

Building a Reliable Multicast Service based on Composite Protocols

By

Sandeep Subramaniam

B.E. (Electronics and Communication Engineering),
Anna University, Chennai, India, 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Professor in charge

Committee Members

Date thesis accepted

To My Parents

Acknowledgements

My sincere and heartfelt thanks to Dr.Gary Minden, my faculty advisor and committee chair-member for all his guidance, support, encouragement and for funding me throughout the course of my thesis work. His ideas, inputs and feedback were very valuable at every stage of my thesis and helped me immensely in my research and thesis work. I would like to thank Dr.Joseph Evans and Dr.Perry Alexander for helping me during the course of my thesis, for reviewing my thesis document and for serving on my committee. Special thanks to Ed Komp for supervising the IANS Composite Protocols project on the whole, for numerous design related discussions that I have had with him, for showing constant interest and enthusiasm in my work and for reviewing my thesis.

I would also like to thank my project team members Magesh, Steve, Srujana and Shyang for all their co-operation and support and helping me in my work

I would also like to thank ITTC staff, all my friends at ITTC and KU for their wonderful company and making my stay in Lawrence very memorable.

Special thanks to my parents for giving me this great opportunity , for all their blessings, love, support and motivation throughout my education.

Abstract

Active networking allows end-users of the network to define, implement and deploy their own customized protocols and services without the need of network-wide standardization. Composite protocols provide an approach for rapid deployment of correct and flexible protocol stacks. A composite protocol is a collection of single-function protocol components arranged in an orderly manner providing a network communications capability. A network service primarily consists of two or more co-operating composite protocols. This thesis demonstrates the feasibility of applying the composite protocol approach to design, specify and implement wider range of protocols and network services. Reliable multicast service is chosen as a case study as it consists of protocols for reliable replication of data in the network, multicast routing and group membership protocols. One of the main challenges in designing a network service is to handle interaction between multiple protocol stacks. In this thesis, we propose a solution for such co-operating protocols to communicate with each other by means of global memory objects. Global memory features, initialization, implementation and their interaction with composite protocol stacks are discussed. The functionality of all the individual protocol components and global memory objects in the multicast service that were implemented are also discussed in detail.

This thesis finally reports results of various functionality and performance tests conducted on a medium-sized test network. The performance of the multicast service has been compared with Linux IP Multicast implementation.

Table of Contents

1. INTRODUCTION.....	1
1.1 ACTIVE NETWORKS.....	1
1.2 MOTIVATION FOR COMPOSITE PROTOCOLS.....	1
1.3 PROTOCOL COMPONENTS, COMPOSITE PROTOCOL, COMPOSABLE SERVICES.....	2
1.4 CROSS-PROTOCOL COMMUNICATION AND GLOBAL MEMORY.....	3
1.5 THESIS ORGANIZATION.....	4
2. COMPOSITE PROTOCOL FRAMEWORK.....	6
2.1 FRAMEWORK MODULES.....	6
2.2 FRAMEWORK FUNCTIONS.....	8
2.3 FRAMEWORK MEMORY MODEL.....	9
3. DESIGN OF COMPOSABLE MULTICAST SERVICE.....	11
3.1 STEPS IN BUILDING A COMPOSABLE SERVICE.....	12
3.1.1 <i>Decomposition:</i>	12
3.1.2 <i>Specification of protocol components</i>	12
3.1.3 <i>Building the stacks</i>	13
3.1.4 <i>Deployment - Placing the stacks in the network</i>	16
3.2 INTRA-STACK COMMUNICATION.....	17
3.3 INTER-STACK COMMUNICATION AND GLOBAL MEMORY.....	19
3.3.1 <i>Global Memory features:</i>	21
3.3.2 <i>Implementing global memory:</i>	22
3.3.3 <i>Initialization</i>	24
3.3.4 <i>Independence</i>	25
4. IMPLEMENTATION.....	26
4.1 THE FRAMEWORK.....	26
4.1.1 <i>Reasons for choosing Ensemble</i>	26

4.1.2	<i>State Machine Executor in Ensemble</i>	27
4.1.3	<i>Mapping of framework functions</i>	29
4.1.4	<i>Timer implementation</i>	30
4.2	THE POINT-TO-MULTIPOINT MULTICAST MODEL:	32
4.3	GLOBAL MEMORY USING SHARED MEMORY MODEL:	33
4.3.1	<i>Shared memory</i> :	33
4.3.2	<i>Creating a global memory object</i>	35
4.3.2.1	Specification of read/write functional interface using CamIIDL	36
4.3.2.2	Implement the functional interface using Shared Memory system calls	38
4.3.2.3	Handling concurrency issues using semaphores:	40
4.3.2.4	Dynamically linking shared global objects with the stacks	44
4.3.3	<i>Global Memory Initialization</i>	44
4.3.4	<i>Multicast Service Objects and their Functional Interface</i>	47
4.3.4.1	Neighbor Table	47
4.3.4.2	Routing Table	48
4.3.4.3	Source Tree	49
4.3.4.4	Prune Table	50
4.3.4.5	Group Table	51
4.3.5	<i>Protocol Interactions Through Global Memory</i>	52
4.4	COMPONENT IMPLEMENTATION	55
4.4.1	<i>MULTICAST DATA STACK components</i> :	55
4.4.1.1	Multicast forwarding	55
4.4.1.2	Replicator	59
4.4.1.3	Multicast in-order component	61
4.4.1.4	End-to-End Reliable (without NACK implosion prevention)	63
4.4.1.5	Reliable with NACK- implosion prevention	68
4.4.2	<i>MULTICAST ROUTING STACK components</i> :	70
4.4.2.1	Neighbor Discovery	70

4.4.2.2	Route Exchange	73
4.4.2.3	Spanning Tree	76
4.4.2.4	Pruning.....	78
4.4.2.5	Grafting.....	81
4.4.3	<i>GROUP MEMBERSHIP STACK components:</i>	83
4.4.3.1	Join/Leave component with its control interface.....	83
5.	TESTING AND PERFORMANCE	87
5.1	FUNCTIONALITY TESTING	87
5.2	PERFORMANCE TESTING	95
5.2.1	<i>Test 1: Measurement of stack latencies</i>	98
5.2.2	<i>Test 2: Measurement of Component Transmit and Receive Latencies</i>	100
5.2.3	<i>Test 3: Measurement of one-way latency</i>	101
5.2.4	<i>Test 4: Measurement of end-to-end throughput</i>	103
5.2.5	<i>Test 5: Measurement of throughput for reliable multicast</i>	105
5.2.6	<i>Test 6: Measurement of join and leave latency</i>	107
5.3	COMPARISON WITH LINUX IP MULTICAST	109
6.	SUMMARY AND FUTURE WORK.....	113
6.1	FUTURE WORK.....	114
	BIBLIOGRAPHY	115

LIST OF FIGURES

FIGURE 1: COMPOSITE PROTOCOL FRAMEWORK	6
FIGURE 2: MULTICAST SERVICE STACKS	13
FIGURE 3: DEPLOYMENT OF STACKS	16
FIGURE 4: GLOBAL MEMORY OBJECTS - FUNCTIONAL INTERFACE	19
FIGURE 5: STATE MACHINE EXECUTOR	28
FIGURE 6: TEST NETWORK	87
FIGURE 7: VARIATION OF STACK LATENCY WITH MESSAGE SIZE	98
FIGURE 8: 6-HOP TEST NETWORK	101
FIGURE 9: VARIATION OF ONE-WAY LATENCY WITH MESSAGE SIZE	102
FIGURE 10: VARIATION OF THROUGHPUT WITH MESSAGE SIZE	104
FIGURE 11: VARIATION OF RELIABLE MULTICAST THROUGHPUT WITH ERROR RATE	106
FIGURE 12: PRUNE DEPTH OF A MULTICAST TREE	108

List Of Tables

TABLE 1: FRAMEWORK FUNCTIONS - CORRESPONDING ENSEMBLE EVENTS	29
TABLE 2: NEIGHBOR TABLE- FUNCTIONAL INTERFACE	48
TABLE 3: ROUTING TABLE – FUNCTIONAL INTERFACE	49
TABLE 4: SOURCE TREE – FUNCTIONAL INTERFACE	49
TABLE 5: PRUNE TABLE – FUNCTIONAL INTERFACE	51
TABLE 6: GROUP TABLE – FUNCTIONAL INTERFACE	52
TABLE 7: VARIATION OF STACK LATENCY WITH MESSAGE SIZE	99
TABLE 8: COMPONENT LATENCIES AT SENDER.....	100
TABLE 9: COMPONENT LATENCIES AT RECEIVER.....	100
TABLE 10: VARIATION OF END-TO-END LATENCY WITH HOPS	103
TABLE 11: VARIATION OF THROUGHPUT WITH MESSAGE SIZE.....	105
TABLE 12: VARIATION OF RELIABLE MULTICAST THROUGHPUT WITH ERROR RATE	107
TABLE 13: VARIATION OF JOIN LATENCY WITH PRUNE-DEPTH	109
TABLE 14: COMPARISON WITH LINUX IP MULTICAST	111

1. Introduction

1.1 *Active Networks*

Traditional data networks passively transfer data from one end of the network to another. Routers in conventional networks just forward user data by processing packet headers. This network property was changed with the advent of active networking. Nodes in an active network called active nodes can, not only forward user data but can also perform customized computations on data flowing through them. Active networking enabled users to inject their customized code or programs within the network thereby tailoring the network to meet user and application specific needs. This mechanism allows introduction of customized network protocols and services without the need of network-wide standardization unlike conventional rigid network implementations. Several active networking architectures like ANTS[1], PLAN[2], Magician[3] etc, have been developed to deploy services need by an application on intermediate nodes of the network. This thesis focuses not on deployment mechanisms but on developing a framework to build user-specific and customized network services that are not only easy to design, test and deploy but are also formally correct in their property and behavior. Composite protocols for innovative active services [4] is a modular approach for specifying and implementing network protocols providing such a framework.

1.2 *Motivation for composite protocols*

Traditional monolithic protocol implementations following the OSI model [5] are modular in design employing the layering principle, with each layer providing a service

to the layer above it. However, layered implementations were found to perform poorly as compared to monolithic implementations, so modularity was compromised for efficiency and performance reasons. Protocol correctness was not considered to be an important aspect in its design. Monolithic implementation of protocols made it difficult to analyze and assert properties about protocol behavior and correctness. The OSI model and the TCP/IP architecture embedded multiple functionality in a single layer. The network layer, IP handled routing, fragmentation etc, TCP handled reliable delivery, sequential delivery, flow-control etc. This architecture does not provide the much-needed flexibility to the user/application of choosing a protocol with a collection of properties. The user/application has to choose TCP even if it wants only its reliable-delivery service and not any of its other services. The idea of code reuse is a common principle being used in software engineering for a long time, but it has not been used yet in protocol implementations. All these aspects of existing protocol implementations motivate the need for design and development of composite protocols.

1.3 Protocol components, composite protocol, composable services

Reliable-delivery, sequential delivery, error checking, some form of routing, authentication, request/reply protocols are some of the common properties or functions which are used in the existing protocols. Any new protocol developed may also demand the use of some of these functions. We call such single-functional protocol modules, protocol components. A group of such protocol components collected and connected together by means of a composition operator constitutes a protocol. For example, TTL, Fragmentation, Header Checksum, Forwarding and Addressing are protocol components, which composed give an IP protocol. Though many forms of composition exist, the most

common form of composition and the one used in our implementation is a linear composition. A collection of two or more cooperating protocols is called a service. Multicast is an example of such a service. Multicast consists of protocols for group membership and management, multicast routing and spanning trees, tunneling and reliable replication of multicast data.

Traditional IP-based multicast network services typically consist of multicast routing protocols like DVMRP[6], MOSPF[7] or PIM[8] and group-management protocols like IGMP[9] in operation. These traditional multicast protocols are decomposed into individual and independent smaller units called protocol components, each performing only a single-function. Each protocol component is completely specified in terms of the Augmented State Machine model, memory requirements and properties [10]. This thesis describes how a component based multicast service is built by linearly stacking protocol components into three different protocol stacks viz. a DVMRP like multicast routing stack for creating and managing multicast routing tables and spanning trees, an IGMP like group-management stack for managing group-memberships and a multicast-traffic delivery stack for reliable transmission of application data.

1.4 Cross-Protocol Communication and Global Memory

The definition and implementation of network services introduce new issues into the active networking environment. Key issues are cross protocol communication among the protocols contributing the service and to the protocol(s) using the service, maintaining independence between active network protocols which use a service and the protocols providing the service, while allowing them to communicate effectively among others.

This thesis focuses on addressing the issue of supporting cross protocol communication that minimizes the degree of interdependence between cooperating protocols while building a network service. To support communication between independent protocols stacks, we require a memory with both scope and extent greater than any single protocol that accesses it. We use the category, global memory, for these memory units. Due to these scope and extent requirements, a global memory must exist independently of any specific protocol. We define an active global memory object for each unit of global memory required in a system. This active object is responsible for initialization and maintenance of the shared information. Any protocol component needing access to the shared information, must contact the corresponding global memory object. Thus global memory objects provide a mechanism for exchange of information between protocol stacks and aid in the development of complex network services.

1.5 Thesis Organization

The rest of this document is organized as follows. Chapter 2 summarizes the salient features of the composite protocol framework. Chapter 3 deals with the detailed design of building a composable service using composite protocol stacks and global memory objects with reliable multicast service as an example. Identification of components through decomposition, specification, composition using linear-stacking and deployment of stacks are described. Also describes the design issues related to inter-stack communication and global memory. Chapter 4 is dedicated to implementation. Reasons for choosing Ensemble[11] as a base framework for implementation are listed. Extensions and modifications made to Ensemble to support our framework, including framework functions, events and timers are discussed. It then describes in detail the

shared memory implementation of global memory, lists the functional interfaces provided by the various global memory objects in the multicast service implementation and how the protocols of the service interact through global memory. Finally, the working of each component used in the service is explained. Chapter 5 on testing, reports results from various experiments conducted and tests performed to confirm the functionality of multicast service across the network. Several network performance tests were also performed. The performance of the composite multicast service is also compared to Linux IP Multicast on a medium sized test network. Chapter 6 summarizes the results of this thesis and suggests enhancements and future work

2. COMPOSITE PROTOCOL FRAMEWORK

2.1 Framework modules

This section describes the various modules of the composite protocol framework

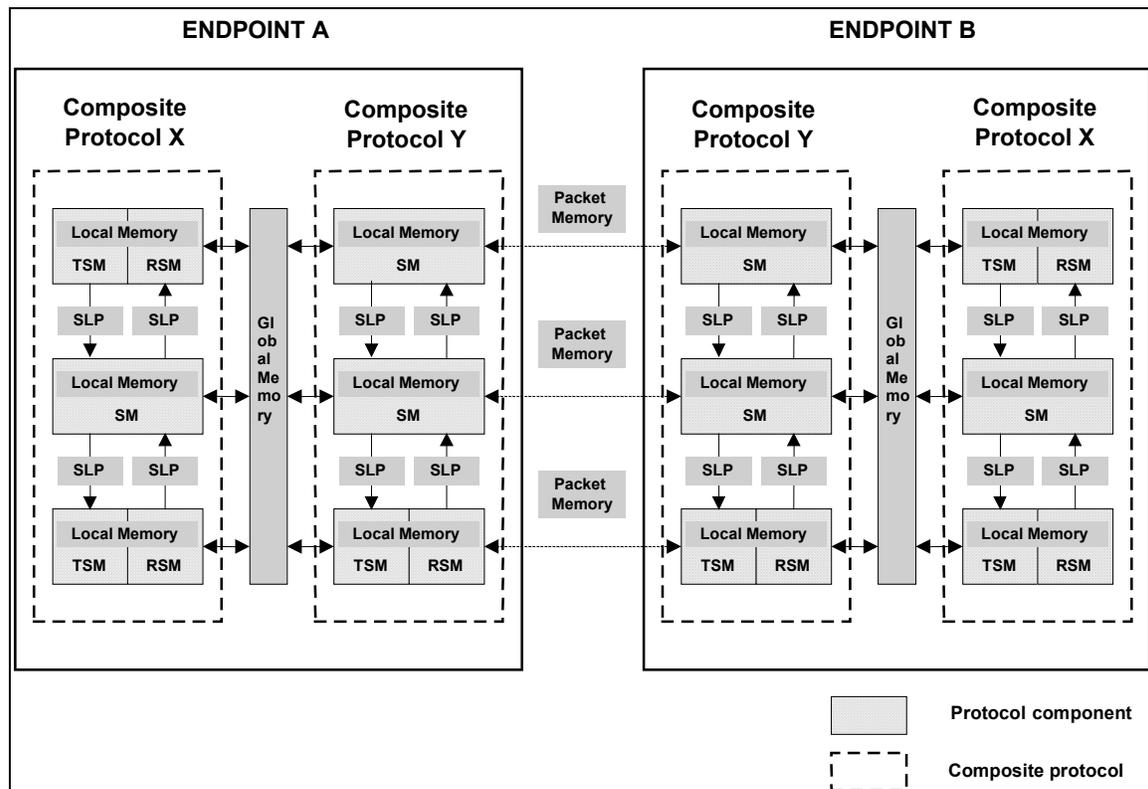


Figure 1: Composite Protocol Framework

The composite protocol framework provides the following to the user:

- A template to formally specify the individual components using AFSMs.
- A composition method to create composite protocols stacks from components.
- A mechanism to construct a service from protocol components.
- Support for dynamic linking of components and switching of protocols on the fly, enabling users to add, remove or re-order components in a stack.

- Support for intra-stack and inter-stack communication between components.

We shall now briefly describe the individual parts that make up the composite protocol framework.

Protocol components are the building blocks of a composable protocol service. Each protocol component implements a single-function either by operating on application data or by independently providing a specific functionality. The former is termed as data-oriented components and the latter control-oriented components. In both cases, peer-to-peer communication between components over the network is necessary. Each protocol component is specified in terms of two Augmented Finite State Machines (AFSMs) viz. Transmit State Machine (TSM) and Receive State Machine (RSM) on the sending and receiving side of the communication channel.

