtime system, and their methods, are described. The documentation was extracted from comments in the source code by the doxygen[39] program.
Figure C.1.: Complete class diagram for the RL run-time system.
C.1. RL::CommItem Class Reference

#include <task.h>

Public Methods

- **CommItem** (int myobj=0, char *myop="", unsigned char *mydata=NULL, int mydataLen=0, TCPTransport *mytransport=NULL)

Public Attributes

- objid_t objid
- char * operation
- unsigned char * data
- int datalength
- TCPTransport * transport

Friends

- class CommQueue

C.1.1. Detailed Description

One item in the communication queue. Internal class, used by the CommQueue (p. 158) class.

C.1.2. Constructor & Destructor Documentation

C.1.2.1. RL::CommItem::CommItem (int myobj = 0, char * myop = "", unsigned char * mydata = NULL, int mydataLen = 0, TCPTransport * mytransport = NULL) [inline]

Creates a new CommItem (p. 156) and initializes the CommItem (p. 156) fields

C.1.3. Member Data Documentation

C.1.3.1. unsigned char* RL::CommItem::data

Operation data
C.1.3.2. int RL::CommItem::dataLength
Length of operation data

C.1.3.3. objid_t RL::CommItem::objid
Object ID for this item

C.1.3.4. char* RL::CommItem::operation
Operation name

C.1.3.5. TCPTransport* RL::CommItem::transport
Pointer to the transport object over which this Item was received or should be sent
The documentation for this class was generated from the following file:

- task.h
C.2. RL::CommQueue Class Reference

#include <task.h>

Public Methods

- CommQueue ()
- void Add (CommItem *newItem)
- CommItem *Remove ()
- CommItem *Peek ()
- CommItem *Search (objid_t o, char *op)
- CommItem *Remove (CommItem *item)

C.2.1. Detailed Description

Communication Queue. Holds operation requests. Internal class, used by the Task (p. 177) class.

C.2.2. Member Function Documentation

C.2.2.1. void RL::CommQueue::Add (CommItem *newItem) [inline]

Adds another item to the end of the queue.

C.2.2.2. CommItem* RL::CommQueue::Peek () [inline]

Returns the first item in the queue, or NULL if the queue is empty.

C.2.2.3. CommItem* RL::CommQueue::Remove (CommItem *item) [inline]

Removes a given item from the queue.

Parameters:

- item The item to remove.

C.2.2.4. CommItem* RL::CommQueue::Remove () [inline]

Removes an item from the beginning of the queue and returns it. Returns NULL if the queue is empty.
C.2.2.5. CommItem* RL::CommQueue::Search (objid_t o, char * op)
    [inline]

Searches the queue for the first item with matching object ID and operation name. Returns NULL if no items were found.

Parameters:
   o Object ID to search for
   op Operation name to search for. Set to NULL to search for all operations for a given Object ID.

The documentation for this class was generated from the following file:

- task.h
C.3. RL::CommThread Class Reference

#include <commthread.h>

Inheritance diagram for RL::CommThread:

```
RL::Thread

RL::CommThread
```

Public Methods

- CommThread (ListenThread *l, Task *t, TCPTransport *trans)
- ~CommThread ()
- void Run ()
- void ShutdownTransport ()

Protected Attributes

- ListenThread * listenthread
- Task * task
- bool closeflag
- TCPTransport * transport

C.3.1. Detailed Description

The CommThread (p. 160) thread will be started by the ListenThread (p. 164). One CommThread (p. 160) will be created for each connection that is accepted.

C.3.2. Constructor & Destructor Documentation

C.3.2.1. RL::CommThread::CommThread (ListenThread * l, Task * t, TCPTransport * trans)

CommThread (p. 160) constructor.

Parameters:

- l The ListenThread (p. 164) that created this thread
- t The task that this thread is a part of
- trans The TCP Transport (p. 188) over which data should be transferred
C.3.2.2. **RL::CommThread::~CommThread ()**

*CommThread* (p. 160) destructor. Deletes the *TCPTransport* (p. 188) object.

C.3.3. **Member Function Documentation**

C.3.3.1. **void RL::CommThread::Run () [virtual]**

The Run method of the *CommThread* (p. 160) reads entries from the *TCPTransport* (p. 188) connection given as a parameter in the constructor, and stores them in the task's entry list.

Reimplemented from *RL::Thread* (p. 193).

C.3.3.2. **void RL::CommThread::ShutdownTransport ()**

Calls Shutdown() on the thread’s associated *TCPTransport* (p. 188) object.

C.3.4. **Member Data Documentation**

C.3.4.1. **bool RL::CommThread::closedflag [protected]**

A flag that decides whether the communication should be shut down or not.

C.3.4.2. **ListenThread* RL::CommThread::*listenthread [protected]**

A link to the *ListenThread* (p. 164)

C.3.4.3. **Task* RL::CommThread::*task [protected]**

A link to the task that the *CommThread* (p. 160) is part of

C.3.4.4. **TCPTransport* RL::CommThread::*transport [protected]**

A communication item. Used to encapsulate data and transmit to the request queue.

The documentation for this class was generated from the following file:

- commthread.h
C.4. RL::Protocol::header Struct Reference

#include <protocol.h>

Public Attributes

- unsigned char proto_version
- unsigned char encoding
- unsigned char message_type
- unsigned char reserved
- int message_length
- int object_id
- int opname_length

C.4.1. Detailed Description

Message header for protocol version 0.

C.4.2. Member Data Documentation

C.4.2.1. unsigned char RL::Protocol::header::encoding

Encoding. This field can have one of two values:

- 0: little-endian
- 1: big-endian.

C.4.2.2. int RL::Protocol::header::message_length

Total length of message, in bytes, including the header.

C.4.2.3. unsigned char RL::Protocol::header::message_type

Message type. One of the following:

- 0: unknown/error
- 1: normal operation request
- 2: normal operation return
- 3: exceptional operation return (i.e. when an exception occurs)
C.4.2.4. int RL::Protocol::header::object_id

Object ID identifying the object to or from which the message is sent.

C.4.2.5. int RL::Protocol::header::opname_length

Length in bytes of the operation name, including a null-terminating character. The operation name immediately follows the header for operation requests. For other types of messages, this field can be set to 0.

C.4.2.6. unsigned char RL::Protocol::header::proto_version

Protocol version. Must be 0.

C.4.2.7. unsigned char RL::Protocol::header::reserved

Reserved. Must be 0.

The documentation for this struct was generated from the following file:

- protocol.h
C.5. RL::ListenThread Class Reference

#include <listenthread.h>

Inheritance diagram for RL::ListenThread:

```
RL::Thread
   ↓
RL::ListenThread
```

Public Methods

- `ListenThread(Task *)`
- `~ListenThread()`
- `void Run()`

Protected Attributes

- `Task * task`

Friends

- `classMainThread`

C.5.1. Detailed Description

The `ListenThread` (p. 164) will listen for incoming connections. When a connection is received, the `ListenThread` (p. 164) will start a `CommThread` (p. 160) to handle the communication.

C.5.2. Constructor & Destructor Documentation

C.5.2.1. RL::ListenThread::ListenThread (Task * t)

`ListenThread` (p. 164) constructor

Parameters:

- `t` The `Task` (p. 177) that created it
C.5.3. Member Function Documentation

C.5.3.1. void RL::ListenThread::Run () [virtual]

The run method. Runs in a loop accepting connections.
Reimplemented from RL::Thread (p. 193).

C.5.4. Member Data Documentation

C.5.4.1. Task* RL::ListenThread::task [protected]

A pointer to the task that started the thread
The documentation for this class was generated from the following file:

- listenthread.h
C.6.  RL::Location Class Reference

#include <location.h>

Inheritance diagram for RL::Location:

```
RL::Location

RL::TCPLocation
```

Public Methods

- Location()
- virtual ~Location()
- virtual Location * Clone ()=0
- virtual const char * ToString ()
- virtual objid_t GetObjectID ()=0

Static Public Methods

- Location * FromString (const char *uri)

Protected Attributes

- string locstring

C.6.1.  Detailed Description

A location is the "address" of a remote object. The Location (p.166) class is the base class of all other location classes. For each kind of network transport, a subclass of Location (p.166) can be defined. A location consists of two pieces of information: a transport-dependent network address, and an object ID.

See also:

TCP Location (p.182)
C.6.2. Member Function Documentation

C.6.2.1. virtual Location* RL::Location::Clone () [pure virtual]

This function creates an identical copy of this Location (p. 166) object and returns this object.
Implemented in RL::TCPLocation (p. 184).

C.6.2.2. Location* RL::Location::FromString (const char * uri) [static]

Converts the string into a Location (p. 166) object and returns that location object.

C.6.2.3. virtual objid_t RL::Location::GetObjectID () [pure virtual]

Returns the object ID of this location.
Implemented in RL::TCPLocation (p. 184).

C.6.2.4. virtual const char* RL::Location::ToString () [virtual]

This method converts the location to a stringified reference
The documentation for this class was generated from the following file:

- location.h
C.7. RL::MainThread Class Reference

#include <mainthread.h>

Inheritance diagram for RL::MainThread:

```
RL::Thread

RL::MainThread
```

Public Methods

- `MainThread (Task *t)`
- `~MainThread ()`
- `void Run ()`

Protected Attributes

- `Task * task`

C.7.1. Detailed Description

The `MainThread` (p. 168) will handle all the work specified by the user in the `Run` method.

