A Framework for Network Software

Deepak Venugopal

&

Nadir Latif

Supervised by

Verónica Gaspes
Preface

This thesis report entitled *A Framework for Network Software* has been written for the partial fulfillment of our Master Degree in Computer Systems Engineering at Halmstad University, Sweden. This report includes a brief documentation of the library that we have created and also how it can be implemented.

We would like to thank our guide and supervisor Verónica Gaspes for all the motivation and timely help she has put in.

We would like to extend our sincere gratitude to all those who have helped us in the successful completion of this project.
Contents

Abstract

1 – Introduction
  1.1 - Purpose 5
  1.2 - Objectives 6
  1.3 - Introduction to Computer Networks 6
  1.4 - Introduction to Network Software 6

2 - Protocols
  2.1 - Introduction to Protocols 8
  2.2 - Protocol Graphs 8
  2.3 - Protocol Frameworks and Development 9

3 - Structuring Network Software
  3.1 - Introduction to Morpheus 11
  3.2 - Morpheus Objects 11
    3.2.1 - Utility Objects 12
    3.2.2 - Protocol Component Objects 12
  3.3 - Protocol Shapes 15

4 - Programming Methodology
  4.1 - Introduction to Object Oriented Programming 16
  4.2 - Introduction to Concurrent Programming 17
    4.2.1 - Introduction to MPD 18
    4.2.2 - Reactive Objects 19

5 - Implementation 21

6 - Conclusions and Suggestions 37

7 - References 38
Abstract

This thesis deals with the creation of a library that can be used in the construction of network protocols. The library provides functions that are useful for organizing protocols as stacks of layers and for processing packets. The thesis implements the main concepts of the special purpose language, Morpheus [2]. Morpheus was designed by Mark Bert Abbott at the University of Arizona in 1994 but has not yet been implemented. We have implemented the main concepts of Morpheus by using a concurrent programming language called MPD (Multithreaded, Parallel and Distributed) [5] and a programming discipline known as reactive objects. Morpheus is based on the concept of protocol shapes. A protocol shape is a template for implementing protocols with certain functions. Three main protocol shapes are router, sequencer and multiplexor. The protocol implementer constructs protocols by extending these shapes. Our library implements these shapes as objects. Each of these shapes in turn extends from a base protocol object called Protocol. We have used reactive objects for allowing communication between protocols.
Introduction

1.1 - Purpose

This project is motivated by the fact that several companies rely on proprietary protocols for the proper functioning of their network hardware. Thus the creation and implementation of protocols is an ongoing process. The design, creation and implementation of such protocols is a tedious process and there is no standard software or programming model that is used for this purpose. The number of people who go online or who prefer to use online services is increasing exponentially. This sudden demand for network or Internet based applications has pushed the limits of software creation. Powerful networking software is essential to make sure that these applications run and communicate smoothly amongst themselves. Network software should be designed in such a way that it works smoothly with the underlying hardware. It should be able to interact with any type of network hardware and at the same time provide a high throughput. These reasons have largely limited the implementation language to c. It is possible to build powerful applications using c but it is time consuming, difficult and error prone. Major companies like CISCO and Nortel and many research labs like the University Corporation for Advanced Internet Development are actively involved in research and development for creating systematic software architectures and languages for constructing network software. Most of these methods try to express the design of protocols on a modular basis such as by using x-kernel [2] while others focus on a more object-oriented approach as in Morpheus [2].

It is quite difficult to define a new methodology and to design a programming language from scratch that can be used in the design of network software. The common approach is to use a general-purpose language that supports all features required for protocol development and then utilize the available features of that language to design the required network software. However this approach does not permit fine grained optimization of the library. Our library is an implementation of the concepts of a protocol implementation language called Morpheus using a concurrent programming language called MPD (Multithreaded, Parallel and Distributed) [5]. This thesis project aims to simplify the creation and implementation of protocols and network software.
1.2 - Objectives

The main objectives of the project are two fold:

- To provide an implementation of the main concepts of Morpheus, a protocol implementation language, using a programming language that supports Object Oriented Programming techniques and concurrency. This project is implemented by using one such language called MPD (Multithreaded, Parallel, and Distributed Programming) on a UNIX environment. The outcome will be a library.
- To test the library by implementing a simple network protocol.

1.3 - Introduction to Computer Networks

The terms networks and networking have become quite common today. To most people a network is a set of cables and hardware that is used to interconnect electronic devices. The main types of networks are: computer, mobile, video and telephone networks. Voice, video and telephone networks all have one thing in common. They are special purpose. This means that they are used to transmit one type of data. i.e. voice, video or keystrokes. They also connect to special purpose devices like televisions, receivers or terminals. On the other hand computer networks are used to transmit different types of information. The generality of computer networks sets them apart from other types of networks.

Computer networks are built mainly from general purpose programmable hardware and are not optimized for specific use, such as making phone calls or delivering video signals. They can carry different types of data and they support a growing range of applications. Initially computer networks consisted of long cables that were used to connect different computers. Today computer networks can make use of both wired and wireless mediums for sharing and distributing information. Computer networks can be broadly classified based on scale, connection methods, and topology and even by the protocol they run on. Computer networks have become so widespread that computing speed is no longer dependant solely on the processing power of nodes but also depends on the speed at which data is transmitted across a network. These days computer networks offer increased reliability through redundant hardware and replicated data and programs.

1.4 - Introduction to Network Software

The exchange of information between two computers or nodes does not depend only on the underlying hardware. It depends to a large extent on the communication protocols and network software that makes use of the hardware. It is estimated that the number of
people who use the internet has crossed over 1 billion and is fast growing. With new computers and hardware being added every second, the internet is by far the biggest network ever built by man. Network architectures and protocol specifications are essential components, but these are not enough to explain the success of the Internet. There are many contributing factors like faster and more powerful hardware, but the main factor that has made the Internet a huge success is that much of its functionality is provided by software that runs on general purpose computers or workstations. This software is usually referred to as network software. Network software is responsible for the efficient transfer and synchronization of data between two communicating machines, also referred to as hosts. The primitive services that are provided by the underlying hardware are subjected to data loss and limit the number of hosts that can be connected to each other. Network software overcomes the shortcomings of the primitive hardware services by adding new services like better flow control, congestion control, error detection and correction etc. These services are usually referred to as communication services.

Communication services may be categorized as unicast, multicast or remote procedure call. Unicast communication involves communication between two hosts. Multicast communication on the other hand refers to communication that goes from one member of a particular group to all the other members of the group. Remote Procedure Call is a communication service in which the node that initiates the communication is blocked until it receives a reply message.