An **AFSM** consists of a finite set of states with a finite set of transitions between one state to another. Each transition is defined by the current-state, next-state, an event, a guard expression, action and local memory update functions. Events trigger a transition from one state to another. Guards are boolean expressions that conditionalize the transition from one state to another. A transition is activated by its corresponding event only if the guard expression evaluates to true. Action functions describe the response of the protocol component to the associated event. They typically consist of executing any one of the well-defined set of framework functions.

Each component has well-defined data interfaces for transmitting and receiving data. On transmission, packets are accepted from a higher layer/component, operated upon if necessary and sent to a lower component after adding a header. On reception packets are accepted from a lower component, its corresponding header stripped and operated upon

and sent to the upper component. The component also has a control interface for handling initialization during start-up and for communication between other components in the stack and the application.

Protocol components are linearly composed in the form of a stack called a composite protocol stack. Stack X and Y in figure 1 are **composite protocol stacks**. Though different methods of component composition are available we have chosen the linear composition method for our framework. Events in the framework are broadly classified into two types viz. data and control events. **Data events** are used for sending and receiving packets either between applications (event generated by application) or between peer-to-peer components (events generated by individual components) e.g. packet arrival event. **Control events** are further sub-divided into two types: stack-wide control events and component specific control events. Stack-wide control events are used for stack initialization, generating timers etc. Component-specific control events are for communication between two or more components in a composite protocol stack.

2.2 Framework Functions

The set of framework functions associated with the above events are given below.

Packet Transfer Functions (associated with data events)

PktSend() - send data from application / higher-level component down the stack.

NewPktSend() - send component's own data to peer.

PktDeliver() - deliver data to application / higher-level component up the stack.

NewPktDeliver() - deliver component's own data to the application

DropPkt() - discard application data

Note: All send functions involve sending data down the stack from the component onto the wire for transmission. All deliver functions involve delivering data up the stack either ultimately to the application or to a component above in the stack.

Buffer Management Functions

KeepPacket() - buffer the data locally

DeliverKeptPacket() - deliver stored data to application/higher-level component

DropKeptPacket() - discard locally stored data

SendKeptPacket() - send buffered application/higher-level data on the wire

Control functions

Stack-wide control events: (timers)

SetTimer() - request a timeout from the framework

ResetTimer () - reset the value of the timer

CancelTimer () - cancel existing timer

Intra-stack communication: (control events)

SendUpControl() - generate a control event and send it up the stack.

SendDownControl () - generate a control event and send it down the stack.

2.3 Framework memory model

We have classified the protocol memory into 4 categories based on its accessibility and scope:

Component-Local Memory: this is internal to the component. Accessible only by the action functions within the TSM and the RSM of the component. They are separately

initialized by at the sending and the receiving sides. E.g.: sliding window buffer in the Reliable Delivery component.

Stack-Local Memory: provides a mechanism for components within a stack to share information. This is accessible to all components in a stack. Since SLPM is associated with an event, the extent of this memory is limited to the life of the event in the stack.

Global Memory: This part of memory is external to a stack and is used for communicating or sharing information between multiple protocol stacks. In our model, global memory access is abstracted through a functional interface for both reading and writing values. Global memory issues shall be discussed in detail under Inter-stack Communication in section 3.

Packet Memory: This memory represents the header added by each component to the data from the application / higher-level component. Each component independently defines its own packet memory. It is accessible only by this component at the sending and receiving ends. All other components have an opaque view of this memory as a read-only linear sequence of bytes.

3. DESIGN OF COMPOSABLE MULTICAST SERVICE

This section describes the various steps involved in building a composable service using our framework with multicast service as a case study. Also discusses intra-stack and inter-stack communication. Multicast is an excellent example of a network service, which is made up of several cooperating protocols. Any form of multicast service would require functions for multicast routing, creation of spanning trees, reliable replication of multicast data and joining/leaving multicast groups. IP Multicast is a collection of multicast routing protocols like DVMRP, MOSPF, PIM, reliable multicast protocols like RMTP[12] and group management protocols like IGMP working in tandem with IP for best-effort multicast delivery. The reason for studying multicast service is that it combines data and control-oriented protocols. TCP and IP are data-oriented protocols, while routing protocols like RIP[13], OSPF[14], DVMRP and group-management protocols like IGMP are control oriented (belong to the control-plane). It should be noted that protocol components that we specify and implement are not in accordance with any Internet standards like RFCs and internet-drafts for DVRMP, RMTP, IGMPv1, and IGMPv2. What we are interested is the basic functionality of these protocols. Only a subset of the standard functionality is specified and implemented. Also, we assume that the reader has a basic understanding how IP multicast and other protocols like DVMRP and IGMP work in general. We now describe the various steps in building a composable service using our framework.

3.1 Steps in building a composable service

3.1.1 Decomposition:

Decomposition is the initial process of identifying the key functional protocol components in a monolithic implementation of a protocol.

For multicast service, we decomposed the monolithic DVMRP protocol into the following protocol components: Neighbor Discovery, Route Exchange, Spanning Tree, Pruning and Grafting. The IGMP protocol was decomposed into the following components: Join/Leave and Query/Report. Other components that form part of the data stack include Multicast Forwarding, Unicast Forwarding, variants of Reliable Multicast like with/without ACK implosion prevention, hop-to-hop reliable, Multicast Inorder, Replicator. These components are not a result of direct decomposition from any other protocol.

3.1.2 Specification of protocol components

Once all the individual components are identified, the next step is to specify each of these components using AFSMs as described in [4]. Each component is represented by a TSM and a RSM, the set of events (data and control) that can invoke this component, its memory requirements: local, stack-local, global and packet memory along with its properties and some assumptions. The individual functionality of each protocol component is described later in section 4.4. While specifying these components, care should be taken to ensure that each protocol component performs only a single-function and is totally independent of other components. Achieving total independence is only an ideal case, practically some minor amount of dependence on other protocol components may be required. Also, it may not be possible to represent each decomposed protocol in

terms of state machines or in accordance with the composite protocol specifications. In such cases the decomposed protocol may have to be either merged with other protocols or re-specified appropriately so that they meet the specifications. E.g. A decomposed protocol having no header information (bits-on-the-wire) can always be merged with another protocol. However, we shall elicit on the individual functionality of each protocol component in section 4.4.

3.1.3 Building the stacks

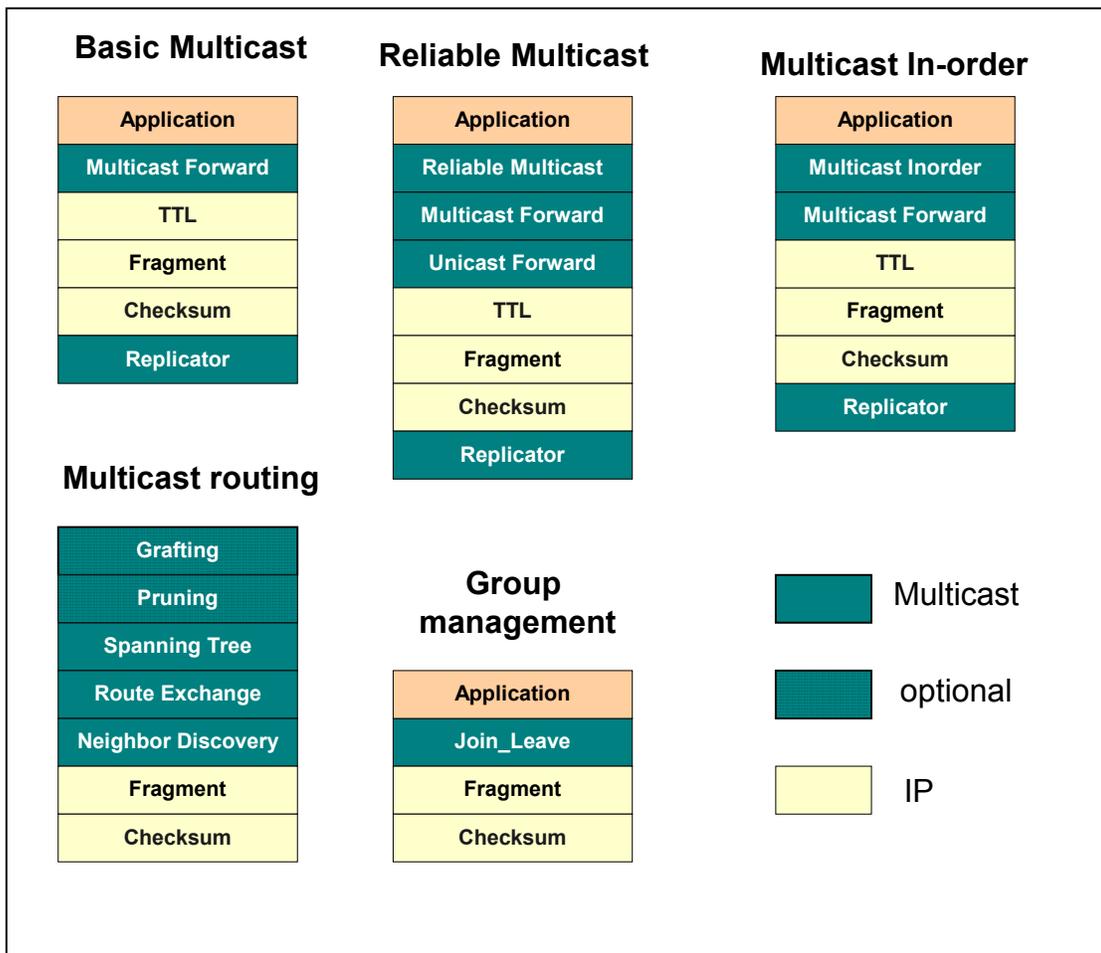


Figure 2: Multicast Service Stacks

Once all the individual protocol components are specified, related components are grouped in protocol stacks called composite protocol stacks. The composable service is just the collection of these stacks and global memory objects (described later in this section). Multicast service is a collection of three stacks viz. Multicast routing stack, Group Management stack and Multicast data/traffic stack and the global memory objects. We have decided to compose stacks using the linear stacking approach. In this approach, while composing stacks, the order of stacking can play an important role depending on whether the components being stacked are property oriented or control oriented.

A property based component is one which provides a well defined property or functionality to the component/application above it by adding headers to application data. Typical examples are TCP components like Reliable delivery, in-order delivery, or IP components like TTL, Fragment. Control based components do not provide any property to the component above, though they implement a separate function on their own. They mainly exchange peer-to-peer messages only.

We find lot of examples of such control components in Neighbor Discovery, Route Exchange etc. When these 2 components are stacked up with Neighbor Discovery on top of Route Exchange, it should be noted that the Route Exchange component does not operate or perform any computation on data sent by Neighbor Discovery. It merely passes it down without appending its header. These types of components are responsible for creating, managing global data structures, which may be accessed by other stacks. They may or may not interact with each other. Interaction if present is generally through control events (Intra Stack communication). Relative ordering of control oriented components does not affect the overall general functioning of the stack. They make

however affect stack performance. It may be a good idea to consider placing the component that exchanges peer-to-peer messages most frequently, bottom-most in the stack and that which exchanges messages least frequently, top-most the stack. Placing the component as low in the stack as possible shall minimize end-to-end delay and also reduce extra overhead (caused by dummy headers) added by other components. Placing the Neighbor Discovery component low in the stack, and Pruning/Grafting high in the stack may be a good stacking arrangement.

Property based components impose a strict ordering on components above/below it. E.g. If reliability is needed hop-to-hop, the reliable component has to be placed below the multicast forwarding component, where-as if reliability is needed end-to-end, it has to be placed above the multicast forwarding component.

The framework offers the much-needed flexibility in this regard. Components can be easily added to stacks, removed from stacks or even re-ordered within stacks rendering different protocol stack properties to the user. Thus , building stacks with an optimal ordering is an important and challenging task in building a service.

3.1.4 Deployment - Placing the stacks in the network

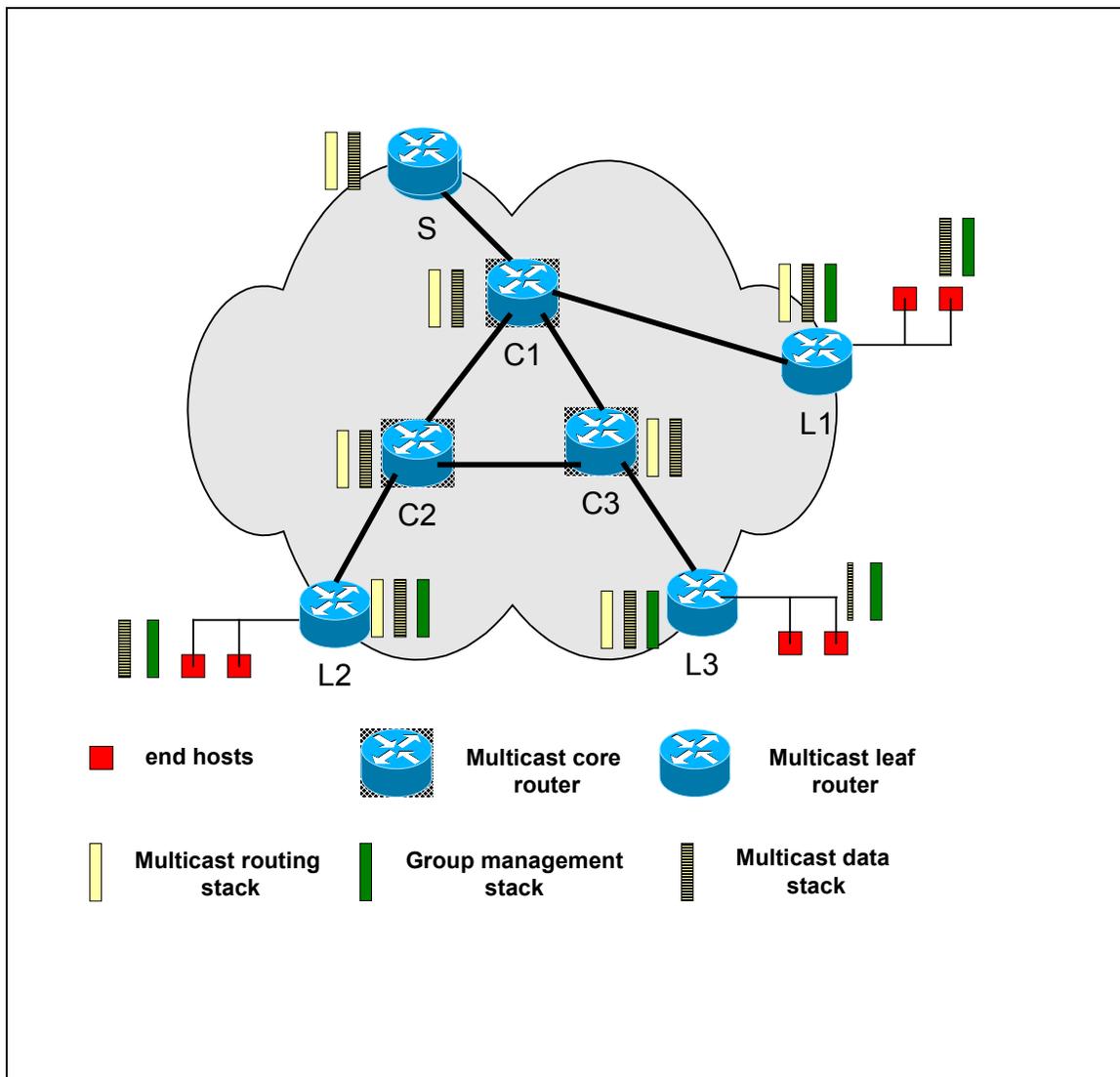


Figure 3: Deployment of Stacks

This thesis focuses mainly on service composition and not on automatic deployment issues in an active network. Automatic deployment of composite protocol stacks and then running these stacks on an Active Node is a subject of future research.

In this thesis, the composite protocol stacks are manually deployed on normal nodes (non-Active nodes).

Figure 3 shows an example multicast network with the different stacks deployed at various nodes:

- Multicast Sender: sends multicast data destined for a particular group. Need not be a part of a multicast group to send a multicast packet. Typically attached to a multicast core-router.
- Multicast Core Router: present in the core of the multicast network. They are responsible for creating and managing multicast routing tables and setting up per source group multicast delivery trees.
- Multicast Leaf Router: these are nodes that do not have downstream neighbors and are directly attached to multicast receivers (end-hosts).
- Multicast Receivers: these are end-hosts that have joined a particular group and are entitled to receive multicast traffic destined to that particular group.

Note that both Multicast core routers and Multicast Leaf routers can also be Multicast Receivers and Multicast Senders

3.2 Intra-stack Communication

Intra-stack communication refers to communication between two components in a stack or communication between the application and a protocol component in the composite protocol stack. This form of communication is handled by use of control events in the framework and by extending components to provide control interfaces. The PIPO (packet-in packet-out) interface is sufficient for data-plane components (property-oriented) as discussed before. E.g. For components like reliable-delivery, checksum, fragment etc, it may be enough to just act and process the packet passed from above.

Each component just adds its own header for payload from above and strips off the corresponding header at the receiving side. This interface will not be sufficient for components that depend on some control information or set of user-level commands from the application. This demands a need for a control interface to enable communication between components or between the application and a component.

The component which implements a control interface offers a service to components above it or to the application and is called the controlled component. The component above this or the application that utilizes the provided service is called the controlling component.

In the multicast service, the JoinLeave component of the GroupMembership stack is an example of a component that makes use of such control events in the framework and is the controlled component. The application which uses its control interface to join/leave multicast groups is the controlling component.