C.7.2. Constructor & Destructor Documentation

C.7.2.1. RL::MainThread::MainThread (Task * t)

Constructor.

Parameters:
- `t` The task that created this thread

C.7.3. Member Function Documentation

C.7.3.1. void RL::MainThread::Run () [virtual]

The `Run()` (p. 168) method will call the task's `Run()` (p. 168) method.

168
Reimplemented from \texttt{RL::Thread} (p. 193).

The documentation for this class was generated from the following file:

- \texttt{mainthread.h}
C.8. RL::ObjectRef Class Reference

#include <objectref.h>

Public Methods

• ObjectRef (Task *t, Location *l)
• ObjectRef (Task *t, const char *s)
• ~ObjectRef ()
• Location *GetLocation ()
• Task *GetTask ()
• const char *ToString ()

Protected Attributes

• Task *thetask
• Location *thelocation
• string locstring

C.8.1. Detailed Description

The RL::ObjectRef class is the base class for all object references.

Author:

Kenneth Pedersen

C.8.2. Constructor & Destructor Documentation

C.8.2.1. RL::ObjectRef::ObjectRef (Task * t, Location * l)

Creates a new object reference

Parameters:

• t Pointer to the calling task
• l A location object containing the location of the remote object. A copy is made of this location object, and the original can be deleted if desired.

C.8.3. Member Function Documentation

C.8.3.1. Location* RL::ObjectRef::GetLocation () [inline]

Returns a pointer to this object’s Location (p.166) object.
C.8.3.2. Task* RL::ObjectRef::GetTask () [inline]

Returns a pointer to this object’s Task (p. 177) object.

C.8.3.3. const char* RL::ObjectRef::ToString ()

Converts this reference to a stringified object reference, and returns it.

C.8.4. Member Data Documentation

C.8.4.1. Location* RL::ObjectRef::thelocation [protected]

Pointer to a location object containing the location of the remote object

C.8.4.2. Task* RL::ObjectRef::thetask [protected]

Pointer to the calling task

The documentation for this class was generated from the following file:

- objectref.h
C.9. **RL::ObjectServant Class Reference**

```
#include <objectservant.h>
```

**Public Methods**

- `ObjectServant (Task *t)`
- `~ObjectServant ()`
- `void Deactivate ()`
- `void Activate (RL::objid_t objid=0)`
- `virtual int Invoke (const char *opname, unsigned char *data, int datalen, unsigned char *outdata, int &outdatalen)=0`
- `bool Select (const char *ops, long int seconds=-1, long int milliseconds=-1)`
- `void Accept (const char *opname=NULL)`
- `bool TryAccept (const char *opname=NULL)`
- `const char * ToString ()`

**Protected Attributes**

- `objid_t myid`
- `Task * task`
- `string locstring`

**C.9.1. Detailed Description**

Base class for Object servants

**C.9.2. Constructor & Destructor Documentation**

**C.9.2.1. RL::ObjectServant::ObjectServant (Task * t)**

Creates a new object servant

**Parameters:**

- `t` Pointer to the calling task.

**C.9.3. Member Function Documentation**

**C.9.3.1. void RL::ObjectServant::Accept (const char * opname = NULL)**

Accepts an operation call. Blocks until a matching operation call has been received.
Parameters:
  \textit{opname} The operation call to accept, or NULL to accept any operation.

C.9.3.2. \texttt{void RL::ObjectServant::Activate (RL::objid \_t objid = 0)}

Activates this object servant

Parameters:
  \textit{objid} The object ID of the object that is to be served. If this parameter is zero, an
  unused, random, object ID is used.

C.9.3.3. \texttt{void RL::ObjectServant::Deactivate ()}

Deactivates this object servant

C.9.3.4. \texttt{virtual int RL::ObjectServant::Invoke (const char \* opname, unsigned
  char \* data, int datalen, unsigned char \*\& outdata, int \& outdatalen)}
  \hfill [pure virtual]

Invokes an operation.

Parameters:
  \textit{opname} The operation to invoke
  \textit{data} The input/inout parameter data
  \textit{datalen} Length of data, in bytes
  \textit{outdata} Return value, and the output/inout parameter data.
  \textit{outdatalen} Length out outdata, in bytes

Returns:
  0 if OK, or other values if an error has occurred

C.9.3.5. \texttt{bool RL::ObjectServant::Select (const char \* ops, long int seconds = -1,}
  \hfill long int \textit{milliseconds} = -1)

Waits for an incoming operation call to occur.

Parameters:
  \textit{ops} The list of operation calls to wait for. Each operation call is separated by colons.
  For example to wait for both Insert and Remove, call Select("Insert:Remove").
  Set to NULL to wait for any operation
  \textit{seconds} Number of seconds to wait. Set to -1 to wait forever.
milliseconds Number of milliseconds to wait in addition to the number of seconds listed as the previous parameter.

**Returns:**
true if an operation call was received, false if the call timed out.

**C.9.3.6. const char* RL::ObjectServant::ToString ()**
Converts this object servant to a stringified object reference, and return it.

**C.9.3.7. bool RL::ObjectServant::TryAccept (const char * opname = NULL)**
Accept an operation call if it has already been received. If no matching operation call has been received, return false.

**Parameters:**
opname The operation call to accept, or NULL to accept any operation

**Returns:**
true if an operation was accepted, false if it was not.

**C.9.4. Member Data Documentation**

**C.9.4.1. objid_t RL::ObjectServant::myid [protected]**
Pointer to the servant’s task. The servant will use this task’s communication system for communication.
The documentation for this class was generated from the following file:

- objectservant.h
C.10. RL::Runtime Class Reference

#include <runtime.h>

Static Public Methods

- void Init (int argc, char **argv[])
- Runtime * GetInstance ()
- objid_t RegisterObjectID (objid_t o)

Protected Methods

- Runtime ()
- ~Runtime ()

Static Protected Attributes

- Runtime instance
- map< objid_t, bool > objidmap

C.10.1. Detailed Description

RL Runtime (p.175) services. The Runtime (p.175) object contains functionality that is used by components of the RL library. Only one Runtime (p.175) object can be created in an application. This object is initialized by the static Init method.

C.10.2. Constructor & Destructor Documentation

C.10.2.1. RL::Runtime::Runtime () [protected]

Creates a new Runtime (p.175) object

C.10.3. Member Function Documentation

C.10.3.1. Runtime* RL::Runtime::GetInstance () [static]

Returns a pointer to the Runtime (p.175) object.
C.10.3.2. void RL::Runtime::Init (int & argc, char ** argv[]) [static]

Initializes the runtime system. The argc and argv parameters are the command-line arguments passed to the application. Any RL-specific arguments on the command line are processed by this method and stripped from the command-line arguments.

C.10.3.3. objid_t RL::Runtime::RegisterObjectID (objid_t o) [static]

Register a new objid ID. Object ID’s must be registered to avoid the case where two objects have the same ID.

Parameters:
   o Object ID of the Object. If this parameter is 0, a new random Object ID is allocated.

Returns:
   The Object ID that was actually registered.

C.10.4. Member Data Documentation

C.10.4.1. Runtime RL::Runtime::instance [static, protected]

The one and only Runtime (p. 175) instance

C.10.4.2. map<objid_t, bool> RL::Runtime::objidmap [static, protected]

Registered object ID’s

The documentation for this class was generated from the following file:

   • runtime.h
C.11.  RL::Task Class Reference

#include <task.h>

Public Methods

- Task()
- virtual ~Task()
- void Start(Location *l=NULL)
- void WaitFor()
- virtual void Run()
- void RLRoutedEventArgs(struct timeval *t)
- CommQueue *RLRequestQueue()
- void RLRoutedEventArgsLock()
- void RLRoutedEventArgsUnlock()
- CommQueue *RLReturnQueue()
- void RLRoutedEventArgsCondWait()
- int RLRoutedEventArgsTimedWait(const struct timespec *time)
- void RLRoutedEventArgsBroadcast()
- const char *ToString()

Protected Methods

- void RLCgpULock()
- void RLCgpUUnlock()

Protected Attributes

- int task_retval
- TCP_Pool pool
- TCP_Transport transport
- CommQueue request_queue
- CommQueue return_queue
- TCP_Location *loc
- ListenThread *listen
- MainThread *main
- list<CommThread *> commlist
- pthread_mutex_t commlist_mutex
- pthread_mutex_t request_queue_mutex
- pthread_cond_t request_queue_cond
- pthread_mutex_t request_queue_cond_mutex
Friends

- class ListenThread
- class MainThread
- class CommThread

C.11.1. Detailed Description

The Task (p. 177) class is the base class of all RL (p. ??) tasks.

C.11.2. Constructor & Destructor Documentation

C.11.2.1. RL::Task::Task ()

Creates a new task.

C.11.3. Member Function Documentation

C.11.3.1. void RL::Task::RLCommListLock () [protected]

Locks the comm list

C.11.3.2. void RL::Task::RLCommListUnlock () [protected]

Unlocks the comm list

C.11.3.3. void RL::Task::RLReadToQueue (struct timeval * t)

RL internal method. Listens to incoming connections and reads an incoming request to the request queue. If no requests have been accepted within the specified time interval, this method returns.