Network software is highly structured and is implemented as layers. Each layer uses and extends the communication services provided by the underlying layers. There are two main benefits of layering. Firstly it helps in managing the complexity of the problem by dividing it into smaller, more manageable components. Instead of implementing a single piece of software to solve the whole problem, different layers are implemented, each of which solves one aspect of the problem. Secondly, it provides a more modular design. If one wants to add a new service then one needs to only modify the functionality at one particular layer. Each layer usually represents a particular network protocol. Protocols specify rules and conventions that govern the exchange of messages. A message in simple terms is a finite number of bits that contains data from higher level protocols and control information.
### Protocols

#### 2.1 - Introduction to Protocols

The flow of data across networks is governed by protocols. Network Protocols can be defined as a set of standards on which network operations should be defined and performed. They provide a standard which allows hosts to exchange information between each other. The implementation of a protocol on a network layer or a host is referred to as a protocol entity. It is this entity that allows the exchange of information between other hosts that implement the same protocol. These hosts are referred to as peers. The communication between two hosts can be either unidirectional or bi-directional. The one way or unidirectional communication is implemented by using asymmetric protocols wherein there are two entities each for sending and receiving. The bidirectional communication on the other hand is implemented by using symmetric protocols wherein all the entities have the same functionality.

Each protocol defines two interfaces. First, it defines a service interface to other objects on the same computer that want to use its communication services. This service interface defines the operations that local objects can perform on the protocol. Operations can often resemble a request-reply wherein an application sends and receives messages. Second, a protocol defines a peer interface to its counterpart on another machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service. This would determine the way a request-reply protocol on one machine communicates with its peer on another machine. A protocol defines a communication service that it exports locally, along with a set of rules governing the messages that the protocol exchanges with its peers. The messages are usually sent to lower level protocols which then deliver the message to the peer. There are usually multiple protocols operating on the same level, each providing a different communication service. The protocols that make up a network system can be easily represented by using a protocol graph.

#### 2.2 - Protocol Graphs

The collection of protocols that make up a network system, as mentioned earlier, can be represented by using a protocol graph. A typical protocol graph is represented below in fig 2.1. The nodes on the graph represent protocols and the arrows represent a depends-on or use relation. The various protocol layers on a network system need not be arranged linearly. In a normal network system, a lower level protocol may provide support to various
higher level protocols, each of which has a different communication service. A given protocol may also utilize several lower level services

![Diagram](image)

**Figure 2.1 (taken from [2])**

### 2.3 - Protocol Frameworks and Development

The development of new computer hardware and the release of software that supports them have pushed software developers into developing new network software. It is a well known fact that network hardware and software are changing at a tremendous rate. Network software lets the system have a high fault tolerance and a high throughput. A change in the basic hardware that makes up the network system leads to a change in the communication service at the foundation of the network architecture. It is quite common that newer network hardware provides a variety of network functions. A modern day router for example can be used to not only route packets but also for header validation, error checking and implementing QoS. The changing hardware can thus be a motivation in changing the software. Application software is also evolving at a rapid rate. New applications like multimedia motivate changes in network software by adding new communication services. Multimedia applications use the same packets and network to transmit both voice and data unlike its predecessors where separate channels were used. Network software must therefore be able to incorporate these changes. Another major factor that motivates the development of network software is that the internet is expanding at a tremendous rate and software should be updated to address routing and addressing changes.
There are many methods that have been suggested for the development of protocols. However there are two main forms of protocol developments that are quite common these days. These are Protocol Frameworks and Formal Description Techniques.

The protocol framework refers to an environment that allows specification and execution of communication protocols. The protocol offers two main services. Firstly a protocol framework specifies a set of structural guidelines that all protocols implemented using that framework must adhere to. Secondly the protocol framework provides a library of common functions. System V Streams and the x-kernel are two main protocol frameworks that have been used in the active development of network protocols.

In System V Streams, all protocols provide a block-oriented interface to their neighboring protocols [2]. Each protocol provides 2 queues, one for servicing a higher level protocol and the other for servicing a lower level protocol. The queues are used to store blocks. Each block buffers identity and the parameters of the operations that it is capable of handling. The arrangement of the blocks in the queue can be modified during runtime. The x-kernel began as an operating system but then grew to be widely used as a network subsystem that runs inside other operating systems. The x-kernel has a call-oriented interface in which all operations are performed by calling specific functions with appropriate parameters. The x-kernel provides an architecture for modular protocol construction. For e.g. Operations involving message transfer take a message as one of the parameters. Messages are represented by an abstract data type whose operations are library routines or macros provided as part of the x-kernel. The x-kernel also has built in support for countdown timers and hash tables. The countdown timer keeps track of messages after they are sent and is used to determine if it has been lost.

Formal Description Techniques (FDT) [2] are used in cases where there are precise requirements or in cases where a protocol has to be built as per a given specification [4]. The main advantage of FDTs is that it allows the programmer to manipulate, analyze, and predict the behavior of the system during the design stage. A protocol specification may be defined by using two properties, general and specific. General properties are properties that every protocol should have, such as being deadlock-free and not having unexecutable code. Specific properties are those that are specific to a particular protocol, such as delivering messages in order.

DSL or Domain Specific Language [4] is a type of programming language that is dedicated to a particular application domain or problem. DSL and its variant DSEL (Domain Specific Embedded Language) can be used to design and implement protocol stacks that are specific to a particular domain.
Structuring Network Software

3.1 - Introduction to Morpheus

Morpheus [2] is an abstract model for network protocol implementation that suggests techniques for supporting the construction of network protocols. This thesis is a practical implementation of the main concepts of Morpheus.

Morpheus specifies high level abstractions for supporting network protocol construction. It does not specify low level machine specific protocol implementation details. High level abstractions such as messages and connections are an important part of Morpheus. They allow main protocol functionality to be expressed in a few lines using concise statements. The programmer does not have to specify low level implementation details. This makes protocol implementation programs easier to understand, write, debug, and modify.

Morpheus encourages a modular implementation of network software in which individual parts of the software interact with each other through well defined interfaces. This allows major functionality to be expressed as individual modules. These modules be designed, implemented, and modified independently of each other. A modular implementation of software allows it to be better understood and designed.

3.2 - Morpheus Objects

Morpheus’s high level protocol abstractions are represented as objects. These objects like those supported by conventional programming languages contain data and procedures. There are two main types of objects in Morpheus. Utility Objects and Protocol Component Objects. Utility Objects provide commonly used functionality such as byte ordering and are directly instantiated. Protocol component objects on the other hand represent basic types of protocols and have to be refined by the protocol implementer by using inheritance.

3.2.1 - Utility Objects

The utility objects help protocols perform common services. The utility objects are Maps, Events and Messages.