SLPM (Stack-Local Packet Memory) can also be viewed as another form of intra-stack communication in our framework. SLPM is an auxiliary data structure attached to the packet as it is processed by components in the stack. SLPM fields are implemented as (name, value) pairs and a set of framework functions are provided to access SLPM. SLPM is often used to transfer packet information between components. A high-level component can add a field to SLPM that is then read and used by a low-level component. E.g. the next-hop IP address is added to SLPM by the Forward component and is read from SLPM by a lower-level data link component.

Thus intra-stack communication is mainly accomplished by use of control events in the framework and in some cases through use of SLPM.

3.3 Inter-Stack Communication and Global Memory

One of the challenging problems in designing a network service is to identify and address the issue of how different protocols interact with each other. Network services require the cooperation of two or more network protocols; that is they need to share information. In

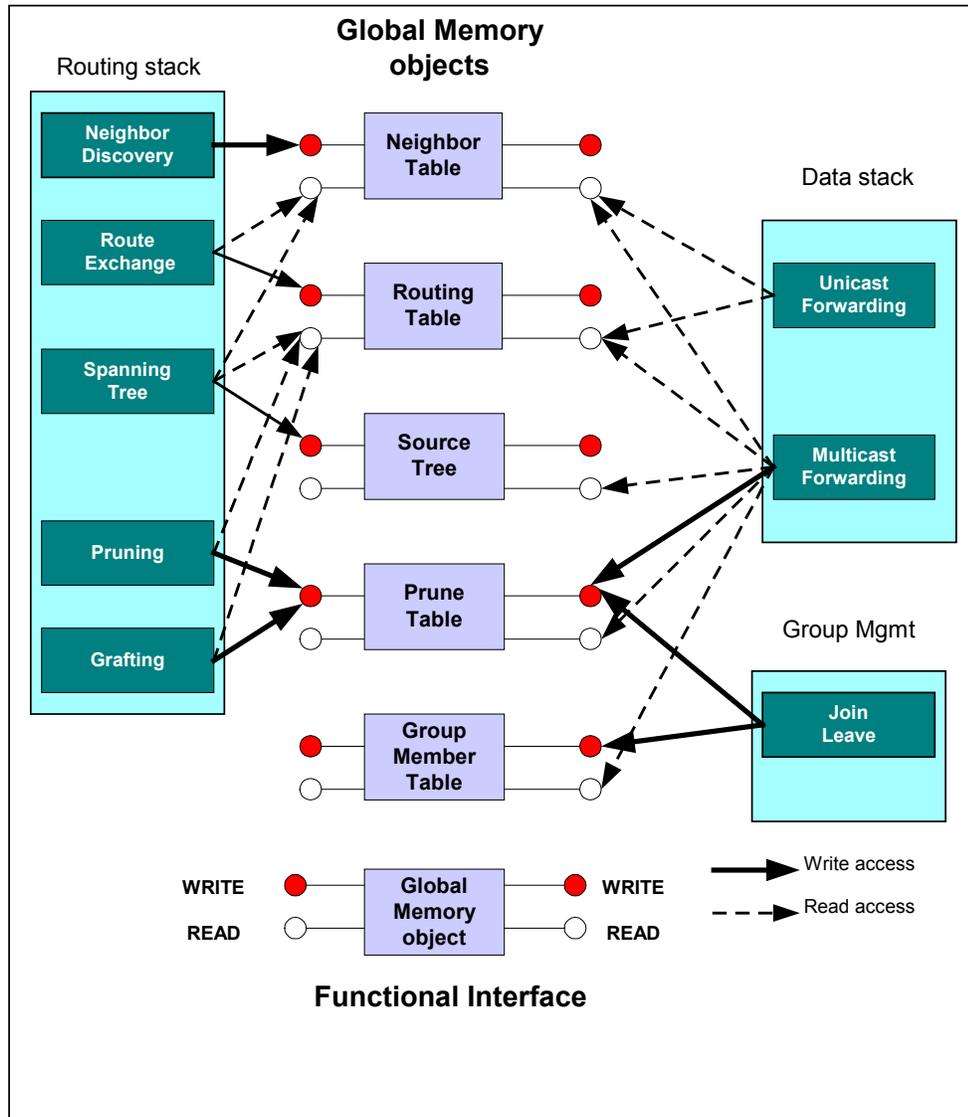


Figure 4: Global Memory Objects - Functional Interface

this section, we will describe our solution to this challenging problem.

Our solution is to generate a global memory object, independent of any protocol that uses it, for the storage of information shared among two or more protocols. The scope and extent of this object must be greater than that of any single protocol, which accesses the information, stored in the global memory object. Access to read / write the contents of the shared information is provided through a functional interface. A protocol component expresses its requirements for access to global memory object(s) by listing the external functions it uses in its implementation. E.g. The RouteExchange component needs a function to write new routes into the Routing Table. So, it would use a function like `addNewRouteEntry (rt_entry)` to add a new route entry to the routing table. The IP forwarding function needs to know the nexthop address for each destination. It would require an external function like `ipaddr getNextHopForDest (dest_addr)` to get the nexthop address. These functions `addNewRouteEntry()` and `getNextHopForDest()` are provided through the write and read functional interface of the global Routing Table object respectively.

Very generally, the global memory object can be regarded as a server, providing access to shared information to its clients, the protocol reading/writing this information. For example, in the TCP/IP world the IP Routing Table is created and maintained by protocols like RIP, OSPF etc. and is accessed by IP while forwarding data packets. In our framework, the routing table is maintained as a global memory object that is external *to* both protocols IP and RIP. We shall now discuss the various features and requirements of global memory in our framework.

3.3.1 Global Memory features:

Functional interface:

In our framework, global memory access is abstracted through a functional interface for both reading and writing data. The functional interface model helps in encapsulating the data and hides the internal representation of the object.

Synchronization:

Protocols can access global memory only through the functional interface, so the use of semaphores and/or any other control mechanisms to provide necessary synchronization are embedded in these functions in a uniform and robust manner. Synchronization is not delegated to the users of the shared object(s). Furthermore, since the interface is truly functional, no pointers are shared, which eliminates any possibility of conflicts from implicit sharing through multiple references to the same object. In a similar manner, implementation of the functional interfaces can apply access-rights controls to limit access to sensitive data. This approach makes protocol interfaces to the global memory are very simple. Complex issues of synchronization and access control are addressed just once in the design and implementation of the global memory object, instead of requiring each protocol that shares the information to incorporate these controls in its implementation. And the solution is much more robust, since the integrity of the shared data cannot be compromised by a single protocol, which does not correctly implement synchronization algorithm.

Extensibility:

The global memory object definition can be extended by adding new functions to its functional interface, to provide services for new protocols developed which use/access information in an existing global memory object. This provides a powerful mechanism for developing new protocols and/or improving existing implementations, while maintaining backward compatibility for previous clients (protocols) that use the global memory object. Previous clients continue to use the existing interfaces while the new protocols use the new extended version.

3.3.2 Implementing global memory:

We now discuss a few approaches to implement global memory.

Process model:

In this model, each global memory object is implemented with a separate process running as a server on each node. Typically, each global memory server is started up during the node initialization sequence. This server process maintains a single internal representation for its global memory object. The server can choose any representation for the data, because this structure is entirely local to the server. The server implements an inter-process communication (IPC) interface according to the functional definition of global memory. Any protocol that accesses a global memory contacts the corresponding server process as a client. Communication between the clients (protocols) and server is limited to the IPC interface advertised by the server process. This implementation strategy is a direct implementation of the abstract model we propose for a global memory object. Unfortunately, the overheads associated with inter-process communication, even

within a single node, may be too large for the performance requirements of network protocol implementations.

Shared-Memory model:

In this model, the data to be shared by multiple stacks is stored in shared memory. The functional interface containing the set of all functions provided by the global memory object is packaged into a dynamic link library (DLL). The protocol stacks, which run as individual processes on a node, will link to the dynamic library defined for the global memory it uses.

Accesses to global memory are simply function invocations in the process image. The actual implementation of the functional interface is entirely opaque to the clients (protocol stacks). The implementation uses operating system calls to access a section of shared memory; so each protocol stack (independent processes) references the same object stored in shared memory. The implementation is responsible for handling synchronization issues, typically using semaphores provided by the operating system in its shared memory interface.

This implementation approach strongly preserves the abstract functional interface we want for global memory. Users of global memory have only an opaque view of it through the functional interface provided by the DLL. Protocol stack implementations remain operating system independent. The implementation of global memory objects, with node local resources, may need to be adapted to the details of shared memory access interface provided by the operating system.

This implementation provides the same abstract view of global memory objects as the server process model, but is significantly more efficient. Global memory access is accomplished through a local function call instead of an inter-process communication.

Node-OS model:

For the highest execution performance, an alternative is to embed global memory objects directly in the operating system on which the protocol stacks run. With this alternative, the operating system (kernel) interface must be expanded to incorporate the functional interface, which defines the global memory object(s). The operating system implicitly operates as the global memory object server. The protocols using the global memory object obtain direct access through the (new) system function calls introduced with the global memory object. This approach is worthy of consideration only for a few special and widely accessed global memory objects, such as the routing table. The solution is vendor/operating system specific. In addition, it requires extensions to the operating system interface. For example, the current TCP/IP implementations use a strategy similar to this (though not employing a pure functional interface) to provide shared access to the routing table.

3.3.3 Initialization

Each global memory is independent of any network protocol, which uses it. From the perspective of a protocol running on a node, the global memory is a "service" provided by the node. Therefore creation of, and initialization of the global memory is a responsibility of the node environment. Dynamic deployment of network services must

determine if the global memory object(s) used by the protocols, which form the service, are already available on the nodes.

Figure 4 illustrates different protocols of the multicast service cooperating by means of global memory objects. *NeighborTable*, *RoutingTable*, *SourceTree*, *PruneTable* and *GroupMemberTable* are all global memory objects that provide a set of read/write functions through their respective functional interfaces. E.g. The *Route Exchange* component of the multicast routing stack writes into global memory using the write interface of the global *RoutingTable* object and the *Multicast Forwarding* component of the multicast data stack reads using the read interface of the object. Each protocol component includes the list of external memory functions it accesses. *getDownStreamNeighborsForSource(src_addr,group_addr)*, *addNewRoute(route_entry)* are typical examples of read and write external functions for the *Route Exchange* component.

3.3.4 Independence

The global memory objects are designed to be mutually independent with each other. E.g. in the above example, the Routing Table does not have any dependencies with the Spanning Tree global memory object and vice versa. The reason is this. A multicast service may need both the global memory objects Routing Table and Spanning Tree, but say another service requires only the services of the Routing Table object; its dependency on Spanning Tree is by design an undesirable feature.

Also the global memory objects are designed so that it can be used across several services. E.g. the Routing Table object can be used in unicast as well as multicast, with possible variations in its set of functional interfaces.

4. IMPLEMENTATION

Ensemble, a group communication system developed primarily by Mark Hayden of Cornell University was used as a base framework for implementation of our composite protocol framework specifications. Extensions and modifications were made to Ensemble to represent each Ensemble layer with the corresponding state machine representation of the component. In this section, we first give reasons on why we chose Ensemble as our implementation framework, then describe briefly the state machine executor built in Ensemble, depict the mapping of our framework functions with Ensemble events and then discuss timer implementation. The features and limitations of the point-to-multipoint multicast model is then described. This is followed by a detailed description of global memory implementation and finally the working of each protocol component that make up the multicast service is explained.

4.1 *The Framework*

4.1.1 Reasons for choosing Ensemble

- Ensemble is written in Ocaml[15], a functional programming language, and dialect of ML[16]. Use of functional programming languages aid in easy formal analysis of code.
- Ensemble uses linear stacking of protocol layers to form a stack, the same composition methodology that our framework demands.
- Event handlers are atomically executed.
- Unbounded message queues between any two layers.

- Provides an uniform interface through its up and down event handlers, thus enabling arbitrary composition of layers to form protocols.
- Provides support for dynamic linking of components and switching of protocols on the fly, enabling users to add or remove components from a stack.

As Ensemble already provided a good base framework for implementing our specification, it was decided to make use of it instead of developing a new framework from scratch. Lot of code necessary for the original group communication to work was removed; only bare essential code was retained. This resulted in a much smaller Ensemble code base.

4.1.2 State Machine Executor in Ensemble

Individual layers that made up an Ensemble stack had no concept of state machines. All layer functionality was implemented as part of their event handlers. With the introduction of state machine representation for each component in our framework, each Ensemble layer was made to internally invoke its corresponding state machine if necessary. A common state machine executor was built for this purpose. Its design is shown in Figure 5. For each component, the pair of state machines TSM and RSM are defined in Ocaml. Each state machine consists of list of states and a set of transitions from each state. Each transition is a defined as a record containing enumerated next-state, current-state value, enumerated event-type, guard function, action function and local-memory update. The state machine executor has common functionality to execute any arbitrary state machine defined as described above. It starts from an initial state, and moves through a set of

states depending on events and guards and executing action and local memory update functions. It also supports synchronous states and transitions.

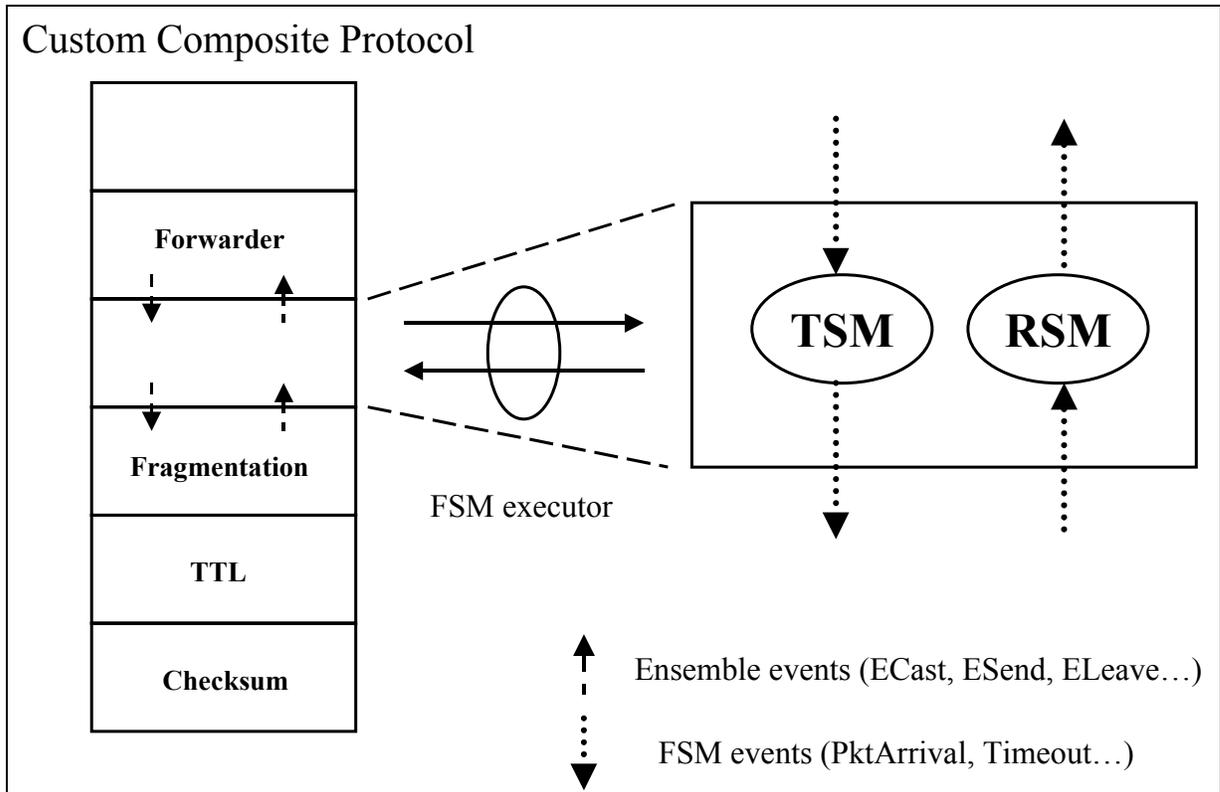


Figure 5: State Machine Executor

For example, for Ensemble down events ESend(Dn) the FSM executor maps to a PktArrival event and invokes the TSM . TSM is then executed as defined. After state machine execution, the FSM executor passes the PktArrival event back to the Ensemble layer through defined framework functions eg. pkt_send.. Similar mapping of events take place for Up Ensemble events, they are directed to the RSM. Certain Ensemble events need not be passed to the FSM if not needed by it. The implementation allows by-pass of such events, which are of no interest to the state machines.

The next-sub section describes the mapping between our framework functions and Ensemble up and down event handlers.

4.1.3 Mapping of framework functions

The table shows the mapping between few of our framework functions (as listed in section 2) and Ensemble UP and DN events

<i>Framework Functions</i>	<i>Ensemble Event</i>
<i>Packet Transfer</i>	
<i>pkt_send(pktmem)</i>	<i>DN (EV, ABV, hdr)</i>
<i>new_pkt_send(pktmem)</i>	<i>DNLM (ev, hdr)</i>
<i>pkt_deliver()</i>	<i>UP (EV, ABV)</i>
<i>new_pkt_deliver(pktpayld)</i>	<i>UP (ev, pktpayld)</i>
<i>Buffer Management</i>	
<i>send_kept_packet(pktpayld)</i>	<i>DN (ev, hdr, pktpayld)</i>
<i>deliver_kept_packet(pktpayld)</i>	<i>UP (ev, pktpayld)</i>

Table 1: Framework Functions - Corresponding Ensemble Events

Words in small letters refer to component generated fields. E.g. In a `new_pkt_deliver()` , the `pktpayld` is generated by the component, whereas in `pkt_deliver()` `ABV` already exists along with the event.

Note the difference between existing and generated fields:

EV: Incoming/Outgoing Ensemble event, ev: component generated Ensemble event.

ABV: Existing packet payload , `pktpayld` : component generated packet payload

hdr : component generated header.

Timer-related framework functions are described in the next sub-section.

4.1.4 Timer implementation

Component specification demands implementation of the following framework functions:

- *set_timer(timer_id: int, timeout: time)*

This function requests a TimerEvent with unique-id timer_id from the framework after time seconds.