Parameters:

- t Pointer to timeval struct specifying the maximum time to wait. Set to NULL to wait indefinately.

C.11.3.4. CommQueue* RL::Task::RLRequestQueue () [inline]

Locks the request queue, and returns pointer to the request queue.
C.11.3.5.  void RL::Task::RRequestQueueBroadcast ()

Broadcast the condition that a new entry was added to the list of incoming calls

C.11.3.6.  void RL::Task::RRequestQueueCondWait ()

Waits for the request queue condition to be signaled

C.11.3.7.  void RL::Task::RRequestQueueLock ()

Locks the request queue

C.11.3.8.  int RL::Task::RRequestQueueTimedWait (const struct timespec *
            time)

A timed wait for the request queue condition to be signaled. It returns an errorcode to
indicate whether it has timed out, been interrupted, or 0 if successful.

Parameters:
        time Pointer to a timespec structure that specifies how long it should wait.

C.11.3.9.  void RL::Task::RRequestQueueUnlock ()

Unlocks the request queue

C.11.3.10. CommQueue* RL::Task::RReturnQueue () [inline]

NOT IN USE: RL internal method. Returns pointer to the return queue.

C.11.3.11. virtual void RL::Task::Run () [virtual]

The Task (p. 177)’s main thread. This method must be implemented by all subclasses.

C.11.3.12. void RL::Task::Start (Location * l = NULL)

Starts this task. Sets up communication and runs the task’s Run method.

C.11.3.13. const char* RL::Task::ToString ()

Returns a partial stringified object reference for the task
C.11.3.14. void RL::Task::WaitFor ()
Waits for this thread to finish executing.

C.11.4. Member Data Documentation

C.11.4.1. list<CommThread *> RL::Task::commList [protected]
The list that contains the pointers to each communication thread

C.11.4.2. pthread_mutex_t RL::Task::commList_mutex [protected]
A variable for mutual exclusion on the commlist

C.11.4.3. ListenThreads RL::Task::listen [protected]
Pointers to the listening and main threads

C.11.4.4. TCPLocation RL::Task::loc [protected]
A location object specifying a TCP/IP address

C.11.4.5. TCPPool RL::Task::pool [protected]
A pool of all connections accepted by this task

C.11.4.6. CommQueue RL::Task::request_queue [protected]
A queue in which incoming requests are stored before they are processed.

C.11.4.7. pthread_cond_t RL::Task::request_queue_cond [protected]
A condition variable used in unison with the mutex variable below

C.11.4.8. pthread_mutex_t RL::Task::request_queue_cond_mutex [protected]
A mutex for the condition variable

180
C.11.4.9.  pthread_mutex_t RL::Task::request_queue_mutex  [protected]

A variable for mutual exclusion on the request_queue

C.11.4.10.  CommQueue RL::Task::return_queue  [protected]

A queue in which outgoing responses are stored before they are send.

C.11.4.11.  int RL::Task::task_retval  [protected]

The return value of the Run method.

C.11.4.12.  TCPTransport RL::Task::transport  [protected]

A transport object used for listening for incoming connections
The documentation for this class was generated from the following file:

- task.h
C.12. **RL::TCPLocation Class Reference**

```c
#include <tcplocation.h>
```

Inheritance diagram for RL::TCPLocation:

```
RL::Location
  ↓
RL::TCPLocation
```

**Public Methods**

- **TCPLocation** ()
- **TCPLocation** (sockaddr_in *addr, objid_t objid=0)
- **TCPLocation** (u_int32_t addr, u_int16_t port=0, objid_t objid=0)
- **TCPLocation** (const char *hostname, unsigned short int port=0, objid_t o=0)
- ~TCPLocation ()
- objid_t GetObjectID ()
- Location * Clone ()

**Static Public Methods**

- Location * FromStringHandler (const char *uri)

**Protected Methods**

- void updateLocationString ()

**Protected Attributes**

- sockaddr_in address
- objid_t objid

**Friends**

- class TCPTransport
C.12.1. Detailed Description

The `TCPLocation` (p.182) class is for specifying TCP/IP addresses. Two pieces of information is stored, a TCP/IP address, and an object ID.

Author:
Kenneth Pedersen

C.12.2. Constructor & Destructor Documentation

C.12.2.1. `RL::TCPLocation::TCPLocation()`

Creates a new TCP location, and zero out address and object ID.

C.12.2.2. `RL::TCPLocation::TCPLocation(sockaddr_in * addr, objid_t objid = 0)`

Creates a new TCP location with the specified address and object ID.

Parameters:
- `addr` Pointer to sockaddr_in structure containing address, in network byte order.
- `objid` The object ID.

C.12.2.3. `RL::TCPLocation::TCPLocation(u_int32_t addr, u_int16_t port = 0, objid_t objid = 0)`

Creates a new TCP location with the specified address and object ID.

Parameters:
- `addr` IP address in host byte order.
- `port` Port number in host byte order.
- `objid` Object ID.

C.12.2.4. `RL::TCPLocation::TCPLocation(const char * hostname, unsigned short int port = 0, objid_t o = 0)`

Creates a new TCP location with the specified address and object ID.

Parameters:
- `hostname` Host name. Corresponding IP address will be looked up using `gethostbyname_r`.
- `port` Port number in host byte order.
- `o` Object ID.
C.12.3. Member Function Documentation

C.12.3.1. Location* RL::TCPLocation::Clone () [virtual]
Clones this TCPLocation (p.182) object and returns the clone.
Implements RL::Location (p.167).

C.12.3.2. Location* RL::TCPLocation::FromStringHandler (const char * uri) [static]
Converts a "rltcp://" URI to a TCPLocation (p.182) object, and returns it. Called by Location::FromString (p.167)

C.12.3.3. objid_t RL::TCPLocation::GetObjectID () [inline, virtual]
Return this location's object ID.
Implements RL::Location (p.167).

C.12.4. Member Data Documentation

C.12.4.1. sockaddr_in RL::TCPLocation::address [protected]
TCP/IP address of this object. See ip manual page for description of sockaddr_in structures.

C.12.4.2. objid_t RL::TCPLocation::objid [protected]
Object ID
The documentation for this class was generated from the following file:

- tcplocation.h
C.13. **RL::TCPPool Class Reference**

```cpp
#include <tcppool.h>
```

**Public Methods**

- `TCPool ()`
- `~TCPool ()`
- `void Add (TCPTransport *)`
- `void Remove (TCPTransport *)`
- `TCPTransport * GetFirst ()`
- `TCPTransport * GetNext ()`
- `TCPTransport * GetByFD (socket_t fd)`
- `TCPTransport * GetCurrent ()`
- `bool Selected (TCPTransport *)`
- `void Select (struct timeval *timeout=NULL)`

**Static Public Methods**

- `void Select (TCPool *readset, TCPool *writeset=NULL, TCPool *exceptset=NULL, struct timeval *timeout=NULL)`

C.13.1. **Detailed Description**

This class implements a collection of `TCPTransport` (p. 188) objects.

**Author:**

Kenneth Pedersen

C.13.2. **Constructor & Destructor Documentation**

C.13.2.1. **RL::TCPPool::TCPPool ()**

Creates a new `TCPPool` (p. 185) object.

C.13.3. **Member Function Documentation**

C.13.3.1. **void RL::TCPPool::Add (TCPTransport *)**

Adds the given transport to the pool.
C.13.3.2. **TCPTransport RL::TCPPool::GetByFD (socket_t fd)**

Returns the transport object with the given file descriptor, or NULL if there is no such transport object.

**Parameters:**

*fd* The file descriptor.

C.13.3.3. **TCPTransport RL::TCPPool::GetCurrent ()**

Gets the current transport object.

C.13.3.4. **TCPTransport RL::TCPPool::GetFirst ()**

Returns the first transport object in the pool, or NULL if the pool is empty.

C.13.3.5. **TCPTransport RL::TCPPool::GetNext ()**

Returns the next transport object in the pool, or NULL if there are no transport objects.

C.13.3.6. **void RL::TCPPool::Remove (TCPTransport *)**

Removes the given transport from the pool.

C.13.3.7. **void RL::TCPPool::Select (TCPPool * readset, TCPPool * writeset = NULL, TCPPool * exceptset = NULL, struct timeval * timeout = NULL) [static]**

Using the Select method, the transport object in a pool can be watched to see if any are ready for reading or writing, or if any exceptions has occured. The Select method waits the given time interval for any of the transport objects to change status. After the Select method returns, the Selected method can be used to find out which transport objects changed status.

**Parameters:**

*readset* The transports in this pool are watched to see if a Read operation will not block.

*writeset* The transports in this pool are watched to see if a Write operation will not block.

*exceptset* The transports in this pool are watched to see if any exceptions have occurred.
timeout The Select method will at most wait the time period indicated by this parameter. Setting this parameter to NULL will cause the Select method to wait indefinitely.

See also:
Selected (p. 187), the select(2) manual page

C.13.3.8. void RL::TCPPool::Select (struct timeval * timeout = NULL) [inline]
Selects the transport object in the pool both for reading and writing. See the static Select method.