Maps provide a general mapping service that maps values of one type into another type. E.g mapping between different types of addresses. Maps provide a general way of storing data and provide operations for entering, looking up, and deleting mappings from one value to another. A Map is implemented as a hash table.
Events provide means for scheduling the execution of specified functions. The events may be scheduled or cancelled. E.g. Operations such as sending periodic “I am alive” messages may be implemented using events. Morpheus implements Events using operating system timing support.

Message objects are used to represent individual messages. These messages are represented by Morpheus in a certain way. For e.g. header fields of messages are always word aligned, making access more efficient. However this requires a constraint on protocol specifications that requires that headers and data each be an integral number of words and that individual header fields be word aligned relative to the start of the header. Additionally the Message objects perform automatic byte order conversion of header fields. The byte ordering used by the target machine, on which the protocol will run must be specified to Morpheus’s compiler which in turn generates the appropriate Message object code for accessing header fields. This allows the protocol implementation to automatically perform byte conversion even though the actual protocol may specify a different byte conversion.

3.2.2 - Protocol Component Objects

Protocol component objects represent base classes that are used by the programmer to implement protocols. A protocol is implemented by deriving a class from one or more protocol component classes. A subclass is derived from a base class by adding more instance variables and/or adding additional procedures. Deriving from a base class in effect adds new state information and/or extends the base class behavior.

The base class to subclass derivation is shown in figure 3.1.
According to Morpheus, a protocol implementation should consist of multiple instances of object subclasses as opposed to instances of the main protocol component classes. It is possible for a single host to have more than one instance of a given protocol in its protocol graph.

The main protocol component base classes are `Protocol`, `OverSap`, `UnderSap`, `OverSession`, and `UnderSession`. OverSaps and UnderSaps are components of Protocols and OverSessions are components of OverSaps while UnderSessions are components of UnderSaps. A diagram representing these base classes is shown in figure 3.2. A protocol entity in Morpheus is represented by a protocol object, which has been derived from one or more base protocol component classes. A Service Access Point (SAP) is an object that implements an interface between a communication service and a user of that service. The communication service and users are both protocol objects. A communication service is implemented by a protocol that is at the top of a directed graph of lower level protocols.

![Diagram of protocol classes](taken from [2])

A SAP can also be referred to as an interface between protocol entities through which they can communicate. Communication in each direction is implemented by a separate pair of SAP objects. E.g. If Protocol A needs something from Protocol B, i.e. A is a user of a service provided by B, then it will get the required information through one of its SAP objects, which will be known as the UnderSap object for the communication. The corresponding SAP object on Protocol B will then be known as the OverSap object for the communication. Hence the protocol that uses the service is referred to as, being a higher level protocol then the protocol that provides the communication service. In a protocol
stack there are multiple protocol entities that are arranged in a systematic way. Each Sap object has a unique address associated with it. A single protocol object may have many SAP objects each of which serves many higher or lower level protocol objects. It is also possibly for an OverSap object to provide service to multiple UnderSap objects belonging to different protocol entities.

Each SAP consists of a pair of objects, with each object belonging to a different protocol. A SAP can hence be termed as a two-way interface and is represented as an OverSap-UnderSap pair. The operations provided by a SAP object are invoked by the adjacent SAP object of the other protocol. For each OverSap or UnderSap object belonging to a Protocol, the other Protocol sharing the object has a corresponding object, an UnderSap or OverSap, which provides operations invoked by the first Protocol. An SAP hence has a two-way interface and is represented as an OverSap-UnderSap pair. The SAP object is said to be over or under based on the protocol of which it is a component. This pair is represented in figure 3.3.

![Figure 3.3 (taken from [2])](image)

A *conversation* is said to occur when messages are exchanged between two SAP objects that form a pair. The interfaces used for conversation are considered to be different from the SAP interface. At each end of the communication, the communication interface is implemented by a *session* object. Operations on a session object do not need to specify the conversation because each session object corresponds to a particular conversation. Each conversation has two corresponding session objects that are unique. One is the *oversession* object and the other is the *undersession* object. Each of these belong to the OverSap and UnderSap object respectively. Each session corresponds to two addresses, which are the
same as that of their respective SAP object. SAP objects create different interfaces for different conversations. Unlike SAP objects Sessions are two way interfaces and allow the flow of data in both directions. Hence a protocol that uses the service of another protocol can send and receive information over its UnderSap. However in order for the other protocol to use the services of the first protocol, a separate SAP pair is required.

3.3 - Protocol Shapes

Morpheus classifies protocols into three main types called Shapes, based on their functionality. A network protocol is created from various combinations of protocols that belong to these three shapes. The three main shapes are multiplexor, worker and router. The shapes are described by the illustration below in figure 3.4. The multiplexor protocol as its name implies is responsible for multiplexing and demultiplexing messages that are sent between a single SAP object and more than one SAP objects. Router protocols are responsible for making routing decisions. They need to choose the best OverSap or UnderSap object for forwarding a message.

![Figure 3.4 (taken from [2])](image)

These routing decisions are either made on every message or for a session as a whole. The routers in Morpheus not only determine the path but they also calculate the series of protocol entities that a message traverses in the protocol graph of a host. The worker protocols provide basic protocol functionality such as error detection, buffering for retransmission and detecting lost, reordered, or duplicated messages. The message data is only manipulated by the worker protocols which act as message filters.
Programming Methodology

This thesis project provides an implementation of the main concepts of Morpheus. The first and foremost step in implementing any software, network or general, is to select an appropriate programming language. The choices can be easily narrowed down after reading and understanding the requirements of Morpheus. Since Morpheus considers protocols as objects, it is always advisable to use a language that supports Object Oriented Programming (OOP). Protocols and network software in general do not run on single machines and so the programming language that is to be selected has to allow the programmer to create distributed applications that run concurrently on several machines at the same time. We will take a brief look at the basics of OOP and Concurrent programs before selecting a language that is appropriate for the implementation of Morpheus.