- *cancel_timer(timer_id:int)*

This function is used to cancel an existing timer with id timer_id

- *reset_timer(timer_id:int , timeout:time)*

This function is used to reset the value of the timer with id timer_id and request another timer that expires after time seconds.

In the Ensemble system, timers are implemented as Control events flowing up and down the stack. *ETimer* the Ensemble heart-beat timer propagates all the way from the layer bottom upto the topmost layer and is again reflected down the stack. But this timer did not have the notion of a timer-id associated with it, which is needed by our specifications. So to cater to this requirement and to interface our timer framework functions with the Ensemble timer, a Timer Module was built.

The Timer Module is defined as a list of timer objects. Timer object is a record of type *timer_rec*:

```

type timer_dir_type =
  | TimerUp           // Up timer events requested by RSM
  | TimerDn          // Dn timer events requested by TSM
type timer_rec = {
  timeoutid : int;           // the unique timeout-id
  timeout : Time.t;         // time-period for expiry of timer
  timer_direction: timer_dir_type; // direction of requested timer
}

```

The Timer module also provides several functions to perform operations on timer objects.

- *create()* : creates a empty list of timer objects.
- *length()*: returns number of timer objects in list.
- *add(timer_rec, timer_list)*: adds a new timer object *timer_rec* to the existing list *timer_list*.
- *sort(timer_list)*: sorts the list *timer_list* based on the increasing timeout value.
- *lookup(timer_list, time, timer_dir)*: returns list of expired timers from *timer_list* based on values of *time* and *timer_dir*.
- *remove_all(timer_list, timeoutid, timer_dir)*: removes all timer objects from list *timer_list* matching *timeoutid* and *timer_dir*.

The framework creates an empty list of timer objects for each component on startup.

When the component invokes the *set_timer()* framework function as described above, a new timer object is created with appropriate values for *timeoutid*, *timeout* and *timer_direction*. This is added to the existing list of timers and then sorted in an increasing order based on the timeout value. When *set_timer()* is invoked by the TSM the *timer_direction* is set to *TimerDn* and when invoked by the RSM is set to *TimerUp*. A

component can request for any number of timers provided each is requested with a stack-wide *timer-id* value.

When an *ETimer* event reaches an Ensemble layer of a component, its time is compared with the list of time values in the *timer_list* to yield a list of expired timers along with their *timeoutid* values. For each expired timer, a new event called *TimerEvent(timeoutid)* is created and then sent to the appropriate state machine (all UP events are sent to RSM and all DN events are sent to TSM. All expired timers are always removed from the list using *remove_all()*.

This ensures and produces the much-needed Timer Event with the unique timerid for the state machine. *Cancel_timer(timeoutid)* framework function directly removes the corresponding timer with id timer-id from the list , even before its expiry.

It should be noted that Timeout events shall be generated for the same state machine that invoked the *set_timer()* framework function.

4.2 The point-to-multipoint multicast model:

The multicast service implemented is for multicast data flow in a point-to-multipoint multicast network. Here, we have a multicast sender transmitting data on a dynamically established and maintained multicast tree to a group of receivers. Receivers (end-hosts) in this model can only join / leave certain multicast groups, they cannot in-turn, multicast to other group members.

This model is well suited and applicable to situations like streaming video/audio from a server, file downloads etc. This will not be appropriate for video-conferencing types of multicast applications where we need a multipoint-to-multipoint data flow. Note that in

our model, we can have N different multicast senders in the network multicasting data on their respective trees, but each should be viewed as N separate multicast data flows. Receivers in a flow are allowed only to send back unicast data back to the sender e.g. ACK packets.

4.3 Global memory using Shared Memory model:

This section describes the implementation of global memory using the Shared Memory approach. A brief description of Linux shared memory, the kernel data structures and shared memory system calls follows.

4.3.1 Shared memory:

Shared memory is another method of inter-process communication (IPC) whereby 2 or more processes share a single chunk of memory to communicate. Shared memory is described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process and subsequently written to and read from by any number of processes.

Kernel `shmid_ds` structure:

The Linux kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type `shmid_ds`, and is defined in `linux/shm.h` as follows:

```
// One shmid data structure for each shared memory segment in the system.
struct shmid_ds {
    struct ipc_perm shm_perm;           // operation perms
    int shm_segsz;                      // size of segment (bytes)
    time_t shm_atime;                  // last attach time
    time_t shm_dtime;                  // last detach time
    time_t shm_ctime;                  // last change time
    unsigned short shm_cpid;           // pid of creator
    unsigned short shm_lpid;           // pid of last operator
    short shm_nattch;                  // no. of current attaches
    //the following are private
    unsigned short shm_npages;         // size of segment (pages)
    unsigned long *shm_pages;          // array of ptrs to frames -> SHMMAX
    struct vm_area_struct *attaches;    // descriptors for attaches
};
```

Shared memory system calls used:

`shmget()`:

`shmat()`:

`shmdt()`:

Accessing a Shared Memory Segment:

`shmget()` is used to obtain access to a shared memory segment.

Prototype : `int shmget(key_t key, size_t size, int shmflg);`

The `key` value is a access value associated with the semaphore ID.

The `size` argument is the size in bytes of the requested shared memory

The *shmflg* argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment.

Attaching and Detaching a Shared Memory Segment:

shmat() and *shmdt()* are used to attach and detach shared memory segments.

Their prototypes are as follows:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

```
int shmdt(const void *shmaddr);
```

shmat() returns a pointer, *shmaddr*, to the head of the shared segment associated with a valid *shmid*. *shmdt()* detaches the shared memory segment located at the address indicated by *shmaddr*.

4.3.2 Creating a global memory object

The steps in creating a global memory object are as follows:

- Specify read/write functional interface using CamlIDL[15].
- Implement the functions using Linux shared memory system calls.
- Handle synchronization issues for each function by appropriate use of the correct semaphore model (multiple readers and single writer)
- Dynamically link the global object with the stacks at run-time.

We shall describe each of the above steps in sufficient detail with examples from the multicast service objects.

4.3.2.1 Specification of read/write functional interface using CamlIDL

CamlIDL:

Camlidl is a stub code generator and COM binding for Objective Caml.

CamlIDL comprises of two parts :

- a stub-code generator that generates the C stub code required for the Caml/C interface based on an IDL specification.
- a library of functions and tools to import COM components in Caml applications and export Caml code as COM components.

In this implementation we make use of only the stub-code generation feature of CamlIDL. It automates the most tedious task in interfacing C libraries with Caml programs. IDL stands for Interface Description Language , which is a generic term for a family of small languages that have been developed to provide type specifications for libraries written in C and C++. *For more information on CamlIDL refer to [17].*

The IDL file:

A typical IDL file describing a set of read/write functional interface would look like this:

```
struct ntable_entry {
....
};
// write functions :
    void write_ntable([in] struct ntable_entry ntable[], [in] int num);
// read functions :
    int getNeighborForInterface([in] int intf);
    boolean isAddrNeighbor([in] int addr);
    int getInterfaceForNeighbor([in] int nbor);
    void read_ntable([out] struct ntable_entry ntable[20]);
```

The function signature including input/output arguments and return types are completely specified. These functions along with the needed data-structures are saved in a *.idl file*

Generating the stub-code:

The *camlidl* stub code generator is invoked as follows:

```
camlidl [options] file1.idl file2.idl ....
```

for each file *f.idl*, *camlidl* generates the following files:

- A Caml interface file *f.mli* that defines the Caml view of the IDL file. It contains Caml definitions for the types declared in the IDL file, as well as declarations for the functions and the interfaces.
- A Caml implementation file *f.ml* that implements the *f.mli* file
- A C source file *f_stubs.c* that contains the stub functions for converting between C and Caml data representations.
- If the *-header* option is given, a C header file *f.h* containing C declarations for the types declared in the IDL file

Eg: camlidl -header ntable.idl generates the following files:

ntable.mli, ntable.ml, ntable_stubs.c, ntable.h

For the IDL specification as in *ntable.idl* (above), *ntable.ml* and *ntable.mli* would contain:

```
type ntable_entry = {
.....
}
external write_ntable : ntable_entry array -> int -> unit = "camlidl_ntable_write_ntable"
external getNeighborForInterface : int -> int = "camlidl_ntable_getNeighborForInterface"
external isAddrNeighbor : int -> bool = "camlidl_ntable_isAddrNeighbor"
external getInterfaceForNeighbor : int -> int = "camlidl_ntable_getInterfaceForNeighbor"
external read_ntable : unit -> ntable_entry array = "camlidl_ntable_read_ntable"
```

n timer _ stubs.c:

example stub-function:

```
value camlidl_n timer _ getNeighborForInterface(value _v_intf)
{
  int intf; /*in*/
  int _res;
  value _vres;
  intf = Int_val(_v_intf);
  _res = getNeighborForInterface(intf);
  _vres = Val_int(_res);
  return _vres;
}
```

The function *getNeighborForInterface(intf)* has to be implemented by the user in the corresponding header file *n timer .h*

Once all these files are generated, the header file has to be implemented which is described next.

4.3.2.2 Implement the functional interface using Shared Memory system calls

Functions are of two types read/write:

- Write functions write data into shared memory segments
- Read functions read data from shared memory segments

Both write/read functions consists of invoking the following system calls:

- Creating a segment using the *shmget()* system call.
- Attach the process to the segment using the *shmat()* system call.
- Perform either write/read of shared data to/from shared memory segment
- Detach the process from the segment after completion using the *shmdt()* system call

A typical code would look like this:

```
eg : WRITE function:
void write_nstable(struct nstable_entry * nstable, int n) {
// Initialize variables
    key_t key;
    int shmid,shmsize
    struct nstable_entry *shm, *s;
// key value
    key = NTABLE_SHMKEY;
    shmsize = sizeof(struct nstable_entry) * n;
// Create the segment
    if ((shmid = shmget(key,4096,IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }
// key is chosen to be a predefined unique value
// shared memory size is 4096 bytes
// Attach the segment
    if ((shm = shmat(shmid, NULL, 0)) == (struct nstable_entry *) -1) {
        perror("shmat");
        exit(1);
    }
// Write DATA into SHARED MEMORY
// Store the array of structures in shared memory
    s = shm;
    memcpy(s,nstable,shmsize);
// Detach the process from the shared memory segment
    if ((ret = shmdt(shm)) < 0) {
        perror("shmdt");
        exit(1);
    }
}
```

All WRITE functions are implemented in a similar manner. READ functions also have similar structure except that they READ from shared memory segments.

4.3.2.3 Handling concurrency issues using semaphores:

Since we have multiple processes modifying the shared memory segment, it is possible that certain errors could crop up when updates to the segment occur simultaneously. This concurrent access is almost always a problem when you have multiple writers to a shared object. Using semaphores to lock the shared memory segment while a process is writing to it can solve this problem.

It should be noted that the implementation allows multiple readers to READ from shared

Pseudo Code:

Global variables:

```
mutex,db : semaphore := 1      // mutual exclusion semaphores  
readcount : integer := 0
```

READER:

```
p(mutex);  
    readcount++;  
    if (readcount is 1) then p(db);  
v(mutex);  
CRITICAL SECTION READ  
p(mutex);  
    readcount--;  
    if(readcount is 0) then v(db);  
v(mutex);
```

WRITER:

```
p(db);  
CRITICAL SECTION for WRITE  
v(db);
```

memory but allows only a single WRITER to write into shared memory at any particular time. So multiple READERS are ALLOWED but multiple WRITERS are NOT ALLOWED.

The solution used for the **Readers and Writers problem** is shown in the above segment of pseudo-code.

In this solution, on semaphore db we have utmost one reader (all other readers will wait on mutex). But once a reader gets in, all waiting readers can get in ahead of waiting writers. When a writer finishes, if there are waiting readers and writers, either readers or a writer will run.

An example of a Linux semaphore implementation for the above solution to handle concurrency control for shared memory access follows:

Let us consider the use of semaphores for accessing the Neighbor Table write function *write_nhtable()*. Only semaphore related code is shown and discussed.

Explanation of Linux semaphore functions used:

semget(): used to create the semaphore set or grab the existing semaphore set.

usage:

```
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

returns a semaphore identifier associated with the key.

key is a unique identifier that is used by different processes to identify this semaphore set.

nsems argument is the number of semaphores in this semaphore set

semflg argument holds the permissions on the new semaphore set .

For creating a new set, the access permissions is bit-wise ORed with IPC_CREAT.

0 is passed for using the existing set. The semaphores are created during the global memory initialization phase itself. So read/write functions here just access the existing semaphore set.

semop():

prototype :

```
int semop(int semid ,struct sembuf *sops, unsigned int nsops);
```

All operations that set, get, or test-n-set a semaphore this system call. Its functionality is dictated by the structure <struct sembuf> that is passed to it.

```
struct sembuf {
```

```
  ushort sem_num;
```

```
  short sem_op;
```

```
  short sem_flg;
```

```
};
```

sem_num is the number of the semaphore in the set that is to be manipulated.

sem_op is the action to be performed on the semaphore.

It depends on whether *sem_op* is *positive*, *negative* or *zero* as given below :

positive: the value of *sem_op* is added to the semaphore's value. Used in a V() operation.

negative : if the absolute value of *sem_op* is greater than the value of the semaphore, the calling process will block until the value of the semaphore reaches that of the absolute value of *sem_op*. Finally, the absolute value of *sem_op* will be subtracted from the semaphore 's value. This is used in the P() operation on the db semaphore in the above example.

zero : the process will wait until the semaphore reaches 0.

**sops* is a pointer to the *struct sembuf* that is filled with semaphore commands. *semid* argument is the number obtained from a call to *semget()*.

READ functions also use these functions in accordance with the algorithm for READERS as described previously.

```
void write_nstable(struct ntable_entry *ntable, int n) {
    key_t ntable_db_key;
    int ntable_db_semid, sem_val;
    struct ntable_entry *shm, *s;
    struct sembuf ntable_db_sb = {0, -1, 0}; /* semop value is -1 */
    ntable_db_key = NTABLE_DBKEY; /* key value */
    /* Grab the db semaphore */
    if((ntable_db_semid = semget(ntable_db_key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }
    /* get the current value of db semaphore
    if((sem_val = semctl(ntable_db_semid, 0, GETVAL, 0)) == -1) {
        perror("semctl");
        exit(1);
    }
    /* P(db) The P() operation on the semaphore */
    if(semop(ntable_db_semid, &ntable_db_sb, 1) == -1) {
        perror("semop");
        exit(1);
    }
    /* ENTER CRITICAL SECTION */
    /* LEAVE CRITICAL SECTION */
    ntable_db_sb.sem_op = 1; /* free resource */
    /* V(db) */ The V() operation on the db semaphore
    if(semop(ntable_db_semid, &ntable_db_sb, 1) == -1) {
        perror("semop");
        exit(1);
    }
}
```

4.3.2.4 Dynamically linking shared global objects with the stacks

For each global memory object *<obj>* we shall have a *obj_stubs.c* and *obj.h* file.

Object file *obj_stubs.o* is created using the command

```
gcc -c -fpic obj_stubs.c
```

Shared object *dllobj.so* is created using the command

```
gcc -shared -lc -o dllobj.so obj_stubs.o
```

All shared objects (*dll_.so files*) are dynamically linked with the stacks that need them at run-time.

4.3.3 Global Memory Initialization

Both shared memory and semaphores, which are part of global memory, are created and stored in the Linux kernel. Global memory initialization on a node has to be done prior to running the composite protocol stacks that use them. Initialization comprises of shared memory initialization and semaphore initialization.

Shared Memory Initialization: consists of creating the necessary shared memory segments for all the global memory objects.

Semaphore Initialization: consists of initializing the set of semaphores (3 of them) for each global memory object.

mutex, *db*, *readcount* are the three semaphores . *mutex* and *db* are initialized to 1 , *readcount* is initialized to 0 .

```

Sample initialization code from <ntable_shminit.c>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include "../keys.h"

/* A maximum of 20 neighbor table entries can be stored in shared memory */
struct ntable_entry {
    int intf_addr;
    int nbor_addr;
    int lastbit;
};

int main ()
{
    key_t key;
    int shmsize,shmids,i,j;
    struct ntable_entry *s,*shm,init_arr[20];

    /* Initialize the init_arr */
    for(i=0;i<20;i++) {
        init_arr[i].intf_addr = 0;
        init_arr[i].nbor_addr = 0;
        init_arr[i].lastbit = 0;
    }
    /* key value */
    key = NTABLE_SHMKEY;

    /* Create the segment */
    if ((shmids = shmget(key,4096,IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment */
    if ((shm = shmat(shmids, NULL, 0)) == (struct ntable_entry *) -1) {
        perror("shmat");
        exit(1);
    }

    /* copy the init_arr to shared memory */
    s = shm;
    shmsize = sizeof(struct ntable_entry) * 20;
    printf("Initializing NEIGHBOR TABLE \n");
    memcpy(s,init_arr,shmsize);
    printf("no of bytes written: %d\n",shmsize);
}

```

The content of shared memory segment and semaphores on a node can be viewed using the Linux command `<ipcs>`

Sample semaphore initialization code

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "../keys.h"

int main (void)
{
    key_t ntable_mutex_key,ntable_db_key,ntable_rc_key;
    int ntable_mutex_semid,ntable_db_semid,ntable_rc_semid;
    union semun arg;

    ntable_mutex_key = NTABLE_MUTEXKEY;
    ntable_db_key = NTABLE_DBKEY;
    ntable_rc_key = NTABLE_RCKEY;

    if((ntable_mutex_semid = semget(ntable_mutex_key,1,0666 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }
    if((ntable_db_semid = semget(ntable_db_key,1,0666 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }
    if((ntable_rc_semid = semget(ntable_rc_key,1,0666 | IPC_CREAT)) == -1) {
        perror("semget");
        exit(1);
    }
    /* initialize mutex and db semaphores to 1 */
    arg.val = 1;
    if(semctl(ntable_mutex_semid,0,SETVAL,arg) == -1) {
        perror("semctl");
        exit(1);
    }
    if(semctl(ntable_db_semid,0,SETVAL,arg) == -1) {
        perror("semctl");
        exit(1);
    }
    return 0;
}
```

4.3.4 Multicast Service Objects and their Functional Interface

The global memory objects used by the multicast service are

- Neighbor Table
- Routing Table
- Source Tree
- Group Table
- Prune Table

For each global memory object, the ML data structure types and the list of functional interfaces they provide is listed in this sub-section. Since shared memory is always available as a contiguous chunk of memory, global memory data structures cannot be stored in the form of linked-lists or hash-tables. All objects are stored as an array of structures (contiguous memory) in its own allocated and initialized shared memory space.