C.13.3.9. bool RL::TCPPool::Selected (TCPTransport *)
Returns true if the given transport object is selected (by a previous call to one of the Select methods), false if it isn’t.
The documentation for this class was generated from the following file:

* tcpool.h
C.14. **RL::TCPTTransport Class Reference**

```cpp
#include <tcptransport.h>
```

**Public Types**

- `enum ConnectionState { UNBOUND, LISTENING, CONNECTED, SHUTDOWN }`

**Public Methods**

- `TCPTTransport ()`
- `~TCPTTransport ()`
- `bool Listen (Location *loc=NULL, int backlog=10)`
- `TCPTTransport * Accept (bool doblock=true)`
- `bool Connect (Location *loc)`
- `int Read (void *buf, int len)`
- `int Write (void *buf, int len)`
- `bool Close ()`
- `bool Shutdown (int how=SHUT_RDWR)`
- `ConnectionState GetState ()`
- `Location * GetPeerAddress ()`
- `TCPTTransport * Clone ()`
- `bool SetBlocking (bool blocking)`
- `bool Select ()`
- `socket_t SocketFD ()`

**Public Attributes**

- `ConnectionState state`

**Protected Methods**

- `socket_t GetSocket ()`

**Protected Attributes**

- `socket_t tcpsocket`
- `TCP.Location peeraddr`
Friends

- class TCPool

C.14.1. Detailed Description

TCP/IP transport mechanism. This class is a thin wrapper around native TCP/IP system calls, like socket(), bind(), accept(), recv(), send(), etc.

Author:
Kenneth Pedersen

C.14.2. Member Enumeration Documentation

C.14.2.1. enum RL::TCPTransport::ConnectionState

The transport object can be in one of three states:

- UNBOUND: Not bound to a local port, and not listening.
- LISTENING: Listening for incoming connections.
- CONNECTED: Connected to a remote port. The Read and Write methods can be used to send and receive data over the connection.

C.14.3. Constructor & Destructor Documentation

C.14.3.1. RL::TCPTransport::TCPTransport ()

Creates a new TCPTransport (p. 188) object.

C.14.4. Member Function Documentation

C.14.4.1. TCPTransport* RL::TCPTransport::Accept (bool doblock = true)

Accepts an incoming connection, and returns a new TCPTransport (p. 188) object representing the connected socket.

Parameters:
- doblock If true, this method will block and wait for a new incoming connection. If false, it will return immediately if no incoming connections are available.

C.14.4.2. TCPTransport* RL::TCPTransport::Clone ()

Creates an identical copy of this TCPTransport (p. 188) object and returns the copy.
C.14.4.3. bool RL::TCPTransport::Close ()

Closes the connection.

C.14.4.4. bool RL::TCPTransport::Connect (Location * loc)

Connects to a remote socket indicated by the given address.

Parameters:

loc Location (p. 166) to connect to. Must be castable to a TCPLocation (p. 182) pointer. The object ID of this location is ignored.

C.14.4.5. Location* RL::TCPTransport::GetPeerAddress ()

Gets the peer address (the location of the socket to which we are connected)

C.14.4.6. socket_t RL::TCPTransport::GetSocket () [protected]

Creates a new socket using the socket system call

See also:

the socket(2) manual page

C.14.4.7. ConnectionState RL::TCPTransport::GetState () [inline]

Get connection state. This can be one of UNBOUND, LISTENING, and CONNECTED.

See also:

ConnectionState (p. 189)

C.14.4.8. bool RL::TCPTransport::Listen (Location * loc = NULL, int backlog = 10)

Listens for incoming connections on the port indicated by the given location object. The Location (p. 166) pointer must be castable to a TCPLocation (p. 182) pointer, and must point to a local TCP/IP address. The object ID of this location is ignored.

Parameters:

loc The location of the local port.

backlog Maximum number of connections to wait for before starting to refuse connections.
See also:
the bind(2) and listen(2) system calls

C.14.4.9.  int RL::TCPTransport::Read (void * buf, int len)

Reads from the connection. This method attempts to read up to len bytes from the
socket into the buffer starting at buf. It will block until the requested number of bytes
are read, the socket is closed, or an error occurs.

Parameters:
  buf   The buffer to read into.
  len   The number of bytest to read.

C.14.4.10. bool RL::TCPTransport::Select ()

A new select method intended for use in the multithreaded version of the library. The
method does only check for incoming data, not for outgoing or exceptions.

C.14.4.11. bool RL::TCPTransport::SetBlocking (bool blocking)

Sets the socket to blocking or non-blocking mode.

Parameters:
  blocking true to set the socket to blocking, false to set it to non-blocking.

C.14.4.12. bool RL::TCPTransport::Shutdown (int how = SHUT_RDWR)

Shutdown the connection

C.14.4.13. socket_t RL::TCPTransport::SocketFD () [inline]

Internal method. Returns the socket’s file descriptor.

C.14.4.14. int RL::TCPTransport::Write (void * buf, int len)

Writes to the connection. This method attempts to write up to len bytes to the socket
into the buffer starting at buf. It will block until the requested number of bytes are
written, the socket is closed, or an error occurs.

Parameters:
  buf   The buffer to write from.
  len   The number of bytest to write.
C.14.5. Member Data Documentation

C.14.5.1. TCP Location RL::TCPTransport::peeraddr [protected]

Address of the peer (the "other end" of the socket)

C.14.5.2. ConnectionState RL::TCPTransport::state

The connection state

C.14.5.3. socket_t RL::TCPTransport::tcpsocket [protected]

Socket file descriptor

The documentation for this class was generated from the following file:

- tcptransport.h
C.15. RL::Thread Class Reference

#include &lt;thread.h&gt;

Inheritance diagram for RL::Thread::

Public Methods

- Thread ()
- virtual ~Thread ()
- void Start ()
- virtual void Run ()
- void WaitFor ()

Protected Attributes

- pthread_t thread_info
- int status

C.15.1. Detailed Description

This is the base class for all the threads in the library. Threads are implemented using posix threads.

C.15.2. Constructor & Destructor Documentation

C.15.2.1. RL::Thread::Thread ()

Constructor

C.15.3. Member Function Documentation

C.15.3.1. virtual void RL::Thread::Run () [virtual]

The Run() (p. 193) method must be redefined in inherited classes
Reimplemented in **RL::CommThread** (p. 161), **RL::ListenThread** (p. 165), and **RL::MainThread** (p. 168).

**C.15.3.2. void RL::Thread::Start ()**

Starts the thread and executes the **Run()** (p. 193) method in the thread. The start method returns immediately, and does not wait for the threads to finish.

**C.15.3.3. void RL::Thread::WaitFor ()**

Waits until this thread has finished executing its **Run()** (p. 193) method.

**C.15.4. Member Data Documentation**

**C.15.4.1. pthread_t RL::Thread::thread_info [protected]**

pthread_t is a datastructure that describes a given thread. **Thread** (p. 193) information is stored here.

The documentation for this class was generated from the following file:

- thread.h
D. IDL Syntax and Semantics

Contents

D.1. Overview ................................................. 196
D.2. Lexical Conventions ................................. 197
  D.2.1. Tokens ........................................... 197
  D.2.2. Comments ........................................ 199
  D.2.3. Identifiers ....................................... 199
  D.2.4. Keywords ......................................... 199
  D.2.5. Literals .......................................... 200
D.3. Preprocessing .......................................... 201
D.4. RL IDL Grammar ....................................... 201
D.5. RL IDL Specification ................................. 202
D.6. Module Declaration ................................. 202
D.7. Interface Declaration ................................ 202
  D.7.1. Interface Header .................................. 202
  D.7.2. Interface Inheritance Specification .......... 202
  D.7.3. Interface Body .................................... 203
  D.7.4. Forward Declaration ................................ 203
  D.7.5. Interface Inheritance .......................... 203
  D.7.6. Local Interface .................................. 203
D.8. Value Declaration ..................................... 204
  D.8.1. Regular Value Type .............................. 204
  D.8.2. Boxed Value Type ................................ 205
  D.8.3. Abstract Value Type ............................ 205
  D.8.4. Value Forward Declaration ..................... 205
  D.8.5. Valuetype Inheritance ........................... 205
D.9. Constant declaration ................................. 205
D.10. Type Declaration ...................................... 205
  D.10.1. Basic Types ..................................... 206
  D.10.2. Constructed Types .............................. 206
  D.10.3. Template Types ................................ 207
  D.10.4. Complex Declarator ............................ 208
The Interface Definition Language (IDL) is the language used to describe the interfaces that client tasks can call and server tasks provide. An interface definition written in IDL completely defines the interface and fully specifies each operation’s parameters. An IDL interface provides the information needed to develop client tasks that use the interface’s operations.

The Rendezvous Library Interface Definition Language (RL IDL) is a subset of the OMG Interface Definition Language (OMG IDL) [1, chapter 3]. This chapter describes the RL IDL semantics, and lists the differences between RL IDL and OMG IDL. Except as noted in this chapter, the IDL specification for OMG IDL is also valid for RL IDL. All language mappings and language mapping issues defined for OMG IDL are not valid for RL IDL, which has its own language mappings.

D.1. Overview

Task are not written in RL IDL, which is a purely descriptive language, but in languages for which mapping from RL IDL concepts have been defined. Currently, there is only one such language: C++.