4.1 - Introduction to Object Oriented Programming

Object oriented programming may be described as the practice of using objects to organize code and data. Each object has its own code and data and has specific behavior and responsibilities. Objects may be regarded as instances of types called classes. The main purpose of object oriented programming is to make the code easier to manage and extend. However this purpose can only be achieved if the implementation of objects is properly planned out. The basic concepts of Object Oriented Programming are as follows:

Abstraction

This is the practice of creating class definitions that do not contain implementations. Their purpose is to describe the functionality of a class in purely abstract terms in such a way that the functionality of the object that is based on the class is easy to understand. [6]

Encapsulation

Encapsulation is the practice of hiding implementation details of an object (i.e code and data) so that the object appears as a self sufficient entity with as few external dependencies as possible. Encapsulation makes the code easy to understand and manage. [6]

Specialization

This can be regarded as the ability of an object to inherit all or some of the operations and data of a super class. The super class is defined as abstract and has no implementation details. Specialization allows us to add these details using objects derived from abstract classes. [6]
**Inheritance**

Inheritance is the ability of a class to include all or part of the definitions of another class in to its own definition. The parent class is termed as the super class while the derived class is termed as the subclass. A super class may be a sub class of some other super class. Similarly a sub class may be a super class of some other class. [6]

**Polymorphism**

Polymorphism may be regarded as the ability to use the same name for operations on objects of different classes. There are two main types of polymorphism in object oriented programming. Inherent polymorphism is the ability to define a method in a class and to ensure that the method is available for all classes derived from the class. The implementation of the method may vary from subclass to subclass. Adhoc polymorphism is the ability to give methods of different classes the same name. [6]

**Code Reuse**

This is the practice of reusing code that may be written by different people. Object oriented programming through its mechanism of polymorphism and inheritance allows even greater code reuse than structured programming. [6]

4.2 - Introduction to Concurrent Programming

A concurrent program contains two or more processes that work together to perform a task. Each process is a sequential program or a sequence of statements that are executed one after the other. A sequential program in other words has a single thread of control while a concurrent program has multiple threads of control. The processes in a concurrent program work together as one single program by communicating with each other using shared memory or by sending message. This may be done using shared variables or by message passing. When concurrent processes use a shared variable or a shared buffer, data is written into a common memory location and is accessed by all the running processes. In the case of message passing, one process sends a message that is received and acted upon by another process. In order to achieve concurrency, the running processes need to synchronize with each other. There are two main types of synchronization, Mutual Exclusion and Condition Synchronization.

Mutual Exclusion is the process by which critical sections are not allowed to execute at the same time. A critical section can be defined as a section of code that accesses a shared resource, such that, the resource must not be concurrently accessed by more than one thread of execution. In other words mutual exclusion is used to prevent the simultaneous use of a common resource. Condition synchronization, on the other hand, is the process by which a
process is delayed until a particular condition holds true. Like processes, some programming languages like C and Java use threads for concurrency and these programs are hence known as multithreaded programs. Threads and processes essentially do the same job; they are concurrency constructs that allow the programmer to construct an application with parts that execute simultaneously. If the system provides both threads and processes the programmer can use whichever he/she prefers to implement a concurrent application. In most applications though, threads are the preferred method of implementing concurrent activities as they impose a much lower overhead on the system during a context switch and therefore execute faster. Although systems built using processes incur greater overheads they are inherently safer. Because each process runs in its own virtual address space, a fault in one process cannot affect the state of other processes. In a multithreaded application, erroneous behavior of one thread can cause erroneous behavior of other threads. A single unhandled error in one of the threads may be sufficient to terminate the entire application.

It is now clear that to implement network software or a framework for network software, it is essential to use a programming methodology that incorporates both concurrency and the power of object-oriented programming. MPD is one such language that allows the use of processes and objects.

4.2.1 - Introduction to MPD

MPD is a concurrent programming language that has been developed by the Department of Computer Science at the University of Arizona. MPD stands for Multithreaded, Parallel, and Distributed Programming. MPD was developed as a variant of the SR (Synchronizing Resources) programming language. SR (Synchronizing Resources) is a language for writing concurrent programs. The main components of the language are resources and operations. Resources encapsulate processes and variables they share while operations provide the primary mechanism for process interaction. MPD provides a novel integration of the mechanisms for invoking and servicing operations. MPD supports all the functionality of a concurrent language such as local and remote procedure call, rendezvous, message passing, dynamic process creation, multicast, and semaphores. MPD also supports shared global variables and operations. MPD has a C-like syntax and C-like control statements and runs mainly on UNIX and SPARC.

The most general MPD program consists of multiple virtual machines, each of which contains resources and globals. A virtual machine is an address space that has been reserved for the execution of a particular process. A resource is similar to a Java class and it declares a pattern for a class of objects, each of which is created dynamically. A resource exports operations and its body contains variables, procedures, and processes that
implement the operations and perform background tasks. A global can be defined as a collection of variables and operations that are shared by all resources in the same virtual machine. A global is created once when it is first imported. Every MPD program contains at least one resource while the most basic MPD program consists of a single resource. In every program one particular resource is considered as a main. The main resource can have any name the programmer chooses; it does not have to be designated ‘main’. The main resource is typically the last one that is compiled and must be the last one linked. The main resource is also the resource that starts the execution of the program. It may not be imported by any other resource nor have any parameters. Before an MPD program is executed, the MPD runtime system automatically creates one virtual machine and one instance of the main resource. The virtual machine is located on the physical machine that is used to start the MPD program. The instance of the main resource is placed in that virtual machine. Virtual machines can be dynamically created on the same physical machine or on different physical machines that are connected by a network such as a LAN. An MPD program that uses virtual machines is called a distributed program. Instances of resources are also created dynamically and are placed on virtual machines. The MPD program is then executed beginning with the first declaration or statement in the body of the main resource.

Other important features of MPD are operations. Operations are the most important mechanism in MPD. Operations are implemented by using a proc or by an input statement and can be invoked synchronously by means of the call statement or asynchronously by means of the send statement. In a call, the operation that is initiating the invocation is blocked or suspended until it receives a response from the operation being invoked. In a send, on the other hand, the operation that is initiating the invoking continues its execution without being suspended or blocked.

One of the benefits of using MPD is that it has built in event handling capabilities, so the programmer does not have to be concerned with handling events.

4.2.2 - Reactive Objects

In common concurrent and distributed programs, the input and output events are usually handled through blocking subroutine calls. E.g input given from the command line is handled using a separate thread of execution. In reality the input and output events can occur at varying frequencies. E.g data may arrive very quickly or in very large quantities. It may also arrive out of order or at unexpected times. This may cause the event handling thread to block or create inconsistencies in data. Deadlocks and loops in the thread will affect the performance of the main program. Exceptions inside the thread may also cause
the main program to crash. These problems are inherent in most thread based event handling techniques.