4.3.4.1 Neighbor Table

The neighbor table stores multiple 1-1 mappings between an interface and the corresponding neighbor detected on that interface. Its functional interface allows creation and update of these mappings through its Write functions and provides functions to retrieve an element of a map given the other. In general, this object can be used by any protocol to store interface-neighbor mappings. For example, it could be used by OSPF's hello-protocol. In this multicast service, the Write interface is used by Neighbor Discovery and the Read interface is primarily used by the Multicast Forwarding component. The table below lists some of the core functions.



WRITE	<i>void write_nhtable([in] struct ntable_entry ntable[], [in] int num);</i>
	invoked by Neighbor Discovery when new neighbor is discovered or when existing neighbor is found dead
READ	<i>[int32] int getNeighborForInterface([in,int32] int intf);</i>
	returns the neighbor's IP address given the interface IP address
	<i>boolean isAddrNeighbor([in,int32] int addr);</i>
	returns true if the input IP address is a neighbor and false if not
	<i>[int32] int getInterfaceForNeighbor([in,int32] int nbor);</i>
	returns the interface's IP address given the neighbor's IP address
	<i>void read_nhtable([out] struct ntable_entry ntable[]);</i>
	returns the entire content of the Neighbor Table. 

Table 2: Neighbor Table- Functional Interface

```

struct ntable_entry {
int32 intf_addr; // interface IP address
int32 nbor_addr; // neighbor IP address
boolean lastbit; //flag
};

```

4.3.4.2 Routing Table

The routing table is a repository for unicast-routes. The metric and next-hop information for each route prefix is stored in this object. In general, any protocol that needs to create and store routes can use this e.g. RIP can also use this. Here the Route Exchange component interacts with this object to store its routes. Multicast Forwarding primarily uses its Read interface during RPF checks and Unicast Forwarding uses it during forwarding unicast packets. The table lists the core functional interfaces for the object.

WRITE	<i>void write_rtable([in] struct rtable_entry rtable[], [in] int num);</i>
	invoked by Route Exchange when new routes are found
READ	<i>[int32] int getNextHopForDest([in,int32] int dest_addr);</i>
	returns the next-hop IP address for a given destination IP address.

Table 3: Routing Table – Functional Interface

```

struct rtable_entry{
int32 rt_netaddr; // network address
int32 rt_netmask;// network mask
int metric; // hop-count
int32 nexthop;// next-hop address
boolean rt_lastbit; //flag }

```

4.3.4.3 Source Tree

The source tree object maintains spanning trees for each multicast source in the network. A spanning tree for each source network contains information on the dependent downstream neighbors for that source. Here, the Spanning Tree component interacts with this object when its creates/updates spanning tree information. The Multicast Forwarding component uses its Read interface during the forwarding process. The table lists the core functional interfaces offered by this object.

WRITE	<i>void write_source_tree([in] struct tree_entry tree[], [in] int num);</i>
	invoked by Poison Reverse component when a Poison packet is received
READ	<i>void getDnStreamNeighborsForSrc([in,int32] int src_addr, [out] t0 nbor_list[]);</i>
	returns downstream dependent neighbors for a particular source address
	<i>void read_source_tree([out] struct tree_entry tree[]);</i>
	returns the entire contents of the Source Tree

Table 4: Source Tree – Functional Interface

```

struct tree_entry{
int32 tree_netaddr;      // network address
int32 tree_netmask;    // network mask
int32 nbor_list[];    // downstream dependent neighbors
boolean tree_lastbit;
};

```

4.3.4.4 Prune Table

The prune table stores interface prune state information for each source-group pair in the network. Interfaces can be in any of the three states: un-pruned, pruned or grafted. This object provides functions to prune/graft specific interfaces for specific source-group pairs. Its Read interface provides functions to retrieve interface state for a specific source-group pair which is used by Multicast Forwarding. The Write functions are used by the Pruning, Grafting and the Join/Leave components. This object is not accessed / used when pruning feature is disabled. The table below lists the core functional interfaces:

WRITE	<i>void pruneIGMPIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);</i>
	Add/update prune table entry for (<i>src_addr,grp_addr</i>) pruning igmp interface <i>intf_ipaddr</i>
	<i>void pruneIGMPIntfforGrp([in,int64] int grp_addr, [in] int intf_ipaddr);</i>
	Add/update prune table entry for (all <i>src_addr</i> 's, <i>grp_addr</i>) pruning igmp interface <i>intf_ipaddr</i>
	<i>void pruneCoreIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);</i>
	Add/update prune table entry for (<i>src_addr,grp_addr</i>) pruning core interface <i>intf_ipaddr</i>
	<i>void graftCoreIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);</i>
	Add/update prune table entry for (<i>src_addr,grp_addr</i>) grafting core interface <i>intf_ipaddr</i>
	<i>void graftIGMPIntfforGrp([in,int32] int grp_addr, [in,int32] int intf_ipaddr);</i>
	Add/update prune table entry for (<i>src_addr,grp_addr</i>) grafting the igmp interface <i>intf_ipaddr</i>

READ	<i>int get_no_of_entries();</i>
	returns the number of entries present in the Prune Table.
	<i>struct prunetable_entry getentry([in] int n);</i>
	returns the nth entry from the Prune Table
	<i>struct prunetable_entry getentryForSrcGrp([in,int32] int src, [in,int32] int grp_addr);</i>
	returns the Prune Table entry corresponding to the (source,group) pair (<i>src,grp_addr</i>)

Table 5: Prune Table – Functional Interface

```

struct intf_entry{
int32 ipaddr // Interface IP address;
int intf_state; // either un-pruned, pruned or grafted
};
struct prunetable_entry {
int32 src_addr; // source address
int32 grp_addr; // multicast group address
struct intf_entry igmp_intf[]; // list of igmp interfaces
struct intf_entry core_intf[]; // list of core interfaces
};

```

4.3.4.5 Group Table

The group table stores group membership information for each interface. It allows dynamic addition of new entries and updating existing entries when members on attached interfaces join and leave multicast groups. It also provides an interface to check if a particular group member is present on an interface. The Join Leave component accesses the Write interface and Multicast Forwarding uses the Read interface. The following table lists the functional interfaces:

WRITE	<i>void write_grptable([in] struct grptable_entry grptable[], [in] int num);</i>
	invoked by the Join-Leave component on the leaf router periodically
READ	<i>boolean checkGrpAddrForIntf([in,int32] int gaddr, [in,int32] int intf_addr);</i>
	checks if the group address <i>gaddr</i> is present on the interface <i>intf_addr</i> , returns true if present.
	<i>void read_grptable([out] struct grptable_entry grptable[10]);</i>
	returns entire contents of the Group Table

Table 6: Group Table – Functional Interface

```

struct grptable_entry{
int32 intf;
int32 grpmem_addr[10];
boolean grp_lastbit;
};

```

4.3.5 Protocol Interactions Through Global Memory

In this sub-section we present a brief operational overview of how the protocol stacks interact with each other using global memory. Global memory is accessed before data transfer, during transfer, when members join and leave groups and also during pruning/grafting of the tree branches.

Before data transfer:

At startup, the global memory objects on all nodes are initialized. Before any transfer of data can take place, the multicast routing and the group management stacks are started. The routing stack components work independently of each other generating and sending packets to their corresponding peers. *Neighbor Discovery* dynamically updates the *Neighbor Table*, *Route Exchange* updates the *Routing Table* and *Spanning Tree* creates and maintains the *Source Tree* global object. *Route Exchange* makes use of *Neighbor*

Table and *Spanning Tree* makes use of *Routing Table* and *Source Tree* global objects for its operation. The pruning and grafting components during this place are not active and thus the global memory *Prune Table* remains un-accessed and empty.

Spanning trees are now fully set-up for data transfer to take place. If members join groups in this phase, the *Join Leave* component updates the *Group Table* at corresponding nodes. They will just remain listening for data, as data transfer has not yet started.

Data Transfer:

Multicast Forwarding in the data stack is the core component, which accesses all the global memory objects. It accesses *Neighbor Table* for interface-neighbor mappings, reads *Routing Table* during its Reverse Path Forwarding [6] check, reads *SourceTree* to get the list of dependent downstream neighbors, reads *GroupTable* to find if there are any group members on its leaf interfaces. If there are no group members on a leaf interface, its prunes the leaf interface and writes into Prune Table. It finally reads from *PruneTable* to get the list of un-pruned/grafted interfaces before forwarding the packet.

Meanwhile, as soon as *PruneTable* entries get created at the leaf nodes, the *Pruning* component becomes active and prunes are sent upward. It should be noted that all the other components of the routing stack *Neighbor Discovery*, *Route Exchange*, *Source Tree* still remain active during this phase dynamically maintaining their respective global objects.

Member join/leave:

The *Group Table* is updated whenever member joins/leaves a group both at leaf router and at end-hosts. In addition to this, when a member joins a group all previously pruned

interfaces corresponding to that group are grafted and this information is written into the *Prune Table*. Thus the Join/Leave component writes into both *Prune Table* and *Group Member Table* as shown above.

Pruning and Grafting: At the core the *Pruning* component writes into *Prune Table* on receiving a prune and *Grafting* writes into the *Prune Table* on receiving a graft.

Thus the stacks work in tandem, interacting with each other using the shared information in the global memory to provide multicast of data through the branches of the multicast tree.

Note: When reliable multicast is used *Unicast Forwarding* uses *Routing Table* to forward unicast NACKs and re-transmissions.

4.4 COMPONENT IMPLEMENTATION

In this section, we shall describe the working of all protocol components used in the multicast service. For each component, we discuss the following features:

- The sender TSM and receiver RSM functionality.
- Memory: Local memory structure, SLPM requirements/usage, packet-memory (bits-on-the-wire) and global memory access.
- Events: how the component responds to data, control and timer events.
- Component- reusability
- Stack placement (position in the stack)

4.4.1 MULTICAST DATA STACK components:

4.4.1.1 Multicast forwarding

This component is the core component in the multicast data stack. It is present on all the nodes i.e. at senders, core and leaf routers as well as end-host receivers. It is responsible for the transmission of multicast data packets on the un-pruned/grafted branches of the multicast tree. Initially when the branches of the tree are not pruned, packets follow the source broadcast tree. But when pruning comes into operation and builds the source-group multicast trees, packets are multicast on the un-pruned branches of the multicast tree.

The TSM is operational only on nodes, which act as Multicast senders. On all other nodes, which either forward multicast data (core and leaf routers) or deliver it to the

application (end-hosts multicast receivers) the TSM remains inactive and only the RSM is operational.

The TSM sends the packet on all un-pruned/grafted interfaces having downstream dependent neighbors for the corresponding (src,grp) pair. The packet is dropped if no downstream neighbors are present for the (src,grp) pair.

Note: In order to prevent sending multiple Esend events (one for each downstream interface) down the stack, this component only generates a single Esend and sends it down with the list of downstream neighbors attached in stack local packet memory (SLPM). The packet will be then handled by the *Replicator* component down below the stack, which actually is responsible for replicating the packet and sending it to the list of downstream interfaces as read from SLPM.

At the router: The RSM contains most of the functionality. It first performs the RPF (Reverse Path Forwarding) check on the packet. This checks if the packet is received on the correct upstream interface, one that is used to reach the source of the multicast packet. If the RPF check fails the packet is dropped. If it is successful, each leaf interface is checked, if any, for group members. If a group member is present on the interface, the packet is multicast on the leaf interface, otherwise the leaf interface is pruned for this (src,grp) pair. The packet is then multicast on all un-pruned/grafted branches of the tree to all dependent downstream neighbors. At the destination (end host multicast receiver) The multicast packet is delivered to the host.

Local memory:

(from /component/dvmp/mcast_forward_sm.ml):

```
type state = {
  node_addr : Addr.set;           // host address
  group_addr : string;           // multicast packet's group address
  mutable noIntf : int;
  totalIntf : int;
  source_addr : Hsys.inet;
}
```

Packet Memory (header):

```
type hdr_t = {
  src_addr : int32;              // multicast packet's source address
  grp_addr : int32              // multicast packet's group address
}
type header = NoHdr | MCastForwHdr of hdr_t
int32 type used to improve performance
```

McastForwHdr is used for all packets sent from the multicast sender or forwarded at core/leaf multicast routers.

SLPM:

setSrcAddr() , *setNextHopAddrList()* SLPM functions used.

- *setSrcAddr()* sets source address of packet in SLPM. TTL component requires this.
- *setNextHopAddrList()* used to set list of next-hop addresses (dependent downstream neighbors) , *Replicator* component requires this.

Global Memory access:

Neighbor Table: *getNeighborForInterface()*, *getInterfaceForNeighbor()*,
isAddrNeighbor()

Routing Table: *getNextHopForDest()*

Source Tree: *getDnStreamNeighborsForSrc()*

Group Table: *checkGrpAddrForIntf()*

Prune Table: *pruneIGMPIntfForSrcGrp()* , *getentryForSrcGrp()*

This component accesses all the global memory objects.

Events:

Data:

- TSM sends all multicast data using *pkt_send()* with *McastForwHdr* if necessary or drops it using *drop_pkt()*. For unicast packets, this component just attaches a dummy *NoHdr*. Note: multicast /unicast packets are identified using a SLPM field "pktType" which is set to "multicast" or "unicast" respectively.
- RSM (at core-routers) forward all packets using *pkt_send()* , at leaf-routers attach a *MCastLeafHdr* and forward using *pkt_send()* and at end-host receivers , deliver using *pkt_deliver()*. All unicast packets with *NoHdr* attached are just passed up without any processing.

Control: does not make use of control events

Timers: does not request any timers.

4.4.1.2 Replicator

This component is actually used by the multicast forwarder to replicate the packet N times and send the packet on N different interfaces. Without this component, the multicast forwarder had to send N separate *ESend* events down the stack to send the packet on N interfaces. This caused lot of overhead and extra processing for the intermediate components in the stack like Fragment, Checksum etc. To prevent this extra overhead the multicast forwarder runs over the replicator (placed bottommost in the stack), and sends only a single *ESend* event with list of next-hop attached in SLPM. The replicator reads from SLPM, gets the list of N next-hop addresses and sends the same packet on N different interfaces. The core-functionality is embedded in the TSM, which reads from SLPM and replicates the packet and sends it. The RSM is almost dummy, it only delivers the packet after setting appropriate SLPM fields like *IncomingInterface()* and *McastSrc()*

Note: This component acts only on "multicast" packets. All "unicast" packets are passed with a *NoHdr* attached.

Local memory:

```
(from /component/mcast/replicator_sm.ml):  
type state = {  
  mutable mcastsrc : Hsys.inet ; // multicast source address of packet  
  node_type : node; // node-identifier , sender, router or receiver  
}
```

Packet Memory (header):

```
type hdr_t = {
    mcastsrc_addr : Hsys.inet;      // multicast source address of packet
    dest_addr : Hsys.inet;         // the intermediate next-hop address
}
type header = NoHdr | Address of hdr_t
```

A *NoHdr* is attached for all unicast packets. *Address* header attached for all multicast packets.

The reason the multicast source address is part of the header is that some components (eg: *RMTP* discussed later) below the multicast forward may need to know the original source address of the multicast packet. So the source address carried as part of header is then set in SLPM at the destination for other components to read. Also, the *dest_addr* is used to set the SLPM field *IncomingInterface* at the destination stack.

SLPM: The TSM reads the list of next-hop addresses from SLPM using the *getNextHopAddrList()* SLPM read function and sets the next-hop destination address using the *setDestAddr()*. The RSM sets the SLPM fields using *setIncomingInterface()* and *setMCastSrc()*.

Global Memory:

Does not make use of any global memory objects.

Events:

Data: Acts only on "multicast" packets, replicates and attaches header *Address*. All "unicast" packets are passed with header *NoHdr*.

Control: Does not make use of control events

Timers: Does not make use of any timers.

Stack Placement:

Has to be placed below the multicast forwarding component and as below/bottom in the stack as possible for reducing the overhead incurred for other intermediate components in the stack. The remaining components described in this sub-section are all property-oriented optional components in the multicast-data stack.

4.4.1.3 Multicast in-order component

This component provides in-order delivery of all packets flowing in a point-to-multipoint multicast network (i.e. from a single sender to multiple receivers). The TSM is fairly simple, each packet is sent after tagging it with a sequence number. The sequence number is incremented monotonically after sending each packet. The core in-order functionality lies in the RSM. The component maintains a separate receive window buffer for each unique sender in the network.

All in-order packets are directly delivered to the application. Out-of order packets are buffered in the receive window. They are actually inserted at the tail of the buffer and then sorted based on increasing sequence numbers. Timers are associated with each buffered packet to prevent it from remaining forever in the buffer. Buffered packets are delivered when their corresponding timers expire.

Limitations: cases when this component does not deliver packets in-order:

- When the buffer is full and an out-of-order packet arrives, the first packet in the buffer is delivered. So the in-order property is limited by the degree of in-orderness, which should not exceed the window size.
- A buffered packet's timer expires (usually set to a large value).