Table D.1 shows a mapping between some RL IDL concepts, and the corresponding Ada and Concurrent C concepts.

A source file containing interface specifications written in RL IDL shuld have an “.idl” extension, but this is not a requirement.

For each IDL language feature mentioned in this chapter, an implementation status is listed. This shows if the language feature described has been implemented in the current version of RL, as shown in table D.2. Note that even if a feature is not implemented, its syntax and semantics may be defined. This is meant as a suggestion for future implementations.
<table>
<thead>
<tr>
<th>RL IDL</th>
<th>Ada</th>
<th>Concurrent C</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Task</td>
<td>Process</td>
<td>*</td>
</tr>
<tr>
<td>Module</td>
<td>Task</td>
<td>Process</td>
<td>*</td>
</tr>
<tr>
<td>Operation</td>
<td>Extended rendezvous</td>
<td>Transaction</td>
<td></td>
</tr>
<tr>
<td>Operation declaration</td>
<td>Entry declaration</td>
<td>Transaction declaration</td>
<td></td>
</tr>
<tr>
<td>Operation invocation</td>
<td>Entry call</td>
<td>Transaction call</td>
<td></td>
</tr>
<tr>
<td>Structures</td>
<td>Records</td>
<td>Structures</td>
<td></td>
</tr>
</tbody>
</table>

Table D.1: IDL, Ada and Concurrent C language concepts.

Notes:
*) This is an approximation; tasks/processes implement interfaces and modules

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Implemented. This language feature is implemented in full.</td>
</tr>
<tr>
<td>P</td>
<td>Partially implemented. This language feature has been implemented partially. The text described what has been implemented.</td>
</tr>
<tr>
<td>N</td>
<td>Not implemented. This OMG IDL language feature is not implemented in RL IDL.</td>
</tr>
</tbody>
</table>

Table D.2: IDL language feature status

Table D.3 is an overview of OMG IDL language features, and their implementation status in RL IDL. Table D.4 lists which types are supported by RL IDL.

### D.2. Lexical Conventions

This section presents the lexical conventions of RL IDL. It defines tokens in an RL IDL specification and describes comments, identifiers keywords and literals - integer, character, and floating point constants and string literals.

RL IDL, as OMG IDL, uses the ASCII character set, except for string literals, and character literals, which use the ISO Latin-1 (8859.1) character set.

#### D.2.1. Tokens

There are five kinds of tokens: identifiers, keywords, literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collective, “white space”), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.
<table>
<thead>
<tr>
<th>Language feature</th>
<th>St.</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Comments</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>/* Comments */</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Identifiers</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Escaped Identifiers</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td></td>
<td>see table D.4</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Modules</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Interface Inheritance</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Local Interfaces</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Value Types</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Typedef</td>
<td>I</td>
<td>truncatable keyword has status N</td>
</tr>
<tr>
<td>Type Declarations</td>
<td></td>
<td>see table D.4</td>
</tr>
<tr>
<td>Exceptions</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Context Expressions</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Attributes</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>CORBA Module</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Table D.3.: OMG IDL language features

<table>
<thead>
<tr>
<th>Type</th>
<th>St.</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Types</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Character Types</td>
<td>I</td>
<td>wchar type is partially implemented</td>
</tr>
<tr>
<td>Floating-point Types</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>String Types</td>
<td>I</td>
<td>wstring type is partially implemented</td>
</tr>
<tr>
<td>Fixed-point Type</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Boolean Type</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Octet Type</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Any Type</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Structures</td>
<td>N</td>
<td>recursive structures not allowed</td>
</tr>
<tr>
<td>Discriminated Unions</td>
<td>N</td>
<td>recursive unions not allowed</td>
</tr>
<tr>
<td>Enumerations</td>
<td>N</td>
<td>recursive enumerations not allowed</td>
</tr>
<tr>
<td>Sequences</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td>P</td>
<td>see description for arrays</td>
</tr>
<tr>
<td>Native Types</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Table D.4.: OMG IDL Types
D.2.2. Comments

**Status:** I The characters // start a comment, which terminates at the end of the line on which they occur.
The characters /* start a comment, which terminates with the characters */. These comments cannot be nested.
Example:

```c
/*
   this is a comment
   /* which ends after this word */
   and not after this word
*/
```

In the example above, the text “and not after this word */” would not be treated as a comment.

D.2.3. Identifiers

**Status:** I An identifier is an arbitrarily long sequence of ASCII alphabetic, digit and underscore ("_") characters. The first character must be an ASCII alphabetic character, and all characters are significant.
As in OMG IDL, when comparing two identifiers to see if they collide, upper- and lower-case letters are treated as the same letter. An identifier must however be spelled identically (e.g. with respect to case) throughout a specification. (In the current release of the RL IDL compiler, it is not checked if identifiers collide.)
There is only one namespace for RL IDL identifiers in each scope, and each identifier cannot be used to refer to different things.

D.2.3.1. Escaped identifiers

**Status:** N Escaped identifiers are not supported.

D.2.4. Keywords

The table D.5 lists the keywords supported by RL IDL. If a keyword is not implemented in full, the implementation status is listed in parenthesis.
<table>
<thead>
<tr>
<th>abstract (N)</th>
<th>exception (N)</th>
<th>octet</th>
<th>struct (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute (N)</td>
<td>enum (N)</td>
<td>oneway (N)</td>
<td>supports (N)</td>
</tr>
<tr>
<td>boolean</td>
<td>factory (N)</td>
<td>out</td>
<td>switch (N)</td>
</tr>
<tr>
<td>case (N)</td>
<td>FALSE</td>
<td>private (N)</td>
<td>TRUE</td>
</tr>
<tr>
<td>char</td>
<td>float</td>
<td>public (N)</td>
<td>typedef</td>
</tr>
<tr>
<td>const (N)</td>
<td>in</td>
<td>raises (N)</td>
<td>unsigned</td>
</tr>
<tr>
<td>double</td>
<td>inout</td>
<td>readonly (N)</td>
<td>union (N)</td>
</tr>
<tr>
<td>context (N)</td>
<td>interface</td>
<td>sequence (N)</td>
<td>valuetype (N)</td>
</tr>
<tr>
<td>default (N)</td>
<td>long</td>
<td>short</td>
<td>void</td>
</tr>
<tr>
<td>double</td>
<td>module</td>
<td>string (P)</td>
<td></td>
</tr>
</tbody>
</table>

Table D.5.: Keywords supported by RL IDL

D.2.5. Literals

This section describes the following literals:

- Integer
- Character (8-bit and 16-bit)
- Floating-point
- String (8-bit and 16-bit)
- Fixed-point

D.2.5.1. Integer Literals

Status: I An integer literal is a sequence of digits, optionally prefixed with 0 (digit zero), 0x, or 0X. If the literal begins with 0, it is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. If the literal begins with 0x, or 0X, it is taken to be an hexadecimal integer (base sixteen). The hexadecimal digits include a or A through f or F with decimal values ten through fifteen, respectively. If the literal does not begin with a digit zero, it is taken to be a decimal (base ten) integer.

D.2.5.2. Character Literals

Status: I An character literal is one or more characters enclosed in single quotes, as in 'x'. 8-bit character literals have type char. They follow the rules given for OMG IDL 8-bit character literals.

Status: P Wide (16-bit) character literals have type wchar, and follow the rules given for OMG IDL 16-bit character literals.
D.2.5.3. Floating-point Literals

Status: I  A floating point literal consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent. It follows the rules given for OMG IDL floating-point literals.

D.2.5.4. String Literals

Status: I  A string literal is a sequence of characters, with the exception of the character with numeric value 0, surrounded by double quotes as in “...”. It follows the rules given for OMG IDL string literals.
8-bit string literals have type string.

Status: P  Wide (16-bit) string literals have type wstring.

D.2.5.5. Fixed-Point Literals

Status: N  Fixed-point literals are not supported.

D.3. Preprocessing

Status: N  Like OMG IDL, RL IDL is preprocessed according to the specification of the preprocessor in “International Organization for Standardization. 1998. ISO/IEC 14882 Standard for the C++ Programming Language, Geneva”.

Lines beginning with # (also called directives) communicate with this preprocessor. White space may appear before the #. The primary use of preprocessing directives is to include other source files with the #include preprocessor directive.

There are no RL IDL-specific preprocessor pragmas. The IDL compiler always generates code for files included in the source file by default, but can be instructed not to generate code for included files.

D.4. RL IDL Grammar

For the most part, RL IDL grammar closely follows OMG IDL grammar rules.
D.5. **RL IDL Specification**

An RL IDL specification consists of one or more type definitions, constant definitions, exception definitions, or module definitions.

D.6. **Module Declaration**

**Status:** 1  The module construct is used to scope RL IDL identifiers.

D.7. **Interface Declaration**

**Status:** 1  An interface declaration consists of an interface header, which includes an interface inheritance specification, and an interface body.

D.7.1. **Interface Header**

**Status:** 1  The interface header names the interface and specifies its inheritance (if any). It consists of the following three elements:

1. An optional `abstract` (Status: N) keyword, that specifies that the given interface is an abstract interface.

2. The interface name, which must be preceded by the keyword `interface`, and consists of an identifier that names the interface.

3. An optional inheritance specification. The inheritance specification is described in the next section.

The identifier that names an interface defines a legal type name.