Reactive objects can be used to overcome these limitations. Reactive objects provide a way of organizing concurrent programs in an object oriented way. A reactive object may be regarded as a combination of an object and a process. Instantiating a reactive object creates a new process that is initially at rest. i.e it does no work. Calling specific methods on a reactive object causes it to become active, and start performing work. Once the work has been performed the object goes back to its initial rest state. The main benefit of using reactive objects is that they are completely independent of the function that “activates” them. Hence exceptions in the reactive objects execution code will not affect the calling function. The only source of deadlock would be the synchronous method call. A cyclic chain of such calls is easily detectable at run time. A reactive object in MPD is implemented by using an in statement inside a while loop. Methods specified in the in statement constitute the execution code of the object and are known as channels. The execution of the process is blocked inside the in statement until a method is invoked by an external function. A method can only be invoked if its corresponding guard statement (if any) evaluates to true. Once the method has been executed the loop causes the process to block inside the in statement again. The following code fragment from our library shows the in statement:

```
In SendMessageOverSap (message, sender, recepiant, message_id) and send_tokens > 0 >
OverSapMessageHandler(message, sender, recepiant, message_id);

[] TerminateOverSap(message, name) -> over_sap_running = false;

OverSapMessageHandler ( message, name, "", -1);

TerminateUnderSap ( name ) ;

ni
```
Implementation

Our library can be used to develop protocols for communication between two or more processes that may be on same or different hosts. To test our library we developed a test application that performs basic communication. The test application is described later. The library we have designed contains a number of resources. Image1 below shows how communication takes place between corresponding protocols (based on our library) on different stacks via lower layer protocols:

Image1: Communication between protocols built using our library.

A detailed explanation of the resources used by our library is as follows:

5.1 - Main

Program execution starts from this resource. Initially the AddProtocols function is called which creates instances of the sizer, sequencer and fcfs protocols on two protocol stacks that are represented by the ProtocolManager resource (described later). The AddProtocols function first creates an instance of the sizer protocol. It then creates two instances of the sequencer protocol, which are connected to an instance of the fcfs protocol. An instance of a protocol is created by calling the AddProtocol function of the
protocol manager resource. This function requires the following as parameters: the name of the protocol links to over sap and under sap resources, a Boolean value specifying whether the protocol is at the top of the stack and the type of protocol. Each protocol stack can have any number of protocols. Once the stacks have been created they are given as parameters to two applications. Each application is represented by the Application resource (described later) and is instantiated in a separate process and on a separate virtual machine. The virtual machines can be on the same or different hosts. We have successfully run the applications on different and same hosts. The applications are created using the Application() method.

```
process P1
{
    app1=create Application("input1.txt", "output2.txt", protocol_managers [1], "localhost2_app2", "localhost1_app1", 1) on vm1;
}

process P2
{
    app2=create Application("input2.txt", "output1.txt", protocol_managers [2], "localhost1_app1", "localhost2_app2", 1) on vm2;
}
```

5.2 - Application

This resource represents the Application. It reads one line from a text file (given as an argument to the resource) and sends it to the other application. It does this by invoking the send operation of its protocol stack. It then invokes the blocking form of the receive operation of its protocol stack. This method blocks until data arrives for the application. The received data is then written to a text file (given as an argument to the resource). This is done repeatedly until all data in the text file has been sent. The application then terminates. The constructor of the application resource is defined as follows:

```
Application (app_name in_file, app_name out_file, cap ProtocolManager protocol_manager, app_name name, app_name dest_app_name, int message_id)
```

The parameters are the names of the input and output files, capability to the applications protocol stack, the name of the first application, the name of the second application, and the type of message that the application will send.

5.3 - GlobalConstants
This resource contains definition of constants that are used by several resources. The character separating the fields in the header and the header from the message, the maximum size of a message, the maximum number of messages that can be sent or delivered by a protocol, the maximum number of protocols used in all applications and the size of the message buffer used by each application are specified in this resource. The GlobalConstants resource is defined as follows:

5.3.1 - GlobalBasicTypes

This resource contains declarations for variable and operation types. Usage of types improves code readability. The resource defines types for sequence number, application name, virtual machine name, protocol name, header field name, type of sap link (e.g. under_sap or over_sap), and protocol message. Types have also been declared for protocol message handling operations and the operation used to get the position of a field given its name. The purpose for using types for these operations is that they are passed as parameters.

5.3.2 - GlobalRecTypes

This resource contains declarations for record types. We have defined only one type of records. They are used to store variables related to a protocol link. These are: the number of sap objects, the sap objects and types of sap objects.

5.3.4 - GlobalResources

This resource is used to store instances of resources that are used by several resources. These include resource instances for Queue, MessageManager and UtilityFunctions resources.

5.4 – MessageManager

This resource is used to manage the saving, removal, retrieval and updating of messages in a buffer. The purpose of the buffer is to hold a message temporarily while it is traveling through the protocol stack of an application. Instead of sending the message string, protocols send the location of the message in the buffer. Only the protocol at the bottom of the stack sends the message string. Description of the functions implemented by the MessageManager resource is as follows:

GetMessage (int message_index) returns protocol_message message
The GetMessage function is used to return the message stored at the specified index in the message buffer without removing the message string.

RemoveMessage (int message_index) returns protocol_message message

The RemoveMessage function is used to return the message stored at the specified index in the message buffer. The message string is removed from the buffer.

SaveMessage (protocol_message message) returns int message_index

The SaveMessage function is used to save the specified message to the message buffer. It returns the position in the buffer where the message is stored.

UpdateMessage (protocol_message message, int index)

The UpdateMessage function is used to replace the message at the specified index with the specified string.

5.5 - UtilityFunctions

This resource provides general operations such as attaching a header to a message, extracting a certain field from a message containing a header and splitting a string around a specified character into two parts. Description of the functions implemented by the UtilityFunctions resource is as follows:

SplitString (protocol_message message_str, char character, res protocol_message str1, res protocol_message str2)

The SplitString function is used to divide the specified string into two parts that lie on either side of the first occurrence of the specified character in the string. The two parts are assigned to result variables. It takes as parameters the name of the message string to be split, the character around which to split the string and the two variables that will be assigned the result.

AttachHeader (protocol_message message, protocol_message header) returns protocol_message updated_message

The AttachHeader function is used to attach the specified message header to the specified message. It takes as parameters a message and a header.

ExtractHeader (ref protocol_message message, int field_position, res protocol_message field_value)
The ExtractHeader function is used to return the specified field from the header in the specified message. It takes as parameters the message string and order of the field in the header and the variable that will be assigned the result.

\[ \text{GetLength (string [\*] array [\*]) returns int array\_length} \]

The GetLength function is used to return the length of the specified array.

\[ \text{DebuggingMessage (protocol\_message debugging\_message)} \]

The DebuggingMessage function is used to display the specified debugging message. The message is only displayed if the global constant DEBUG=1.