Local Memory

```
(from /components/mcast/mcast_inorder_sm.ml)
type 'abv seq_rec = {
  // buffer contents
  sq_num : int;           // sequence number of buffered packet
  payload : 'abv pktpayld; // payload of the buffered packet
}
type 'abv recv_window = { // receiver window parameters
  mutable exp_seq : int;           // next expected sequence number
  mutable last_seq_num : int;     // the last sequence number delivered to the application
  mutable count : int;           // count of no of packets buffered
  mutable j : int;
  buffer : 'abv seq_rec array;    // packet buffer
  max_size : int;                // size of the receiver window
}
type 'abv state = {
  buffer_sweep : Time.t // maximum time a packet can be buffered at the receiver
  mutable send_next : int; // The next sequence number to be generated at the sender
  recv_buffer : (Hsys.inet * ('abv recv_window)) list ref; // unique buffer for each sender
  mutable window : 'abv recv_window // window for the current source
}
```

Packet Memory (header):

```
type header = NoHdr | DataPkt of int
```

Header carries just the in-order sequence number.

SLPM: not used

Global Memory: not used

Events:

Data: TSM sends all "multicast" packets with header *DataPkt* using *pkt_send()*.

RSM delivers all in-order packets using *pkt_deliver()*, and buffered packets sent using *deliver_kept_packet()*.

Control: does make use of any control event.

Timers: buffer timer used to deliver packets buffered for too long.

Stack Placement: to be placed above the multicast forwarding component to provide the desired end-to-end point-to-multipoint in-order delivery.

4.4.1.4 End-to-End Reliable (without NACK implosion prevention)

This component provides end-to-end reliable and in-order delivery of packets in a point-to-multipoint network (i.e. from a single sender to multiple receivers). The working of this component is based on RMTP, but this does not implement the NACK-implosion prevention mechanism. (NACKs are sent all the way up the tree to the original multicast sender). This component is operational only at the multicast sender and at all end-host multicast receivers.

The multicast sender handles:

(a). Transmission of multicast packets, (b) buffering of un-ACKed data in send buffer (c) NACK processing (d) Re-transmission of data using either multicast or unicast.

The receiver is responsible for:

(a). periodic transmission of a NACK packet (reporting packets that are not yet received) back to the sender. (b) buffering out-of-order packets in receive buffer. (c) delivering in-order data to the application.

Timers used are *dally_timer* (T_{dally}), *retrans_timer* ($T_{retrans}$) and *nack_timer* (T_{nack}).

Transmission/buffering of multicast packets: (handled by TSM at multicast sender)

Each multicast packet is tagged with a sequence number. (starts from 0 and is monotonically increased for every packet). All packets sent are buffered in *send_buffer* for later re-transmission if needed. The *retrans_timer* is also started after sending the first packet.

It should be noted that in this type of multicast network, the sender does not explicitly know who the receivers are. Receivers can dynamically join/leave a particular multicast session. The goal is to provide reliable delivery to the current members of the session. So the creation and termination of sessions is timer based. *dally_timer* is used for this purpose. After sending the last packet in the session, the *dally_timer* is started. T_{dally} is defined as at least twice the lifetime of the packet in the network. Receivers send back their REQ packets only if they have lost packets. The *dally_timer* is reset on receiving a REQ from any of the receivers. Also, time interval between sending two consecutive REQs is much smaller than T_{dally} . So, expiry of the *dally_timer* implies that either (a). all current receivers have correctly received all packets (b). something exceptional like a permanent link breakdown has occurred. This ensures termination of the session and all connection state (e.g. *send buffer contents*) are deleted.

Negative Acknowledgement packets (NACKs)

NACK packets are used to periodically (*Tnack*) report the contents of the receiver window to the sender. They contain the next expected sequence number at the receiver and a sequence list of packets that have not received. When all packets are correctly received and in-order, the receiver window is empty and thus no NACK packets are sent.

Receiving NACKs (handled by RSM at multicast sender)

The sender buffers all NACK packets in *nack_buffer* received during every period *Tretrans*. These NACK packets from different receivers in the network will be later processed when the *retrans_timer* expires. 

NACK processing and retransmissions (handled by TSM at sender):

When the *retrans_timer* expires, the *nack_buffer* is processed and a *retrans_list* is created. Each element in the list contains the packet sequence number and list of receivers that has requested this packet. to be transmitted. For each retransmission, if the number of receivers requesting packet exceeds a threshold *Mcast_Threshold*, the packet is re-transmitted using multicast, if not is it unicast back to the particular receiver.

Local Memory

```
(from /components/mcast/mcast_reliable_sm.ml)

type 'abv state = {
  node_addr : Addr.set;           // host address
  dally_sweep : Time.t;          // The dally timer interval
  retrans_sweep : Time.t;        // The re-transmit timer interval
  ack_sweep : Time.t;            // The ACK timer interval
  buffer_sweep : Time.t // maximum time a packet can be buffered at the receiver
  mutable send_next : int; //The next sequence number to be generated at the sender
  mutable send_left : int;       // Send window left edge
  mutable send_count : int;      // no of packets in the send buffer
  send_buffer : 'abv seq_rec array; // store all un-ACKed packets at the sender
  ack_buffer : (Hsys.inet * ack_buf) list ref; // store all ACK packets
  send_max : int;                // send window size
  // store all out-of-order packets at the receiver, a buffer for each unique sender
  recv_buffer : (Hsys.inet * ('abv recv_window)) list ref;
  mutable window : 'abv recv_window ; // window for the current source
  threshold : int ;              // re-transmission unicast/multicast threshold
  mutable pktseqno : int;        // a loop count
}

recv_window and seq_rec are same as that used in mcast_inorder component.
```

Packet Memory (header)

```
// buffer to store contents of ACK
type ack_buf = {
  buf_left_edge : int;
  buf_bitvector : bool array;
}

type ack_header = {
  src : Hsys.inet; // The source address of the ACK
  left_edge : int; // the sequence number corresponding to the left-edge
  bitvector : bool array; // the bit vector , 0 for lost/un-received pkts 1 for received pkts
}

type header = NoHdr | DataPkt of int | Ack of ack_header
```

Two types of packets: data sent using header *DataPkt*, ACKs sent using header *Ack*.

SLPM:

getMCastSrc() used to get the source address of the multicast packet. This should be set by the multicast forwarding component down below.

Global Memory:

Does not make use of any global memory objects

Events:

Data:

- Initial transmission of multicast data by TSM using *pkt_send ()* with header *DataPkt*.
- Unicast re-transmission by TSM using *send_kept_packet ()* with "unicast" tag in SLPM.
- Multicast re-transmission by TSM using *send_kept_packet ()* with "multicast" tag in SLPM.
- Deliver in-order received packets using *pkt_deliver ()*
- Deliver buffered packets using *deliver_kept_packet ()*
- ACKs sent by RSM using *new_pkt_send ()* with header *ACK*

Control: does not make use of any control events

Timers:

dally_timer(timer-id: 1001) and *retransmit_timer(timer-id:1002)* both used by TSM

ack_timer(timer-id: 1003) used by RSM

Stack Placement: must be placed above the multicast forwarding component to provide end-to-end reliable delivery.

4.4.1.5 Reliable with NACK- implosion prevention

The component described in the previous section does not prevent the NACK-implosion problem. NACK implosion refers to the undesirable situation when an upstream link gets congested due to excessive number of NACK packets flowing through it resulting from the flow of several individual NACKs from downstream receivers.

The RMTP approach to solve the NACK-implosion problem is as follows:

RMTP is based on a hierarchical structure where the receivers are grouped into local regions or domains and in each domain there is a special receiver called *designated receiver DR* which is responsible for sending NACKs periodically to the sender, for processing NACKs from receivers in its domain and for re-transmitting lost packets to receivers in its domain. Since lost packets are recovered by local retransmissions as opposed to retransmissions from original sender, the end-to-end latency is considerably reduced and the overall throughput is improved as well. Since only *DRs* send NACKs back to the sender, instead of all receivers sending their NACKs to the sender, only one NACK is generated per local region and thus NACK implosion is prevented. Receivers now send their NACKs periodically to the *DR* in their local region.

We now describe only the modifications and enhancements made to the previous end-to-end reliable component to yield this RMTP-like component.

The following modifications had to be made:

(a). Change in stack position: Earlier, the end-to-end reliable component was placed above the *mcast_forward* component. But here we need the *DRs* (which are actually core/leaf routers in the network) to act on data packets, send NACKs etc. As on routers, data packets are always only forwarded by the *mcast_forward* component and are never

delivered above, packets would never reach this component if it were placed above the *mcast_forward* component. So this component has to be placed below *mcast_forward*.

(b). Node Types: The end-to-end reliable component is operational only at the original sender (S) and at the end-host receivers (Rs). Here we define two more node types *DRs* and *NDRs* (*non-designated receivers*).

(c). Sender: same functionality except that re-transmissions cannot be multicast; they can only be unicast back to the sender. This is because *mcast_forward* is above this component.

(d). Non-designated receivers (*NDRs*): This does not act on data packets, it only passes them around. The RSM reads the sequence number from the packet and sets the SLPM field *RelSeqNo* , which is then read by the TSM (after packet turn around by *mcast_forward*) and placed back onto the header.

(e). Designated receivers (*DRs*): are responsible for sending NACKs periodically back to the original sender, storing out-of-order packets in receive buffers, deliver in-order packets to the component above and also store them in *send_buffer* for later retransmission to receivers (*Rs*), process NACKs from receivers in their region.

(f) Normal receivers (*Rs*): same functionality except that the NACKs are now sent to the corresponding configured *DR*.

Local Memory and Packet Memory: same as in previous component.

SLPM access:

SetRelSeqNo and *getRelSeqNo()* are used to set/get the sequence number to/from SLPM on *NDRs*.

getMcastSrc() used to find out original source address of multicast packet. This SLPM value should be set down below by either the *Replicator* (for multicast pkts) or by the *Forward* (for unicast pkts).

Global Memory: does not make use of global memory objects.

Events: All data events are handled in a similar way . No control events are used. Make use of the timers *dally_timer*, *retrans_timer* and *ack_timer* as before.

Note: the choice of configuring a node as a DR is done manually at configuration time.

Stack Placement : As discussed earlier placed below the *mcast_forward*.

Other components: *Unicast_Forward*, *TTL* , *Fragment* , *Checksum* are a few of the other components that can/are used in the multicast data stack.

Unicast_Forward : used by the stack to send unicast packets eg : ACKs / retransmissions

4.4.2 MULTICAST ROUTING STACK components:

4.4.2.1 Neighbor Discovery

The main functionality of this component is to dynamically discover neighbors (multicast routers) on all its interfaces.

The TSM periodically broadcasts *probe packets* (hello packets) on all multicast-enabled interfaces. Each probe packet sent on a particular interface contains a list of neighbors for which neighbor probe messages have been received on that interface.

Packets from other components above, if any, are passed with a dummy header *NoHdr* attached.

The RSM first checks if the neighbor probe packet is received on one of its locally defined interfaces and if yes, updates in its local memory: the neighbor address and the interface on which it is received. It then checks for 2-way adjacency i.e. if the local interface address is present in the neighbor list of the probe packet. If present, then a 2-way adjacency is established and neighbor is discovered on that interface. This information is written into and maintained in the global memory data structure Neighbor Table.

Packets with header *NoHdr* are not processed and are delivered to the component above.

The RSM also provides a keep-alive function in order to quickly detect neighbor loss.

When a neighbor is discovered for the first time, the timer *neighbor_expiry* is set. If no probe packet is received within the time *neighbor_expiry_sweep* the timer is cancelled and this neighbor entry is removed from the global memory Neighbor Table. On receiving *probe packets*, this timer value is reset.

Local memory:

```
(from /components/dvmp/neighbor_discovery_sm.ml)

type state = {
  probe_sweep : Time.t;           // probe timer interval timer-id: 20000
  neighbor_expiry_sweep : Time.t; // neighbor-expiry timer interval
  ntable      : (Hsys.inet * Hsys.inet) list ref; // an association for the local neighbor table
  ipaddr      : Hsys.inet ref;    // the incoming IP interface address
}
```

Packet Memory (header):

```
type hdr_t = {  
  src_addr : Hsys.inet;           // source IP address  
  dest_addr : Hsys.inet;         // probe broadcast address  
  neighboraddr_list : Hsys.inet list; // list of neighbor IP addresses  
}  
type header = NoHdr | Probe of hdr_t
```

This component does not depend on any other component for addressing. So address information is carried as header in this component itself. All probe packets are sent with header *Probe* and all packets from component above are sent with *NoHdr*.

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used. The TTL component that runs below this component expects the *SrcAddr* field in SLPM (the source address of the packet). The *DestAddr* field in SLPM carries the next-hop address and is used by a component below *bottom* to forward the packet to the next-hop.

Global Memory access:

Writes into global memory *Neighbor Table* using the external function *write_ntable()*

Events:

Data:

- TSM passes all *ESend* events with *NoHdr* attached using *pkt_send()* and RSM delivers all packets with *NoHdr* using *pkt_deliver()*.
- Probe packets sent with header *Probe* using *new_pkt_send()*.

Control: does not need control events.

Timers:

- *probe_timer* with timer-id 20000 for periodically sending *probe packets*.
- *neighbor_expiry_timer* for detecting dead neighbors. Timer-id is chosen as the integer value of the neighbor's IP address.

Component-reuse: This component can be used in other protocols where there is a need for neighbor discovery e.g. in unicast routing protocols like *RIP* and *OSPF*.

Stack-placement: This component being a control oriented peer-to-peer component can be placed anywhere among the DVMRP components in the stack. For performance reasons it is recommended that this component be placed lowest among the other multicast routing components as this sends peer-to-peer messages most frequently.

4.4.2.2 Route Exchange

The main functionality of this component is to dynamically create and maintain the routing tables at the multicast routers through periodic exchange of *route exchange packets* with neighbors. This is a RIP-like protocol component, with metric based on hop-counts.

The TSM periodically sends *route exchange* packets to all its neighbors. The list of neighbors is read from the global memory *Neighbor Table*. Each *route exchange packet* contains a list of routes with each route comprised of a network prefix, mask and metric. All packets from any component above are passed with a dummy header *NoHdr* attached.

The RSM, for each route exchange packet received, first checks with its local *route cache* if the received route is a new route or not. If new then the route is stored in the local route cache. If not, then the received metric for the route is compared with the

existing metric after adding the cost of the incoming interface to the received metric. If the resultant metric is better than the existing one, then the local route cache is updated. After all the received routes are processed, the contents of the local route cache are written to a global data structure *Routing Table* in global memory. The *Routing Table* contains entries of the form *prefix, mask, metric, next-hop*.

All packets with a *NoHdr* attached are just passed up to the component above.

Local memory:

```
(from /components/dvmrp/route_exchange_sm.ml)
type state = {
  sweep : Time.t;           //route_exchange timer interval
  node_addr : Addr.set;     // host's address
  rt_list : ((Hsys.inet * Hsys.inet) * (int * Hsys.inet)) list ref; // local routing table
  mutable noRoutes : int;
  mutable noRoutesChecked : int;
}
```

Packet Memory (header):

```
type route = {
  net_addr : Hsys.inet;           // network IP address
  netmask : Hsys.inet;           // network mask
  metric : int;                   // route metric/ hop-count
}
type hdr_t = {
  src_addr : Hsys.inet;           // source of the Route_Exchange packet
  dest_addr : Hsys.inet;         // Route_Exchange destination address
}
```

This component does not depend on any other component for addressing. So address information is carried as header in this component itself. All route exchange packets are

sent with header *RouteExchange* and all packets from component above are sent with the dummy header *NoHdr*.

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used in a similar way as in Neighbor Discovery.

Global Memory access:

- Reads from the Neighbor Table using the external function *getNeighborForInterface()*
- Writes into the Routing Table using the external function *write_rtable()*

Events:

Data:

- TSM passes all *ESend* events from above with *NoHdr* attached using *pkt_send()* and RSM delivers all packets with *NoHdr* using *pkt_deliver()*.
- Route Exchange packets are sent with header *RouteExchange* using *new_pkt_send()*.

Control: does not need control events.

Timers:

Route_exchange_timer with timer-id 30000 for periodically sending *route_exchange* packets.

Component-reuse: This component can be re-used in other distance vector-based unicast routing protocols like RIP.

Stack-placement: This component being a control oriented peer-to-peer component can be placed anywhere among the DVMRP components in the stack. However, it needs to be placed over TTL for sending *route exchange packets* with a TTL of 1.

4.4.2.3 Spanning Tree

In DVMRP, the poison reverse functionality and creation of spanning trees is embedded as part of the route exchange process itself. Here the functionality is built into a separate component. This component enables each upstream router to form a list of dependent downstream routers for a particular multicast source. Each downstream router informs its upstream router that it depends on it to receive multicast packets from a particular source. This is done through periodic exchange of *Poison Reverse* packets.

The TSM needs access to the global memory *Neighbor Table* and *Routing Table*. The entries in the *Routing Table* are grouped based on next-hop information. All prefixes having the same next-hop are grouped together in different lists called *poison reverse lists*. Each of these lists is sent in the form of *poison reverse packets* to their corresponding next-hops (which are actually upstream neighbors for the source networks in the list). All packets from any component above are passed with a dummy header *NoHdr* attached.

The RSM on the upstream neighbor uses all the *poison reverse lists* it receives to form a *spanning tree* for each source. Thus, this component builds a list of downstream dependent neighbors for each source network. The tree is stored in global memory as *Source Tree*.

All packets with a *NoHdr* attached are just passed up to the component above.

Local memory:

```
(from /components/dvmp/poison_reverse_sm.ml)
type prefix = {
  net_addr : Hsys.inet;           // Network address
  netmask  : Hsys.inet;           // Network mask
}
type state = {
  poison_reverse_sweep : Time.t; // poison_reverse timer interval
  node_addr : Addr.set;          // hosts unique address
}
```

Packet Memory (header):

```
type hdr_t = {
  src_addr : Hsys.inet;           // source IP address (downstream router)
  dest_addr : Hsys.inet;          // destination IP address (upstream router)
  src_nw_list : prefix list;      // poison reverse list
}
type header = NoHdr | PoisonReverse of hdr_t
```

This component does not depend on any other component for addressing. So address information is carried as header in this component itself. All poison reverse packets are sent with header *PoisonReverse* and all packets from component above are sent with the dummy header *NoHdr*.