**Status:** N  Local interfaces are not supported. Abstract interfaces are not supported.

D.7.2. **Interface Inheritance Specification**

**Status:** N  The interface inheritance specification lists the interfaces (if any) from which this interface is derived.
D.7.3. Interface Body

**Status: I** The interface body contains the following kinds of declarations:

- Constant declarations (status N), which specify the constants that the interface exports.
- Type declarations (status I), which specify the type definitions that the interface exports.
- Exception declarations (status N), which specify the exception structures that the interface exports.
- Attribute declarations (status N), which specify the associated attributes exported by the interface.
- Operation declarations (status I), which specify the operations that the interface exports and the format of each, including operation name, the type of data returned, the types of all parameters of an operation and legal exceptions that may be returned as a result of an invocation (status N).

Empty interfaces are permitted (that is, those containing no declarations).

D.7.4. Forward Declaration

**Status: I** A forward declaration declares the name of an interface without defining it. This permits the definition of interfaces that refer to each other. Multiple forward declarations of the same interface name are legal. It is illegal to inherit from a forward-declared interface whose definition has not yet been seen.

**Status: N** Local interfaces are not supported, and thus the `local` keyword cannot be used with forward declarations.

D.7.5. Interface Inheritance

**Status: N** An interface can be derived from another interface, which is then called a base interface of the derived interface. Interface inheritance follows the rules and semantics for OMG IDL.

D.7.6. Local Interface

**Status: N** Local interfaces are not supported.
D.8. Value Declaration

Status: N  There are several kinds of value type declarations: “regular” value types, boxed value types, abstract value types, and forward declarations.

D.8.1. Regular Value Type

The regular value type consists of a value header, that names the value type and specifies its inheritance, and a value body that defines its state members, initializers and operations.

D.8.1.1. Value Header

The value header consists of two elements:

1. The value’s type name, prefixed by the \texttt{valuetype} keyword.
2. An optional value inheritance specification.

Value types that use custom marshaling are not supported, and the \texttt{custom} keyword is not supported.

D.8.1.2. Value Element

A value can contain all the elements that an interface can as well as the definition of state members, and initializers for that state.

D.8.1.3. Value Inheritance Specification

The value inheritance specification lists the value types this value types is inherited from, and any interfaces this interface support.

The \texttt{truncatable} keyword is not supported.

D.8.1.4. State Members

Each state member defines an element of the state, which is marshaled and sent to the receiver when the value type is passed as a parameter. A state member is either public or private. The annotation directs the language mapping to hide or expose the different parts of the state to the clients of the value type. The private part of the state is only accessible to the implementation code and the marshaling routines.
D.8.1.5. Initializers

Initializers / constructors for non abstract value types can be defined. These look like
local operation signatures, except that they are prefixed with the keyword factory, have
no return type, and must use only in parameters.

D.8.2. Boxed Value Type

The boxed value type provides a short-hand way of defining a value type with no inher-
itage or operations and a single state member.

D.8.3. Abstract Value Type

Value types may also be abstract. They are called abstract because an abstract value
type may no be instantiated. No state members or initializers may be specified, however,
local operations may be specified.

D.8.4. Value Forward Declaration

A forward declaration declares the name of a value type without defining it. Boxed value
types cannot be forward declared. It is illegal to inherit from a forward-declared value
type whose definition has not yet been seen.

D.8.5. Valuetype Inheritance

The terminology, name scope rules, and name collision rules for valuetypes are identical
to those for interfaces. Valuetype inheritance follows the rules defined for OMG IDL.

D.9. Constant declaration

Status: N The syntax and semantics for a constant declaration follows the rules defined
for OMG IDL, except for the fact that wide character, wide string, and fixed-point
constants are not allowed.

D.10. Type Declaration

Status: I RL IDL, like OMG IDL, provides constructs for naming data types; that is,
it provides C language-like declarations that associate an identifier with a type using the
typedef keyword.
D.10.1. Basic Types

**Status: I** The following basic types are allowed:

- **float** Single-precision floating-point number
- **double** Double-precision floating-point number
- **long double** Double-extended-precision floating-point number
- **short** 16-bit signed integer (range \(-2^{15} \ldots 2^{15} - 1\))
- **long** 32-bit signed integer (range \(-2^{31} \ldots 2^{31} - 1\))
- **long** 64-bit signed integer (range \(-2^{15} \ldots 2^{15} - 1\))
- **unsigned short** 16-bit unsigned integer (range \(0 \ldots 2^{16} - 1\))
- **unsigned long** 32-bit unsigned integer (range \(0 \ldots 2^{32} - 1\))
- **unsigned long long** 64-bit unsigned integer (range \(0 \ldots 2^{64} - 1\))
- **char** 8-bit quantity used to encode characters
- **wchar** 16-bit quantity used to represent wide characters (status P)
- **boolean** a boolean data item can only have the values TRUE and FALSE
- **octet** 8-bit quantity that is guaranteed not to undergo any conversion when transmitted

**Status: N** The following type is not supported

- **any** Any types can be used to express any OMG IDL type.

D.10.2. Constructed Types

**Structs**, **unions** and **enums** are the constructed types.

D.10.2.1. Structures

**Status: N** Structures are like value types, except that they cannot contain operations and initializers, and cannot inherit.

They follow the OMG IDL rules for structures.
D.10.2.2. Discriminated Unions

**Status: N** The discriminated unions are a cross between the `union` and `switch` statements. They follow the OMG IDL rules for unions.

D.10.2.3. Constructed Recursive Types and Forward Declarations

**Status: N** Constructed recursive types are not allowed.

D.10.2.4. Enumerations

**Status: N** Enumerated types consist of ordered lists of identifiers. They follow the OMG IDL rules for enumerations.

D.10.3. Template Types

The template types are sequences, strings and wstrings.

D.10.3.1. Sequences

**Status: N** A sequence is a one-dimensional array with two characteristics: a maximum size (which is fixed at compile time, and may be “infinite”), and a length, which is determined at run time.

D.10.3.2. Strings

**Status: I** The `string` type consists of all possible 8-bit quantities except null. A `string` is similar to a sequence of `char`.

D.10.3.3. Wstrings

**Status: P** The `wstring` type consists of all possible 16-bit quantities except null. A `wstring` is similar to a sequence of `wchar`. The support for marshaling and unmarshaling `wstrings` is incomplete.

D.10.3.4. Fixed Type

**Status: N** The `fixed` data type is not supported.
D.10.4. Complex Declarator

Complex declarators include arrays. Native types, as defined in OMG IDL, are not supported.

D.10.4.1. Arrays

**Status: P**  The arrays defined for OMG IDL are multidimensional, fixed-size arrays. All dimensions of the array are explicitly sized, and is fixed at compile time. When an array is passed as a parameter in an operation invocation, all elements of the array are transmitted.

*Extension:* Arrays in RL IDL are variable-sized.

*Partially Implemented:* Arrays cannot be used as return types, and multidimensional arrays and arrays of strings are not supported.

D.10.4.2. Native Types

**Status: N**  Native types are not supported.

D.10.4.3. Deprecated Anonymous Types

**Status: N**  The anonymous types that are deprecated in OMG IDL are not allowed in RL IDL.

D.11. Exception Declaration

**Status: N**  Exception declaration permit the declaration of struct-like data structures, which may be returned to indicate that an exceptional condition has occurred during the performance of a request. The exception declaration follows the syntax rules of OMG IDL.

There is a set of standard system exceptions which correspond to standard run-time errors. These exceptions differ from the ones defined for CORBA.

D.12. Operation Declaration

**Status: I**  Operation declarations in RL IDL are similar to C function declarators, and follows the syntax rules of OMG IDL.

An operation declaration consists of:
• An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked (the one\text{way} attribute, which has status N).

• The type of the operation’s return result, the type may be any type that can be defined in OMG IDL. Operations that do not return a result must specify the void type.

• An identifier that names the operation in the scope of the interface in which it is defined.

• A parameter list that specifies zero or more parameters for the operation.

• An optional raises expression (status N) that indicates which exception may be raised as a result of the operation invocation.

• An optional context expression (status N) that indicates which elements of the request context may be consulted by the method that implements the operation.

D.12.1. Operation Attribute

\textbf{Status: N} The operation attribute specifies the invocation semantics the communication service must provide.

Currently there is only one operation attribute defined, the one\text{way} attribute. The one\text{way} attribute causes the operation invocation to return immediately, without waiting for the invocation request to be transmitted. The operation cannot contain any output parameters and must specify a void return type. It cannot contain a raises expression, but the invocation may raise standard system exceptions.

The invocation semantics are at-most-once if the one\text{way} attribute is specified. Without this attribute, the invocation semantics are at-most-once if an exception is raised, and exactly-once if the operation invocation returns successfully.

D.12.2. Parameter Declarations

\textbf{Status: I} Parameter declarations in RL IDL follows the syntax for OMG IDL, except that the wide string type is not allowed as a parameter type. in, inout, and out parameters are supported. It is expected that an implementation will not attempt to modify an in parameter. If an exception is raised as a result of an invocation, the values of the return result and any out and inout parameters are undefined.