5.6 - ProtocolManager

This resource is used to represent the protocol stack. It has a method called AddProtocol, which can be used to add protocols to the stack. It also has methods that can be used to retrieve and remove protocols from the stack. It also has methods for sending and retrieving messages using the protocol stack. It supports a method called Terminate, which essentially terminates the reactive sap object used in the protocol stack. The constructor of the ProtocolManager resource takes one parameter which is the name of the application that will use the ProtocolManager. Description of the functions implemented by the ProtocolManager resource is as follows:

\[ \text{AddProtocols (cap ProtocolManager protocol\_managers [MAX\_STACK\_COUNT], cap ProtocolManager stack\_links [MAX\_STACK\_COUNT] [MAX\_STACK\_COUNT-1])} \]

It is used to add a protocol to the stack. Its first parameter is an array of capabilities to resources of type protocol manager that represent the protocol stacks, while its second parameter is an array of capabilities to resources of type protocol manager that represent the links of each protocol stack.

\[ \text{RemoveProtocol (protocol\_name name) returns cap Protocol protocol} \]

It is used to remove a protocol from the stack. Its only parameter is the name of the protocol to be removed.

\[ \text{GetProtocol (protocol\_name name) returns cap Protocol protocol} \]

It is used to return a capability to the specified protocol without removing it from the stack. Its only parameter is the name of the protocol to be retrieved.

\[ \text{SendMessage (protocol\_message message, protocol\_name sender, protocol\_name recipient, int message\_id) returns int status\_code} \]
It is used by an application to send a message to another application. Its parameters are the message to be sent, the name of the source and destination applications and the type of message.

\[ \text{RecieveMessageBL} \left( \text{res protocol\_message message, res protocol\_name sender} \right) \]

It is used by an application to receive a message. The function blocks until a message has arrived for it in the message queue. Its parameters are the variables that will be assigned the value of the received message and the name of the sender of the message.

\[ \text{RecieveMessageNB} \left( \text{res protocol\_message message, res protocol\_name sender} \right) \text{ returns int is\_received} \]

It is used by an application to receive a message. The function does not block if a message has not arrived for it in the message queue. Its parameters are the variables that will be assigned the value of the received message and the name of the sender of the message. The function returns 1 if a message was present at the time the function was called. It returns 0 otherwise.

\[ \text{Terminate} \left( \text{app\_name name} \right) \text{ returns int status\_code} \]

It is used by an application to terminate the reactive objects in its stack. The function causes a terminate message to be sent down the stack. Its only parameter is the name of the application that calls it. It returns a status code indicating successful or unsuccessful termination.

\[ \text{InitialiseProtocolStack} \left( \text{cap ProtocolManager protocol\_manager [MAX\_STACK\_COUNT-1]} \right) \]

It is used to initialize all protocols in the stack. This involves linking each protocol in the stack. Its only parameter is an array of protocol stacks that will be linked to the current protocol stack.

\[ \text{GetLastProtocolSap()} \text{ returns cap Sap protocol\_sap} \]

It is used to get the sap object of the last protocol in the stack. It returns a capability to the sap object. The function is used to link the bottom protocols of the stacks.

5.7 - Protocol

This resource represents the base class for all protocols. It contains definition of operations that are common to all protocols. The resource implements some basic operations such as GetName, which returns the name of the protocol. It also implements send and receive operations that are invoked by the application. The constructor of the
protocol resource takes as parameters the name of the protocol, the over sap and under sap links of the protocol, a capability to the sap object of the protocol, boolean values indicating whether the protocol is at the top or bottom of the stack and a capability to a method that returns the position of a field given the field name. Description of the functions implemented by the Protocol resource is as follows:

get_field_position GetFieldPosition

It is used to return the position of the specified field in the header of a protocol message. Its only parameter is the name of the field.

message_handler OverSapMessageHandler

It is used to handle a message that has arrived on the over sap channel of the sap object of the application. This function is called by the sap object of the protocol. Its parameters are the protocol message, the sender and recipient of the message and the message type. The sender of the message is the last protocol that forwarded the message, while the recipient is the application that will receive the message.

message_handler UnderSapMessageHandler

It is used to handle a message that has arrived on the under sap channel of the sap object of the application. This function is called by the sap object of the protocol. Its parameters are the protocol message, the sender and recipient of the message and the message type. The sender of the message is the last protocol that forwarded the message, while the recipient is the application that will receive the message.

initialise_protocol InitialiseProtocol

It is used to set the over sap and under sap links of the protocol. It also creates instances of the base protocol resources. This is done to overcome the limitations of inheritance in MPD. Its parameters are the name of the protocol, the over sap and under sap links of the protocol, a capability to the sap object of the protocol and a capability to a method that returns the position of a field given the field name.

GetProtocol() returns cap Protocol protocol

It is used to return a capability to the protocol.

GetProtocolLinks() returns protocol_links saps

It is used to return the over sap and under sap links of the protocol.

GetSAP() returns cap Sap protocol_sap
It is used to return the sap object of the protocol.

\textit{GetName()}

It is used to return the name of the protocol.

\textit{GetName (res protocol\_message message, res protocol\_name sender, protocol\_name recepiant)}

It is used to retrieve a message from the queue that has arrived for the application. Its parameters are the two variables that will be assigned the message and name of the recipient of the message.

\textit{MessageReceived()} \textbf{returns bool message\_received}

It is used to determine if a message has arrived for the application or not.

\textit{GetVMName()} \textbf{returns virtual\_machine vm\_name}

It is used to return the name of the virtual machine on which the protocol is running. The name of the virtual machine is assumed to be the suffix of the protocol name.

\textit{IsLowestProtocol()} \textbf{returns bool is\_lowest}

It is used to determine if the current protocol is the lowest protocol in the stack.

\textit{IsTopProtocol()} \textbf{returns bool is\_top}

It is used to determine if the protocol is the highest protocol in the stack.

\textit{GetMessageHeader(protocol\_name sender)} \textbf{returns protocol\_message header}

It is used to get the header that will be attached to the message. The sender parameter is optional. The function returns a list of fields separated by "$\$".

\textit{GetHeaderField(protocol\_message message\_str, int field\_position)} \textbf{returns protocol\_message field\_value}

It is used to extract the specified field value from the header. Its parameters are the position of the field and the message string.

\textbf{5.8 - Queue}

This resource is used to implement message queue that is used to store messages that have arrived for the application. It contains operations for adding, removing items from the queue and getting the number of elements in the queue. Description of the functions implemented by the Queue resource is as follows:
AddItem(protocol_message message) returns int status

It is used to add a message string to the back of the queue. Its only parameter is the message string.

RemoveItem(res protocol_message message) returns int status

It is used to remove a message from the queue. It returns -2 if the queue is empty and 1 otherwise. The message is assigned to the result variable.

GetSize() returns int queue_size

It is used to return the size of the queue.