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used in a similar way as in Neighbor Discovery.

Global Memory access:

- *Neighbor Table* READ using external function *getInterfaceForNeighbor()*
- *Routing Table* READ using *read_rtable()*.
- *Source Tree* WRITE using *write_source_tree()*

Events:

Data:

- TSM passes all *ESend* events from above with *NoHdr* attached using *pkt_send()* and RSM delivers all packets with *NoHdr* using *pkt_deliver()*.
- *Poison Reverse* packets are sent with header *PoisonReverse* using *new_pkt_send()*.

Control: does not need control events.

Timers:

Poison_Reverse_timer with timer-id 50000 for periodically sending the *PoisonReverse* packets.

Stack-placement: This component being a control oriented peer-to-peer component can be placed anywhere among the DVMRP components in the stack. However, it needs to be placed over TTL for sending the *poison reverse packets* with a TTL of 1.

4.4.2.4 Pruning

The primary purpose of this component is to create and maintain the global data structure *Prune Table* on each node that stores the list of pruned downstream interfaces for each source/group pair. This along with the *Spanning Tree* component constructs per source-group multicast trees at each node. (Note: the *Spanning Tree* component by itself constructs a per-source broadcast tree at each node).

The TSM is responsible for sending *prune packets* for a particular source-group pair addressed to the corresponding upstream neighbor under the following conditions:

- (a). If all its downstream dependent neighbors have sent prunes and all its IGMP interfaces are also pruned.
- (b). If all its downstream dependent neighbors have sent prunes and there are no IGMP interfaces (at multicast core routers).
- (c). If there are no downstream dependent neighbors and all IGMP interfaces are pruned (at multicast leaf routers).

For this, the TSM reads all the entries of the Prune Table periodically using a prune timer and if needed sends a prune packet for the (source, group) upstream towards the source.

All packets from any component above are passed with a dummy header *NoHdr* attached.

The RSM is mainly responsible for updating the global memory *Prune Table*. When a *prune packet* for (*src, grp*) is received on an interface *intf*, it adds an core interface prune entry in the Prune Table containing source *src*, group *grp* and incoming core interface *intf* (interface to be pruned). All packets with a *NoHdr* attached are just passed up to the component above. Note that the TSM reads from the *Prune Table* and the RSM writes to the *Prune Table*.

Local memory:

```
(from /components/dvmp/pruning_sm.ml)
type state = {
  prune_sweep : Time.t;           // timer for periodically checking Prune Table entries
  node_addr : Addr.set;          // host address
  mutable noEntriesChecked : int;
  mutable total_prunes : int;
  mutable prune_entry : prunetable_entry; // a prune table entry
}
```

Packet Memory (header):

```
type hdr_t = {
  saddr : Hsys.inet;           // address of router sending prune
  dest_addr : Hsys.inet;      // address of router receiving the prune
  pr_src_addr : Hsys.inet ;    // multicast source address
  group_addr : string;        // group address
}
type header = NoHdr | Prune of hdr_t
```

A *NoHdr* header is attached for all messages from above. For messages generated from this component a *Prune* header is attached.

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used in a similar way as in Neighbor Discovery.

Global Memory access:

- *Neighbor Table*: *getInterfaceForNeighbor()* and *getNeighborForInterface()*
- *Routing Table*: *getNextHopForDest()*
- *Source Tree*: *getDnStreamNeighborsForSrc()*
- *Prune Table*: *getEntry()* , *get_no_of_entries()* and *pruneCoreIntfforSrcGrp()*

Events:

Data:

- TSM passes all *ESend* events from above with *NoHdr* attached using *pkt_send()* and RSM delivers all packets with *NoHdr* using *pkt_deliver()*.
- *Prune* packets are sent with header *Prune* using *new_pkt_send()*.

Control: does not need control events.

Timers:

Prune_timer with timer-id 70000 for periodically sending the *Prune* packets.

Stack-placement: This component being a control oriented peer-to-peer component can be placed anywhere among the DVMRP components in the stack. However, it needs to be placed over TTL for sending the *prune packets* with a TTL of 1.

4.4.2.5 Grafting

This component is responsible for removing the appropriate pruned branches of the multicast tree when a host rejoins a multicast group. When a group join occurs for a group that the router has previously sent a prune, the global *Prune Table* is updated by the *Join Leave* component to un-prune the local IGMP interface for that particular group.

The TSM periodically reads from the global *Prune Table*, and sends a separate *graft packet* for a particular *(src,grp)* to appropriate upstream routers for each source network under the following conditions:

- (a) On leaf-routers if the interface attached to all hosts is un-pruned.
- (b) On core routers if a graft packet is received on any of the previously pruned downstream interfaces.

All packets from any component above are passed with a dummy header *NoHdr* attached. The RSM on receiving a *graft packet* writes to the global *Prune Table* to update the list of grafted core interfaces per source-group. Thus, this component along with the *Pruning* component maintains the global Prune Table by dynamically updating the list of pruned/grafted downstream interfaces for each source-group pair. All packets with a *NoHdr* attached are just passed up to the component above.

This component assumes a Reliable component underneath it for reliability of its Graft packets. This obviates the need for this component to handle Graft ACK packets as in traditional DVMRP.

Local memory:

```
(from /components/dvmrp/grafting_sm.ml)
type state = {
  graft_sweep : Time.t;           // timer for periodically checking PruneTable
  node_addr : Addr.set;          // host address
  mutable noGraftCheckedt : int;
  mutable total_grafts : int;
  mutable prune_entry : prunetable_entry; // an entry of Prune Table
}
```

Packet Memory (header)

```
type hdr_t = {
  sr_addr : Hsys.inet;           // address of router sending graf)
  dest_addr : Hsys.inet;         // address of router receiving the graft)
  graft_src_addr : Hsys.inet ;   // multicast source address
  graft_group_addr : string;     // group address
}
```

A NoHdr header is attached for all messages from components above. Packets generated from this component attach a Graft header.

```
type header = NoHdr | Graft of hdr_t
```

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used in a similar way as in Neighbor Discovery.

Global Memory access:

- *Neighbor Table*: *getInterfaceForNeighbor()*
- *Routing Table*: *getNextHopForDest()*
- *Prune Table*: *getEntry()* , *get_no_of_entries()* and *graftCoreIntfforSrcGrp()*

Events:

Data:

- TSM passes all *ESend* events from above with *NoHdr* attached using *pkt_send()* and RSM delivers all packets with *NoHdr* using *pkt_deliver()*.
- *Graft* packets are sent with header *Graft* using *new_pkt_send()*.

Control: does not need control events.

Timers:

Graft_timer with timer-id 80000 for periodically sending the *graft* packets.

Stack-placement: This component being a control oriented peer-to-peer component can be placed anywhere among the DVMRP components in the stack. However, it needs to be placed over TTL for sending the *graft packets* with a TTL of 1.

4.4.3 GROUP MEMBERSHIP STACK components:

4.4.3.1 Join/Leave component with its control interface

Initially, the IGMP protocol was decomposed into two separate components: *Join_Leave* and *Query_Report*. The *Join_Leave* component to handle user join and leave to a multicast group and the *Query_Report* component to handle group membership updates from end-hosts to leaf-routers. But the *Join_Leave* component did not fully satisfy our definition of a protocol component. Its TSM did not send packets on the wire and it had

no RSM functionality. So, finally these were merged into a single component called *Join_Leave*. Another interesting feature about this component is that it is asymmetric in nature. The TSM and RSM functionality differs depending on where the component is deployed at the end-host or at the leaf multicast router. So, in order to make the state machines symmetric both the state machines contain exclusive transitions for end-hosts and routers.

We describe the TSM and RSM functionality separately at the end-hosts and at the leaf-router.

At the end-host:

The TSM responds to control event *EControl* of type *JoinGroup* and *LeaveGroup*. (These events are generated by the application when the host wants to join or leave a particular multicast group). The local *group cache* is updated when these events occur to always store the current list of group addresses to which this host belongs. The RSM responds to the *Query packets* from the leaf-router by sending back a separate *Report packet* for each group of which it is a member.

At the multicast-leaf router:

The TSM periodically performs the following tasks on expiry of the query timer:

multicasts *query packets* on the local network to the "all-hosts-group".

computes the list of newly joined as well as the list of newly left group addresses on each attached interface over the last timer interval. For each newly joined group address on a particular interface the global memory Prune Table is updated by grafting the interface for that group address.

writes the contents of the local *router_group_cache* into global memory *Group Table*.

The RSM processes the Report packets received from its attached hosts and updates the local *router_group cache*. Note that the local *router_group cache* maintains information on list of group members on each attached interface. It should be noted that the component at the end-host is initialized "actively" and that at the router "passively" through *EActiveInit* and *EPassiveInit* events respectively.

Local memory:

```
(from /components/igmp/join_leave_sm.ml)
type state = {
  query_sweep : Time.t;           // timer interval for query timer
  group_list : (string list) ref; // list of group address of which this host is a member
  router_group_list : ((Hsys.inet * string list) list) ref; // group address list at router
  prev_router_group_list : ((Hsys.inet * string list) list) ref; // value in previous time-interval
  node_addr : Addr.set;           // host address
}
```

Packet Memory (header):

```
type hdr_t = {
  src_addr : Hsys.inet;
  dest_addr : Hsys.inet;
  group_address : string;
}
type header = NoHdr | Query of hdr_t | Report of hdr_t
```

Note: For Query packets, *src_addr* is the address of the leaf router's interface. For Report packets *src_addr* is the address of the host sending the report and *dest_addr* is the address of the multicast leaf router. *group_address* in Report packets refers to the group address being reported.

SLPM:

setSrcAddr() and *setDestAddr()* SLPM functions are used in a similar way as in Neighbor Discovery.

Global Memory access:

- *Group Table: write_grptable()*
- *Prune Table: graftIGMPIntfforGrp()*

Events:

Data: *Query* and *Report packets* are sent with header *Query* and *Report* using *new_pkt_send()*.

Control: Responds to *EControl* event of type *JoinGroup* and *LeaveGroup*.

Timers: *Query timer* with timer-id 40000 for periodically sending *query packets*.

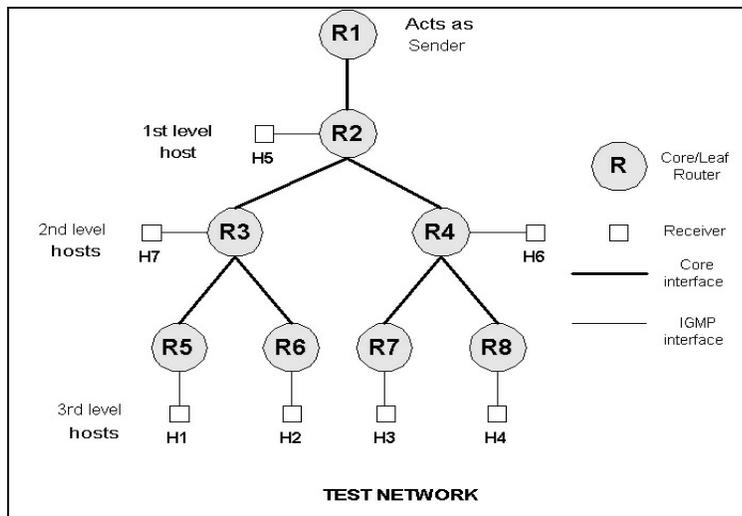
5. Testing and Performance

This section describes the nature and results of various tests and experiments that were performed to verify correct operation of the composite multicast service running on a reasonably sized 12-node multicast network. The tests can be divided into two major categories, functionality testing and performance testing. In functionality testing, the primary objective is to verify the correct operation of all protocol components and the service as a whole. In performance testing, we conduct test experiments to measure various network parameters like end-to-end throughput, one-way latency, join/leave latencies and also study and observe their variance and effect for different stack combinations, message sizes, error rates etc. Section 5.1 describes the functionality test and section 5.2 describes the various performance measurement tests that were performed using composite protocol stacks.

5.1 Functionality Testing

The following figure shows the test network set-up that was used.

Figure 6: Test Network



The test network consists of 8 routers (R1 to R8) and 7 hosts (H1 to H7). All links are point-to-point 100 Mbps Ethernet.

Addressing scheme: All core links i.e. links connecting routers, have network address of the form 10.10.xy.0/24. where $x < y$. eg: the link connecting R1 and R2 is named as 10.10.12.0/24 and the interface at R1s end has always a lower IP address 10.10.12.1 and R2 has a higher IP address 10.10.12.2. All leaf interfaces have addresses of the form 10.n.1.0/24, where n is the router-no they connect to, e.g: The link between R5 and H1 is addressed as 10.5.1.1 at the router end and as 10.5.1.2 at the host-end. Knowing this addressing scheme will help in better understanding of the test results later on in the section.

Stacks: The multicast data stack is run on all nodes (sender, routers and receivers).

The *multicast routing stack* is run only on the routers from R1 to R8. The *group membership stack* is run on all leaf routers (R2 to R7) and hosts (H1 to H7).

Global Memory Initialization: This has to be done prior to running the stacks on each node. So on each node the script `/ensemble/global_memory/shminit` is run that allocates and initializes the various global memory objects to be used by the stack. The script `/ensemble/global_memory/sem_initall` is then run to initialize all semaphore values used.

Note: the Linux `ipcs` command can be used to view shared-memory and semaphore related information.

Configuration files: `node.itable` and `node.igmptable` are 2 configuration files that are needed by the stack to initialize their interface addressing information. `Node.itable` consists of total list of interfaces and `node.igmptable` consists of list of leaf interfaces.

Running the Multicast Routing stack:

The following command-line shows how to run the *Multicast Routing stack* on router R1:

```
../demo/dvmrp_appl -remove_prop forward -add_prop neighbor_discovery -add_prop route_exchange -add_prop poison_reverse -add_prop grafting -add_prop pruning -pstr interface_table=bn1.itable -pstr igmp_interface_table=bn1.igmptable -port 9500
```

The stack ordering from top to bottom is *pruning*, *grafting*, *poison_reverse*, *route exchange*, *neighbor_discovery* over the default *checksum* component. The component that generates packets most frequently is kept at the bottom-most. So *neighbor_discovery* was placed at the bottom and *grafting* was placed at the topmost. We expect to have a better performance improvement if this ordered is maintained. There is no need to make use of *TTL* as *multicast routing stack* packets are not sent farther than a hop. Also the default *forward* component is removed as there is no need of forwarding. The interface information is read from the two input files *.itable* and *.igmptable*.

```
GLOBAL MEMORY OUTPUT AT ROUTER 1:
```

Neighbor Table			
Neighbor	Interface		
10.10.12.2	10.10.12.1		

Routing Table			
Net	Netmask	Metric	NextHop
10.8.1.0	255.255.255.0	3	10.10.12.2
10.7.1.0	255.255.255.0	3	10.10.12.2
10.6.1.0	255.255.255.0	3	10.10.12.2
10.5.1.0	255.255.255.0	3	10.10.12.2
10.10.47.0	255.255.255.0	2	10.10.12.2
10.10.48.0	255.255.255.0	2	10.10.12.2
10.4.1.0	255.255.255.0	2	10.10.12.2
10.10.35.0	255.255.255.0	2	10.10.12.2
10.10.36.0	255.255.255.0	2	10.10.12.2
10.3.1.0	255.255.255.0	2	10.10.12.2
10.2.1.0	255.255.255.0	1	10.10.12.2
10.10.24.0	255.255.255.0	1	10.10.12.2
10.10.23.0	255.255.255.0	1	10.10.12.2
10.10.12.0	255.255.255.0	0	10.10.12.1

Spanning Tree : Empty

Group Member Table:Empty

Prune Table: Empty

The stack is run over UDP on port no 9500. Similar commands are executed on all routers (from R1 to R8). We now show the global memory output at 3 routers, R1, R3 and R5. Output at other routers are similar. The output corresponds to when the full tree is active and no pruning has started. The global memory output is self-explanatory. From the above output it can be noted that all core and leaf interfaces are advertised by the

```

GLOBAL MEMORY OUTPUT AT ROUTER 3

Neighbor Table

Neighbor          Interface
10.10.36.2        10.10.36.1
10.10.23.1        10.10.23.2
10.10.35.2        10.10.35.1

Routing Table

Net               Netmask          Metric           NextHop
10.8.1.0          255.255.255.0   3                10.10.23.1
10.7.1.0          255.255.255.0   3                10.10.23.1
10.6.1.0          255.255.255.0   1                10.10.36.2
10.10.47.0        255.255.255.0   2                10.10.23.1
10.10.48.0        255.255.255.0   2                10.10.23.1
10.4.1.0          255.255.255.0   2                10.10.23.1
10.5.1.0          255.255.255.0   1                10.10.35.2
10.2.1.0          255.255.255.0   1                10.10.23.1
10.10.24.0        255.255.255.0   1                10.10.23.1
10.10.12.0        255.255.255.0   1                10.10.23.1
10.10.23.0        255.255.255.0   0                10.10.23.2
10.10.35.0        255.255.255.0   0                10.10.35.1
10.10.36.0        255.255.255.0   0                10.10.36.1
10.3.1.0          255.255.255.0   0                10.3.1.1

Spanning Tree

Source           Mask              Downstream Dependent Neighbors
10.8.1.0         255.255.255.0    10.10.36.2    10.10.35.2
10.7.1.0         255.255.255.0    10.10.36.2    10.10.35.2
10.4.1.0         255.255.255.0    10.10.35.2    10.10.36.2
10.10.48.0       255.255.255.0    10.10.35.2    10.10.36.2
10.10.47.0       255.255.255.0    10.10.35.2    10.10.36.2
10.6.1.0         255.255.255.0    10.10.35.2    10.10.23.1
10.5.1.0         255.255.255.0    10.10.23.1    10.10.36.2
10.3.1.0         255.255.255.0    10.10.35.2    10.10.36.2    10.10.23.1
10.10.36.0       255.255.255.0    10.10.35.2    10.10.23.1
10.10.23.0       255.255.255.0    10.10.35.2    10.10.36.2
10.10.12.0       255.255.255.0    10.10.35.2    10.10.36.2
10.10.24.0       255.255.255.0    10.10.35.2    10.10.36.2
10.2.1.0         255.255.255.0    10.10.35.2    10.10.36.2
10.10.35.0       255.255.255.0    10.10.36.2    10.10.23.1

Group Member Table

Interface:10.3.1.1
Groups: 225.0.0.5

Prune Table : Empty

```

route exchange component. The spanning tree at R1 is empty as downstream neighbors do not exist for any source in the network. The group table is empty, as there are no attached leaf interfaces. *Prune Table* also does have any entries as pruning has not started.