D.12.3. Raises Expressions

\textbf{Status: N} A raises expression specifies which exceptions may be raised as a result of an invocation of the operation. The syntax for its specification follows OMG IDL rules.
In addition to the set of exception listed in the \texttt{raises} expression (if any), an operation invocation may raise one of the standard system exceptions.

\section*{D.12.4. Context Expressions}

\textbf{Status: N} A \texttt{context} expression specifies which elements of the client’s context may affect the performance of a request by the object. It consists of a list of string literals, and follows OMG IDL syntax rules.

\section*{D.13. Attribute Declaration}

\textbf{Status: N} An interface can have attributes as well as operations; as such, attributes are defined as part of an interface. An attribute definition is logically equivalent to declaring a pair of accessor functions; one to retrieve the value of the attribute and one to set the value of the attribute.

Attribute declarations follows OMG IDL syntax rules and has the same semantics. The optional \texttt{readonly} keyword indicates that there is only a single accessor function - to retrieve the attribute value.

\section*{D.14. CORBA Module}

\textbf{Status: N} None of the information listed in the corresponding section in the OMG IDL specification is valid for RL IDL. RL IDL may have a corresponding module with RL-defined names and types, if so, that will be specified at a later time.

\section*{D.15. Names and Scoping}

\textbf{Status: R} RL IDL follows the OMG IDL names and scoping rules.

\subsection*{D.15.1. Qualified Names}

RL IDL follows the OMG IDL rules for qualified names.

\subsection*{D.15.2. Scoping Rules and Name Resolution}

RL IDL follows the OMG IDL Scoping Rules and Name Resolution Rules.
D.15.3. Special Scoping Rules for Type Names

RL IDL follows the special scoping rules for type names defined by OMG IDL.
E. RL IDL C++ Language Mapping

Contents

E.1. C++ Compiler Requirements .......................... 212
E.2. The RL Namespace ..................................... 213
E.3. Modules ............................................. 213
E.4. Interfaces ............................................ 213
E.5. Valuetypes .......................................... 213
E.6. Constants ............................................ 213
E.7. Type Declaration ....................................... 214
   E.7.1. Typedef ......................................... 214
   E.7.2. Basic Types .................................... 214
   E.7.3. Constructed Types .............................. 214
   E.7.4. Template Types ................................ 215
   E.7.5. Complex Declarators ............................ 216
E.8. Exceptions ........................................... 216
E.9. Operations .......................................... 216
E.10. Attributes .......................................... 216

This document describes how Rendezvous Library Interface Definition Language (RL IDL) concepts are mapped to C++.

E.1. C++ Compiler Requirements

The IDL compiler generates C++ code that only can be compiled with C++ compilers that meet the following requirements:

- Exceptions must be supported.
- Namespaces must be supported.
- The bool type must be supported.
E.2. The RL Namespace

The RL namespace contains all library-defined classes, constants, types and other definitions.

These include

- The ObjectRef class, which is the base class of all object reference classes.
- The ObjectServant class which is the base class of all object servant classes.
- The various type declarations for base types.

E.3. Modules

Module declarations are mapped to C++ namespaces.

E.4. Interfaces

Interface declarations are mapped to C++ classes. The interface itself is mapped to a C++ class that has the same name as the interface with no base class and one pure virtual method for each of the interface’s operations. In addition, an object reference class and an object servant class is created. These classes contain the client and server stub code, respectively.

Assuming that the interface is called interface_name, the reference class is named interface_nameRef and the servant class is named interface_nameServant.

E.5. Valuetypes

Valuetype declarations are mapped to C++ classes. They are not supported by the current release of the IDL compiler, and will not be further described here.

E.6. Constants

Constant declarations are mapped directly to corresponding C++ constant declarations. Top-level constants are mapped directly to file-level constants in C++, and interface-level constants are mapped to C++ class constants. Example:

```cpp
    const long a = 234;
```
interface B {
    const short c=2;
};

would be mapped to

    const long a = 234;

class B : public RL::Object {
    static const short c;
};

    const short B::c=2;

The reason why the two constants are being treated differently is because you cannot
directly give a class constant a value in the declaration in C++.
Constants are not supported by the current release of the IDL compiler.

E.7. Type Declaration

E.7.1. Typedef

Type definitions using the RL IDL typedef keyword will be mapped directly to the
corresponding C++ typedef declarations.

E.7.2. Basic Types

Table E.1 shows the mapping from RL IDL types to C++ type definitions, and the
Corresponding native type for 32-bit architectures (for which sizeof(int)=4). For other
architectures, the C++ native types may be different, but the C++ type definitions will
be the same. To ensure interoperability between platforms, use the C++ type definitions
instead of the native types.
The wchar IDL type may also be mapped to the unsigned wchar_t C++ type, if
that type exists.

E.7.3. Constructed Types

Constructed types are not supported by the current release of the IDL compiler, and will
not be described in detail here. A brief description follows.
<table>
<thead>
<tr>
<th>RL IDL type</th>
<th>C++ type def.</th>
<th>C++ native type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>RL::Float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>RL::Double</td>
<td>double</td>
</tr>
<tr>
<td>long double</td>
<td>RL::LongDouble</td>
<td>long double</td>
</tr>
<tr>
<td>short</td>
<td>RL::Short</td>
<td>short int</td>
</tr>
<tr>
<td>long</td>
<td>RL::Long</td>
<td>long int</td>
</tr>
<tr>
<td>long long</td>
<td>RL::LongLong</td>
<td>long long</td>
</tr>
<tr>
<td>unsigned short</td>
<td>RL::UnsignedShort</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>unsigned long</td>
<td>RL::UnsignedLong</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>char</td>
<td>RL::Char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>wchar</td>
<td>RL::Wchar</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>boolean</td>
<td>RL::Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>octet</td>
<td>RL::Octet</td>
<td>unsigned char</td>
</tr>
</tbody>
</table>

Table E.1.: Mapping of basic types

E.7.3.1. Structures

Structures are mapped to C++ structures, and have no base class.

E.7.3.2. Discriminated Unions

Discriminated unions are mapped to C++ unions.

E.7.3.3. Enumerations

Enumerations are mapped directly to corresponding C++ enumerations.

E.7.4. Template Types

E.7.4.1. Sequences

Sequences are mapped to C++ classes which implement operations to get and set the sequence’s maximum length. The class will use operator overloading to give array-like element access (using the [] operator).

E.7.4.2. Strings

Strings are mapped to the C++ type const char *.
E.7.4.3. Wstrings

Wstrings are mapped to the C++ type `const short int *`, or the type `const wchar_t *`. Support for wstrings are partial in the current release of the IDL compiler. Presently, wstrings are mapped to `const char *`.

E.7.5. Complex Declarators

E.7.5.1. Arrays

One-dimensional arrays are mapped to the corresponding C++ pointer type, for example are `long[]` mapped to `RL::Long *`.

E.8. Exceptions

Exceptions are mapped to C++ classes. They are not supported by the current release of the IDL compiler, and will not be described in detail here.

E.9. Operations

Operations are mapped to C++ pure virtual methods. Input parameters are passed by value, and parameters are passed by reference. For array parameters, an additional parameter is added after the array parameter. It has the type `RL::Long`, and the same name as the array parameter with "_size" appended. For example, a input parameter of the `long[]` type would be mapped to the following pair of parameters.

```plaintext
RL::Long * param_name, RL::Long param_name_size
```

This additional parameter contains the number of elements in the array.

E.10. Attributes

Attributes are mapped to a C++ class data member, and a pair of accessor functions to set and retrieve the data member's value. Since attributes are not supported by the current release of the IDL compiler, they are not described in detail here.
F. RL IDL Compiler Invocation

In figure F.1, a short description of the options available in the RL IDL compiler is given. It is identical to the text presented when this command is given:

    rlidl --help

In figure F.2, a syntax tree that was produced with the -t option to rlidl is presented. It was compiled from the following IDL file.

    interface Hello {
        void hi(in string a);
    };

Usage: rlidl [OPTION...] FILE
RL IDL compiler
(C) 2002 Kenneth Pedersen, Kjetil Pedersen, Carl Erik Hauge

--debug-parser Print debugging output from bison parser
-q, --quiet Be quiet -- only print critical errors
-t, --vcg-tree=FILE Produce a VCG syntax tree to FILE.
-v, --verbose Be verbose about what the compiler’s doing
-?, --help Give this help list
--usage Give a short usage message
-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

RL IDL compiler comes with NO WARRANTY, to the extent permitted by law.
You may redistribute copies of RL IDL compiler under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING.

Report bugs to <kennep@stud.ntnu.no>.

Figure F.1.: RL IDL Usage
Figure F.2: Syntax tree produced by the IDL compiler
G. Installing RL

Contents

G.1. Getting RL ...................................................... 220
   G.1.1. Prerequisites ............................................. 221
G.2. Compiling RL .................................................. 221
   G.2.1. Configuring ............................................... 221
   G.2.2. Making ................................................... 222
G.3. Installing RL .................................................. 222

In this chapter instructions for downloading, compiling and installing RL will be given. The installation procedure consists of the following steps:

1. Getting a copy of the RL source code

2. Compiling the source code

3. Installing the binaries

RL uses GNU automake and autoconf, so compiling and installing it should be easy if one has installed other automake/autoconf packages before.

G.1. Getting RL

The RL source code, and any packages that are needed for compiling RL, is included on the CD that is included with this report. However, they can also be downloaded from the web.