5.9 - SAP

This resource is used to allow other resources to communicate with a protocol. Instances of SAP resource are reactive objects. When an instance of the SAP resource is created two processes are started. Each allows communication in one direction (i.e. up or down the stack). Each process contains an in statement that has two channels; one for sending a message and the other for terminating the process. When a message arrives on the OverSap channel, it is passed as parameter to the OverSapMessage handling function. This function is implemented by a protocol such as the sequencer. Similarly a message arriving on the UnderSap channel is passed as parameter to the UnderSapMessage handling function. The constructor of the sap resource takes the following parameters: capabilities to the oversap and undersap message handlers and the name of the the protocol of the sap object. Description of the functions implemented by the SAP resource is as follows:

SendMessageOverSap(protocol_message message,protocol_name sender,protocol_name recepiant,int message_id)

It is used to send a message over the over sap channel. Its parameters are the message string, name of the sender and recepiant of the message and the message type.

SendMessageUnderSap(protocol_message message,protocol_name sender,protocol_name recepiant,int message_id)

It is used to send a message over the under sap channel. Its parameters are the message string, name of the sender and recepiant of the message and the message type.

TerminateOverSap(protocol_message message,app_name name)

It is used to send the terminate message over the terminate over sap channel. Its parameters are the termination message string and the name of the application.
**TerminateUnderSap(protocol_message message, app_name name)**

It is used to send the terminate message over the terminate under sap channel. Its parameters are the termination message string and the name of the application.

**GetName() returns protocol_name proto_name**

It is used to return the name of the protocol of the sap object.

**GetVMName()**

It is used to return the name of the virtual machine of the protocol of the sap object.

**SetSendTokens(int tokens)**

It is used to set the maximum number of messages that the protocol is allowed to send. Its only parameter is the number of the tokens.

**SetDeliverTokens(int tokens)**

It is used to set the maximum number of messages that the protocol is allowed to deliver. Its only parameter is the number of the token.

**IsRunning() returns bool is_running**

It is used to determine if the SAP processes are running or not.

**5.10 - Worker**

This resource extends the Protocol resource. It contains code that is common to all worker resources like sequencer. E.g. its contains methods for returning capability to the oversap and undersap objects. It also contains operations that implement default oversap and undersap message handling code. Description of the functions implemented by the Worker resource is as follows:

**GetOverSap() returns cap Sap protocol_over_sap**

It is used to return the over sap object.

**GetUnderSap() returns cap Sap protocol_under_sap**

It is used to return the under sap object.

**DefaultOverSapMessageHandler(int message_id, protocol_name recepiant, protocol_message message, protocol_message message_header)**
It contains default code for handling messages received by the oversap of the protocol. Its parameters are the type of message, the name of the protocol, the protocol message and the protocol header.

\[
\text{DefaultUnderSapMessageHandler}(\text{int message_id, protocol_name recepiant}, \text{protocol_message message}, \text{protocol_message message_header})
\]

It contains default code for handling messages received by the undersap of the protocol. Its parameters are the type of message, the name of the protocol, the protocol message and the protocol header.

\[
\text{GetSequenceNumber}(\text{protocol_message message_str}) \text{returns int field_value}
\]

It is used to extract the sequence number from the message header. Its only parameter is the message string. It returns the sequence number field inside the message header.

**5.11 - Sequencer**

This resource extends the worker resource. It is an example of a protocol that can be implemented using our library. It performs the same operation as the Sequencer given in the PhD dissertation upon which our thesis is based. It contains operations for handling messages received on the oversap and undersap channels of its sap object. It also contains the operation GetMessageHeader, which creates and returns the sequencer protocol header and the operation GetFieldPosition which returns the position in the header of a certain field (specified by name). The sequencer's role is to send messages to lower layer protocols provided the message's sequence number is less than the number of send tokens, where the number of send tokens is the maximum number of messages that the sequencer protocol is allowed to send. The sequencer also delivers messages to upper layer protocols provided the message is not out of order or duplicate and its sequence number is less than the number of deliver tokens, where the number of deliver tokens is the maximum number of messages that the sequencer protocol is allowed to deliver. The sequencer does not define its own functions but instead implements the functions defined in its base resources.

It implements the following function, InitialiseProtocol, OverSapMessageHandler, UnderSapMessageHandler, GetMessageHeader and GetFieldPosition.

**5.12 – Router**

This resource extends the Protocol resource. It contains code that is common to all router protocols like Sizer. E.g. it contains methods for returning capability to an array of undersap objects and the oversap object. It also contains operations that implement default oversap and undersap message handling code. These operations are similar to the default message handling
operations provided by the worker resource. However instead of sending or delivering a message to the next protocol, the message is placed in a send or deliver queue depending on the protocol that sent it. When an instance of the Router resource is created two processes are started that check the send and deliver queues respectively. The queues are checked periodically after an interval that can be specified in the GlobalConstants resource. Messages in these queues are handled by operations provided by resources such as Sizer that extend Router. The constructor of the router resource takes the following parameters: capabilities to the send and deliver queue handlers. Description of the functions implemented by the Router resource is as follows:

\[ \text{GetOverSap}() \text{returns cap Sap protocol\_over\_sap} \]

It is used to return the over sap object.

\[ \text{GetUnderSaps}() \text{returns cap Sap protocol\_under\_saps[MAX\_PROTOCOL\_COUNT]} \]

It is used to return the under sap objects.

\[ \text{DefaultOverSapMessageHandler}(\text{int message\_id, protocol\_name recepiant, protocol\_message message, protocol\_name sender, protocol\_message message\_header}) \]

It contains default code for handling messages received by the oversap of the protocol. Its parameters are the type of message, the recipient of the message, the message string, the sender of the message and the message header.

\[ \text{DefaultUnderSapMessageHandler}(\text{int message\_id, protocol\_name recepiant, protocol\_name sender, protocol\_message message}) \]

It contains default code for handling messages received by the undersap of the protocol. Its parameters are the type of message, the recipient of the message, the message string, the sender of the message and the message.

\[ \text{GetMessageForQueing}(\text{int message\_id, protocol\_name recepiant, protocol\_message message, protocol\_name sender}) \text{returns protocol\_message} \]

It is used to format the message so that it can be queued. Its parameters are the type of message, the recipient of the message, the message string and the sender of the message. It returns the message string that will be queued.

\[ \text{IsQueueEmpty}(\text{bool is\_send}) \text{returns bool is\_empty} \]

It is used to determine if the specified queue is empty. Its only parameter is a boolean value that specifies the queue to check.
GetMessageFromQueue(bool is_send) returns protocol_message_details
message_details

It is used to remove a message from the queue. Its only parameter is a boolean value that specifies the queue to check. It returns the message as a record containing the message details.

SendMessage(bool from_over_sap, cap Sap sap, protocol_message_details message_details)

It is used to send the message to the next protocol. Its parameters are a boolean value indicating if the message to be delivered had arrived from the over sap object of the router, a capability to the sap object of the router and a record variable containing details of the message to be sent.

5.13 – Sizer

This resource extends the router resource. It is an example of a protocol that can be implemented using our library. It performs the same operation as the Sizer given in the PhD dissertation upon which our thesis is based. It contains operations for handling messages received on the over sap and under sap channels of its sap object. These operations simply call the default operations provided by the router resources, which are described above. The sizer contains two methods for servicing the queues, which are periodically called. The method for servicing the send queue checks the length of the message and if it is less than a certain number, sends it through its first under sap object. Otherwise the message is sent through its second under sap object. The method for servicing the deliver queue simply sends the message through the over sap object. The under sap channels may receive messages from multiple lower layer protocols. The sizer also contains the operation GetMessageHeader, which creates and returns the sizer protocol header and the operation GetFieldPosition which returns the position in the header of a certain field (specified by name). It implements the following functions defined in its base resources: InitialiseProtocol, OverSapMessageHandler, UnderSapMessageHandler, GetMessageHeader, and GetFieldPosition. Description of the functions implemented by the Sizer resource is as follows:

SendQueueHandler()

It is used to handle messages that have been placed in its send queue. It is called periodically within a separate process that is started in the router protocol.

DeliverQueueHandler()

It is used to handle messages that have been placed in its deliver queue. It is called periodically within a separate process that is started in the router protocol.

5.14 – Multiplexor
This resource extends the Protocol resource. It contains code that is common to all multiplexor protocols like Fcfs. It is very similar to the Router resource. E.g. it contains methods for returning capability to an array of oversap objects and the undersap object. It also contains operations that implement default oversap and undersap message handling code. The default message handling code provided by the multiplexor is identical to that of the Router resource. The Multiplexor resource also uses two processes to service the send and deliver queues. Messages in these queues are handled by operations provided by resources such as Fcfs that extend Multiplexor. The constructor of the multiplexor resource takes the following parameters: capabilities to the send and deliver queue handlers. Description of the functions implemented by the Multiplexor resource is as follows:

GetOverSaps()returns cap Sap protocol_over_sap

It is used to return the oversap objects.

GetUnderSap()returns cap Sap protocol_under_saps[MAX_PROTOCOL_COUNT]

It is used to return the undersap object.

DefaultOverSapMessageHandler(int message_id, protocol_name recepiant, protocol_message message, protocol_name sender, protocol_message message_header)

It contains default code for handling messages received by the oversap of the protocol. Its parameters are the type of message, the recipient of the message, the message string, the sender of the message and the message header.

DefaultUnderSapMessageHandler(int message_id, protocol_name recepiant, protocol_name sender, protocol_message message)

It contains default code for handling messages received by the undersap of the protocol. Its parameters are the type of message, the recipient of the message, the message string, the sender of the message and the message header.

GetMessageForQueing(int message_id, protocol_name recepiant, protocol_message message, protocol_name sender)returns protocol_message

It is used to format the message so that it can be queued. Its parameters are the type of message, the recipient of the message, the message string and the sender of the message. It returns the message string that will be queued.

IsQueueEmpty(bool is_send)returns bool is_empty

It is used to determine if the specified queue is empty. Its only parameter is a boolean value that specifies the queue to check.
GetMessageFromQueue(bool is_send) returns protocol_message_details message_details

It is used to remove a message from the queue. Its only parameter is a boolean value that specifies the queue to check. It returns the message as a record containing the message details.

SendMessage(bool from_over_sap, cap Sap sap, protocol_message_details message_details)

It is used to send the message to the next protocol. Its parameters are a boolean value indicating if the message to be delivered had arrived from the over sap object of the router, a capability to the sap object of the router and a record variable containing details of the message to be sent.

5.15 - Fcfs (first come first served)

This resource extends the Multiplexor resource. It is an example of a protocol that can be implemented using our library. It performs the same operation as the Fcfs given in the PhD dissertation upon which our thesis is based. It contains operations for handling messages received on the over sap and under sap channels of its sap object. These operations simply call the default operations provided by the multiplexor resource, which are described above. The fcfs contains two methods for servicing the queues, which are periodically called. Both these methods simply forward the messages in the queue on a first come first served bases to the next protocol. The over sap channel may receive messages from multiple higher layer protocols. The fcfs also contains the operation GetMessageHeader, which creates and returns the sizer protocol header and the operation GetFieldPosition which returns the position in the header of a certain field (specified by name). It implements the following functions defined in its base resources: InitializeProtocol, OverSapMessageHandler, UnderSapMessageHandler, GetMessageHeader, and GetFieldPosition. It implements the following functions:

SendQueueHandler()

It is used to handle messages that have been placed in its send queue. It is called periodically within a separate process that is started in the router protocol.

DeliverQueueHandler()

It is used to handle messages that have been placed in its deliver queue. It is called periodically within a separate process that is started in the router protocol.

5.16 - How to use the protocol library

To use the protocol library, the protocol implementer must implement a resource that extends the either the protocol resource or one of the three shape based resources. This resource must implement an operation called InitializeProtocol, which takes as parameters: the
name of the protocol links of the protocol, sap object of the protocol and a capability to an operation that takes as argument the name of a header field and returns its position in the header. The InitialiseProtocol operation must create an instance of each resource that it extends. For each such instance it should invoke the InitialiseProtocol operation. The protocol should also implement two operations for handling messages received from the oversap and undersap channels of the protocols sap object. If the protocol extends the router or the multiplexor protocols then it must provide two additional methods called SendQueueHandler and DeliverQueueHandler for handling the send and deliver queues respectively.

5.17 – Test application

To test our library we developed a simple protocol stack consisting of the router connected to two sequencers which in turn are connected to a multiplexor. The protocol stack was used to exchange message between two applications. Each application reads a text file and sends its contents one line at a time to the other application. The receive operation was set to blocking, but could also have been set to non blocking.

Image2 below shows the relation between different resources that make up the protocol stack. The protocol stack consists of the sizer protocol followed by two sequencer protocols, which are followed by the multiplexor protocol.
Conclusions and Suggestions

The outcome of the thesis is a library for building protocol stacks that can be used by distributed applications to communicate with each other. The library is organized in shapes that provide a high level of abstraction which make it easy to create protocols. The programmer does not have to be concerned with implementing standard protocol features such as flow control and protocol multiplexing. A new protocol can be created by extending any of the given resources of the protocol libraries. These protocols can then be used for communication by distributed applications.

However, our approach does have a few limitations. The library is written in *mpd*, a language that has limited support. There are no support forums or well-known users available for *mpd*. The library is restricted strictly for academic use as *mpd* is not a commercial language. The protocols created can only be used only inside distributed applications and different applications cannot communicate using this library. It is also not very portable as it is only designed to run on UNIX.

It should be possible for two different applications to communicate via our protocol by using RPC (Remote Procedure Call).
References