The RL package itself can be downloaded from the following URL:

http://faerun.dhs.org/tribal/software.php
<table>
<thead>
<tr>
<th></th>
<th>Version</th>
<th>Download URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex</td>
<td>2.5.4a</td>
<td>ftp://ftp.gnu.org/gnu/non-gnu/flex/flex-2.5.4a.tar.gz</td>
</tr>
<tr>
<td>bison</td>
<td>1.35</td>
<td>ftp://ftp.gnu.org/gnu/bison/bison-1.35.tar.gz</td>
</tr>
<tr>
<td>treec</td>
<td>0.0.6 (patched)</td>
<td><a href="http://www.faerun.dhs.org/diploma/software.php">http://www.faerun.dhs.org/diploma/software.php</a></td>
</tr>
<tr>
<td>STL</td>
<td>3.3</td>
<td><a href="http://www.sgi.com/tech/stl/">http://www.sgi.com/tech/stl/</a></td>
</tr>
</tbody>
</table>

Table G.1: Prerequisites for compiling RL

G.1.1. Prerequisites

Table G.1 lists the prerequisites for compiling RL. These must be installed before trying to compile the package. In addition, a C++ compiler, the make tool, and a set of standard UNIX utilities are needed. Note that STL, the standard template library, usually come packaged with the C++ compiler. The flex and bison programs are standard UNIX tools and usually come packaged with the operating system. In the unlikely event that they are missing, they can be downloaded from the URL’s listed in the table.

G.2. Compiling RL

The compilation process consists of the following steps:

1. Configuring the source package using the configure script
2. Compiling the package using the make program

G.2.1. Configuring

The first step of the compilation process is to configure the source three for the target system. This is done by the configure script. Enter the following command:

```
./configure
```

By default, the configure script will configure RL to be installed in /usr/local. If another installation location is wanted, it can be specified with the --prefix option to configure, like in the following example:

```
./configure --prefix=/usr/local/RL
```

configure also supports many other options. For a complete listing of the options supported by configure, issue the following command:

```
./configure --help
```
G.2.2. Making

To begin the compilation process, issue the following command:

make

This will compile all of RL’s components.

G.3. Installing RL

RL can be installed with the following command in the target installation location:

make install

The following files are installed ($prefix is used to denote the target installation location specified when configuring the package):

- The RL library: $prefix/lib/libRL.*
- Header files for development: $prefix/include/RL/*
- The IDL compiler: $prefix/bin/rlidl
- The directory service: $prefix/bin/rldirectory
- The IDL specification for the directory service: $prefix/share/DirectoryService.idl
H. About the Source Code

Contents

- H.1. Key Figures .............................. 223

H.1. Key Figures

To save space (and trees), the complete source code will not be included in the appendix. Instead, it is included on the CD that accompanies this report, and it can also be downloaded from the web (see appendix G for download instructions). The complete source code for the directory service is included in this appendix, since it is only a few pages long.

Table H.1 presents some key figures on the RL source code, and table H.2 shows the same figures for the area computation problem used for testing. The source line counts were calculated with the `sloccount` program[38], and exclude comments and blank lines. Makefiles and scripts used for running the tests and collecting and extracting the results are also excluded from this count.

H.2. Directory Service Source Code

Since the directory service is a fairly simple application, and serves as a good example of how to write an RL application, the complete source code is included here.

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of code lines</th>
<th>Number of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL Run-time system</td>
<td>1390</td>
<td>33</td>
</tr>
<tr>
<td>IDL Compiler</td>
<td>3578</td>
<td>19</td>
</tr>
<tr>
<td>Directory Service</td>
<td>126</td>
<td>5</td>
</tr>
<tr>
<td>Total, core</td>
<td>5094</td>
<td>57</td>
</tr>
</tbody>
</table>

Table H.1.: Source code key figures
<table>
<thead>
<tr>
<th>Component</th>
<th>Number of code lines</th>
<th>Number of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial version</td>
<td>270</td>
<td>3</td>
</tr>
<tr>
<td>MPI version</td>
<td>749</td>
<td>6</td>
</tr>
<tr>
<td>OOMPI version</td>
<td>726</td>
<td>6</td>
</tr>
<tr>
<td>CORBA version</td>
<td>944</td>
<td>9</td>
</tr>
<tr>
<td>RL version</td>
<td>831</td>
<td>8</td>
</tr>
<tr>
<td>Total, area computation problem</td>
<td>3520</td>
<td>32</td>
</tr>
</tbody>
</table>

Table H.2: Source code key figures

/***********************************************************/
directoryservice.h - Dir.Service impl. header
begin
copyright : Mon May 13 2002
email     : (C) 2002 by Kjetil Pedersen
cynon@hotmail.com
/***********************************************************/

namespace DirectoryService {

/**
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 **/  
#endif

#define DIRECTORYSERVICE_H

#include "item.h"
#include "Directory/Service.h"

class DirectoryService : public Directory::Service, public RL::Task {
  int objectid;
  Item *head;
  Item *pos;
  Item *end;
public:

224
DirectoryService();
void Bind(const char *n, const char *obj);
const char* Resolve(const char *n);
void Unbind(const char *n);
void Dump();
DirectoryService();
protected:
  void Run();
  Item* Traverse(const char *n);
};
#endif

/*****************************************************************************
directoryservice.cpp - Dir.Service Implementation
-----------------------
begin : Mon May 13 2002
copyright : (C) 2002 by Kjetil Pedersen
e-mail : cynon@hotmail.com
*****************************************************************************/

/*****************************************************************************
* *
* This program is free software; you can redistribute it and/or modify *
* it under the terms of the GNU General Public License as published by *
* the Free Software Foundation; either version 2 of the License, or *
* (at your option) any later version. *
* *
******************************************************************************/

#include "Directory/ServiceServant.h"
#include "directoryservice.h"
#include <stdlib.h>
#include <stdio.h>

DirectoryService::DirectoryService() {
  objectid = 1337;
  head = new Item();
  //pos = new Item();
  //end = new Item();
  pos = head;
  end = head;
  printf("\nDirectory Service initialized\n");
}
DirectoryService::DirectoryService()
{
}

void DirectoryService::Bind(const char *n, const char *obj){
    Unbind(n);
    Item *last = new Item();
    end->next = last;
    end = last;
    last->name = n;
    last->object = obj;
    printf("\nObject \%s bound to name \%s\n", obj, n);
}

void DirectoryService::Unbind(const char *n){
    if(pos = Traverse(n)){
        printf("\nReference \%s to object \%s deleted\n", n, pos->next->object);
        Item *temp = pos->next;
        if(pos->next->next != NULL) // checks if it is the last item
            pos->next = pos->next->next;
        else{
            pos->next = NULL;
            end = pos;
        }
        delete temp;
    }
    else
        printf("\nReference \%s not found\n", n);
}

const char* DirectoryService::Resolve(const char *n){
    if(pos = Traverse(n)){
        printf("\nFound \%s\n", pos->next->object);
        return pos->next->object;
    }
    return "";
}

Item* DirectoryService::Traverse(const char *n){
    pos = head;
    while(pos->next != NULL){
        if(strcmp(pos->next->name, n) == 0)
            return pos;
        pos = pos->next;
    }
    return NULL;
}

226
void DirectoryService::Dump(){ //test function, delete...
    printf("\nStart List");
    pos = head;
    int t = 1;
    while (pos->next != NULL){
        printf("\nSlot %i: reference `%s’ bound to `%s’", t, pos->next->object, pos->next->name);
        pos = pos->next;
        t++;
    }
    printf("\nEnd List\n");
}

void DirectoryService::Run(){
    Directory::ServiceServant bs(this, this);
    bs.Activate(objectid);
    while(1){
        bs.Accept();
    }
    return;
}

.translatesAutoresizingMaskIntoConstraints

itemId - description

begin : Mon May 13 2002
copyright : (C) 2002 by Kjetil Pedersen
email : cynom@hotmail.com

*******************************************************************************

/*******************************************************************************
*
* This program is free software; you can redistribute it and/or modify *
* it under the terms of the GNU General Public License as published by *
* the Free Software Foundation; either version 2 of the License, or *
* (at your option) any later version. *
* *
*******************************************************************************

#ifndef ITEM_H
#define ITEM_H

227
class Item {
    public:
    const char *name;
    const char *object;
    Item *next;
    Item();
    ~Item();
};
---

begin : Mon May 13 15:02:27 GMT 2002
copyright : (C) 2002 by Kjetil Pedersen
email : cynom@hotmail.com

******************************************************************************

/**
 * This program is free software; you can redistribute it and/or modify *
 * it under the terms of the GNU General Public License as published by *
 * the Free Software Foundation; either version 2 of the License, or *
 * (at your option) any later version. *
 * *
 * ****************************************************************************/

#include <config.h>
#endif

#include <iostream.h>
#include "directoryservice.h"
#include <stdlib.h>

int main(int argc, char *argv[])
{

    cout << "Starting Directory Service" << endl;
    TCPLocation srvloc(INADDR_ANY,RL_DIRECTORY_SERVICE_PORT,0);
    DirectoryService interface;
    interface.Start(&srvloc);
    interface.WaitFor();

    return EXIT_SUCCESS;
}
---

229
References