The spanning tree displays the list of downstream dependent neighbors for each source network/mask pair in the network. The *Group Table* indicates that a member of the group 225.0.0.5 is present on the interface 10.3.1.1. This is a result of the host H7 joining the group 225.0.0.5.

```

GLOBAL MEMORY OUTPUT AT ROUTER 5

Neighbor Table

Neighbor          Interface
10.10.35.1        10.10.35.2

Routing Table

Net              Netmask          Metric           NextHop
10.8.1.0         255.255.255.0   4                10.10.35.1
10.7.1.0         255.255.255.0   4                10.10.35.1
10.4.1.0         255.255.255.0   3                10.10.35.1
10.10.48.0       255.255.255.0   3                10.10.35.1
10.10.47.0       255.255.255.0   3                10.10.35.1
10.6.1.0         255.255.255.0   2                10.10.35.1
10.3.1.0         255.255.255.0   1                10.10.35.1
10.10.36.0       255.255.255.0   1                10.10.35.1
10.10.23.0       255.255.255.0   1                10.10.35.1
10.10.12.0       255.255.255.0   2                10.10.35.1
10.10.24.0       255.255.255.0   2                10.10.35.1
10.2.1.0         255.255.255.0   2                10.10.35.1
10.10.35.0       255.255.255.0   0                10.10.35.2
10.5.1.0         255.255.255.0   0                10.5.1.1

Spanning Tree

Source           Mask              Downstream Dependent Neighbors
10.5.1.0         255.255.255.0    10.10.35.1

Group Member Table

Interface:10.5.1.1
Groups: 225.0.0.5

Prune Table: Empty

```

The group table entry is the result of host H1 joining the group 225.0.0.5. As output from other routers are remarkably similar we do not show the output. This output verifies the correct operation of three *Multicast Routing stack* components: *Neighbor Discovery*, *Route Exchange* and *Spanning Tree* as the *Neighbor Table*, *Routing Table* and *Source Tree* entries are all correctly created and maintained.

To test functionality of the *Pruning* and *Grafting* components the following sequence of events were made to occur.

Initial State: We have an un-pruned tree rooted at the source R1 as shown in the figure 6. All the hosts have joined the group 225.0.0.5 and have started receiving data from the source.

Event A: H1 leaves group 225.0.0.5.

Observation: We observe changes in global memory at routers R5 and R3. We show group table and prune table contents only, as contents of other tables are not expected to change due to group joins and leaves. At router R5, the leaf interface 10.5.1.1 connecting H5 and H1 gets pruned for the (source, group) pair of (10.10.12.1,225.0.0.5) after H1 leaves. The group member table also deletes the membership entry.

At router R3, the core interface 10.10.35.1(interface connecting R3 and R5) gets pruned.

```

AT ROUTER R3:
AFTER H1 LEAVES GROUP 225.0.0.5

Group Member Table

Interface:10.3.1.1
Groups: 225.0.0.5

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
None|
Core Interfaces:
10.10.35.1      Pruned

```

```

AT ROUTER R5:
AFTER H1 LEAVES GROUP 225.0.0.5

Group Member Table

Interface:10.5.1.1
Groups: None

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.5.1.1      Pruned
Core Interfaces:
None

```

This is the result of the downstream router R5 sending a prune for the (source, group) pair of (10.10.12.1,225.0.0.5) upwards to R3.

Event B: H2 leaves group 225.0.0.5

Observation: we observe changes in group table and prune table entries at R6 and R3.

```
AFTER H2 LEAVES GROUP 225.0.0.5

AT ROUTER R3:

Group Member Table

Interface:10.3.1.1
Groups: 225.0.0.5

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
Core Interfaces:
10.10.35.1      Pruned
10.10.36.1      Pruned
```

```
AFTER H2 LEAVES GROUP 225.0.0.5

AT ROUTER R6:

Group Member Table

Interface:10.6.1.1
Groups: None

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.6.1.1      Pruned
Core Interfaces:
None
```

At router R6, the leaf interface 10.6.1.1 gets pruned, and the group member table deletes the entry for the group 225.0.0.5. At router R3, both the core interfaces 10.10.35.1 and 10.10.36.1 get pruned.

Event C: H7 also leaves the group 225.0.0.5.

Observation: We observe the effect of this leave on routers R3 and R2.

```
AFTER H7 LEAVES GROUP 225.0.0.5

AT ROUTER R2:

Group Member Table

Interface:10.2.1.1
Groups: 225.0.0.5

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces: None
Core Interfaces:
10.10.23.1      Pruned
```

```
AFTER H7 LEAVES GROUP 225.0.0.5

AT ROUTER R3:

Group Member Table

Interface:10.3.1.1
Groups: None

Prune Table

Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.3.1.1      Pruned
Core Interfaces:
10.10.35.1      Pruned
10.10.36.1      Pruned
```

At router R3, the leaf interface 10.3.1.1 gets pruned as a result of which R3 sends a prune upstream towards R2. The group member table is also updated deleting the membership

entry. At router R2, the core interface 10.10.23.1 gets pruned as a result of receiving a prune on that interface from downstream router R3. At this stage, the whole left-side of the tree is pruned. We now observed the effect of group leaves on pruning of trees and global memory contents. Events D and E are group re-joins. We shall observe its effect on grafting of trees next.

Event D: H1 re-joins the group 225.0.0.5

Observation: We observe the effect of this join at R3 and R2. The corresponding branches of the tree are grafted back.

```

AFTER H1 RE-JOINS GROUP 225.0.0.5
AT ROUTER R3:
Group Member Table
Interface:10.3.1.1
Groups: None
Prune Table
Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.3.1.1      Pruned
Core Interfaces:
10.10.35.1    Grafted
10.10.36.1    Pruned

```

```

AFTER H1 RE-JOINS GROUP 225.0.0.5
AT ROUTER R2:
Group Member Table
Interface:10.2.1.1
Groups: 225.0.0.5
Prune Table
Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces: None
Core Interfaces:
10.10.23.1    Grafted

```

At R3 and R2, we find that the core interfaces 10.10.35.1 and 10.10.23.1 are grafted respectively.

Event E: H2 re-joins the group 225.0.0.5

Observation: We observe the effect of join on routers R6 and R3.

```

AFTER H2 RE-JOINS GROUP 225.0.0.5
AT ROUTER R6:
Group Member Table
Interface:10.6.1.1
Groups: 225.0.0.5
Prune Table
Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.6.1.1      Grafted
Core Interfaces:
None

```

```

AFTER H2 RE-JOINS GROUP 225.0.0.5
AT ROUTER R3:
Group Member Table
Interface:10.3.1.1
Groups:
Prune Table
Source:10.10.12.1      Group:225.0.0.5
IGMP Interfaces:
10.3.1.1      Pruned
Core Interfaces:
10.10.35.1    Grafted
10.10.36.1    Grafted

```

At R6, the leaf interface gets grafted back again. At upstream router R3, both the core interfaces are now grafted. At this stage multicast traffic starts flowing to both H1 and H2. We have thus observed the effect of joins on the working of grafting component.

Testing the *Group Membership* stack for functionality is fairly simple. Just check if the leaf router's group table is updated for every host's join or leave event. The functionality of the data stack was verified using per-component log messages and monitoring traffic on the links using network sniffers like tcpdump. The very fact that data was delivered successfully from end-to-end proved most of the functionality. The multicast data stack is rigorously tested with various network metrics like throughput and latency. This is described in the next section.

5.2 Performance Testing

Functionality testing only proves that the components work as intended, but gives no indication on how fast or slow the stacks are. The multicast data stack is tested for performance based on network measurement metrics like end-to-end latency and throughput. Several performance measurements were made using our composite protocol stacks. The list of performance tests that were conducted is as follows. Each test experiment is explained in detail later with the results analyzed.

Test 1: Measurement of stack latencies at sender, router and receivers for the basic multicast data stack for varying message sizes. The results are tabulated and plotted.

Test 2: Measurement of per-component transmit and receive state machine latencies for all components of the basic multicast data stack for varying message sizes. The results are tabulated.

Test 3: Measurement of end-to-end one-way latency for the basic multicast stack. NTP was used to synchronize the machines. We plot the variation of one way latency with message size and number of hops.

Test 4: Measurement of end-to-end throughput for the basic multicast stack for varying message size.

Test 5: Measurement of end-to-end throughput for the reliable multicast stack for different link error probabilities. The results are tabulated as well as plotted.

Test 6: Measurement of join latency and leave latency. Join latency measurements were made for varying prune depth values.

The basic multicast stack consists of the components *MCAST_FORWARD*, *FRAGMENT*, *CHECKSUM* and *REPLICATOR*. The reliable multicast stack consists of the components *MCAST_RELIABLE*, *MCAST_FORWARD*, *UCAST_FORWARD*, *FRAGMENT*, *CHECKSUM*, *REPLICATOR* and *RANDOM DROP*. *RANDOM_DROP* is a component that simulates link error and drops packets with a user defined error probability of p .

Several factors were considered and changes made to make the components from merely functional to relatively high-speed, low delay units.

Some of them are listed below:

- *Choice of Ocaml compiler:* Using Ocaml high-performance native-code compiler *ocamlopt* instead of byte-code compiler *ocamlc*. The native-code compiler produces code that runs faster than the byte-code version at the cost of increased compilation time and executable code size. However, compatibility with the byte-code compiler is extremely high, the same source code should run identically when compiled with *ocamlc* and *ocamlopt*.

- Reducing the number of global memory lookups. On an average each global memory function lookup access time was measured to be about 20 μ s. A typical packet trace in the multicast forwarding component at a router made about 6-8 global memory function lookups. This induces lot of per-packet delay. To avoid such a high per-packet delay, it was decided to use fast-lookup caches inside the multicast forwarding component. These caches were part of the component's local memory. Global memory lookups are now not made for each and every packet, they are made only once in N packets, where N is called the global memory lookup frequency. Considering a packet flow of 1000 packets and a N value of 100, 990 packets would use values from the cache and only 10 packets would use actual global memory values. Caches are always refreshed once in N packets. For a highly stable network where there are not many route changes or group joins or leaves one would want to have a high value of N and for a highly dynamic network with lot of route changes and group joins/leaves, a low value of N has to be chosen. The uses of caches significantly improved forwarding delays at a router.
- *Order of guards*: The order in which the guards are executed at a particular state can also affect performance. It should be taken care that the most frequently occurring guard condition is executed first. This is because guards are evaluated only till the first true match is found.
- *Removing costly memory and file operations*: File operations are very costly and should be always removed if possible. Several costly memory operations were modified for better performance.

The individual tests are now explained in detail.

5.2.1 Test 1: Measurement of stack latencies

The stack latencies are measured at sender, router and receivers for the basic multicast data stack for varying message sizes. At the sender, the stack latency is defined as the time taken for an application packet, to traverse through the transmit state machines of the sender stack till its written onto the UDP/ETH socket. At the router, it refers to the total time spent in the Ensemble stack to forward a packet and at the receiver it refers to time elapsed between the reception of the packet from an ETH/UDP socket and delivery to the application.

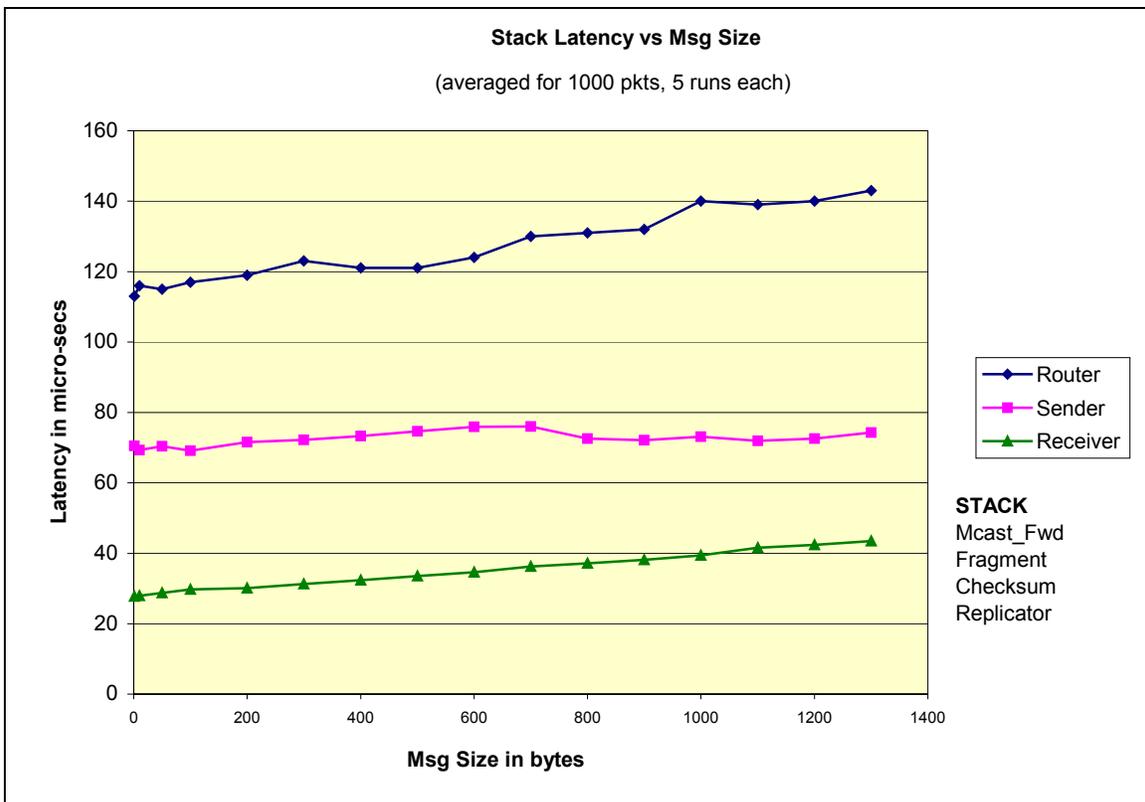


Figure 7: Variation Of Stack Latency With Message Size

The above plot shows how the stack latencies at the sender vary with message size. All values are computed after averaging over 1000 packets and 5 runs each. Message size is varied from 1 byte to 1300 bytes. From the graph, we find that on the whole, latencies increase with increase in message size. This fact is mainly attributed to the checksum component that is the only component in the stack whose performance depends on message size. At the sender, this result is not that evident. But at the routers we find a significant increase in latency from 113 μ s for 1 byte message to 143 μ s for a 1300-byte message. At the receiver it increases from 27.8 μ s to 43.5 μ s. The global memory lookup frequency was set to 100.

The results are also tabulated as under:

Stack Latency			
Msg Size	Sender	Router	Receiver
(bytes)	(in micro-seconds)		
1	70.53	113	27.72
10	69.33	116	27.94
50	70.45	115	28.79
100	69.18	117	29.72
200	71.57	119	30.06
300	72.23	123	31.31
400	73.3	121	32.40
500	74.68	121	33.53
600	75.97	124	34.64
700	76.01	130	36.20
800	72.61	131	37.13
900	72.11	132	38.15
1000	73.11	140	39.42
1100	72	139	41.62
1200	72.62	140	42.40
1300	74.27	143	43.46

Table 7: Variation Of Stack Latency With Message Size

5.2.2 Test 2: Measurement of Component Transmit and Receive Latencies

In this test, we measure the transmit and receive latencies of individual components in the multicast stack for different message sizes.

Msg	Component Latency (SENDER)			
Size	(in microseconds)			
(in bytes)	MCAST	FRAG	CHK	REPL
1	29.57	7.98	8.01	6.28
10	32.83	8.35	8.06	6.03
50	36.15	8.18	8.44	6.31
100	34.96	7.96	8.86	6.49
200	35.86	8.15	9.89	6.63
300	30.62	12.60	10.85	6.47
400	30.73	9.84	14.94	6.45
500	27.78	12.24	16.01	7.03
600	26.86	8.78	20.89	7.90
700	26.34	9.18	17.08	11.34
800	26.56	9.22	16.62	8.65
900	26.26	8.52	18.44	7.70
1000	26.09	8.23	20.49	7.61
1100	26.12	8.30	20.23	7.94
1200	25.96	7.98	19.94	7.03
1300	27.15	8.49	20.93	7.57

Table 8: Component Latencies At Sender

Msg	Component Latency (RECEIVER)			
Size	(in microseconds)			
(in bytes)	MCAST	FRAG	CHK	REPL
1	3.06	3.193	9.693	4.568
10	3.06	3.21	10.23	5.58
50	3.06	3.21	10.32	5.59
100	3.06	3.26	10.52	4.58
200	3.06	3.17	11.68	4.61
300	3.23	3.33	12.38	4.75
400	3.13	3.19	13.39	4.68
500	3.21	3.30	14.46	4.79
600	3.29	3.31	15.40	4.79
700	3.16	3.28	16.23	4.90
800	3.31	3.27	17.70	4.86
900	3.33	3.28	18.35	5.15
1000	3.26	3.37	19.41	5.04
1100	3.84	3.42	21.22	5.18
1200	3.30	3.26	21.70	5.12
1300	3.31	3.38	22.70	5.22

Table 9: Component Latencies At Receiver

From the results, we find that the checksum component’s latency increases significantly with message size, both at the sender and at the receiver. Other components do not show significant increase.

Test 3: Measurement of one-way latency

One-way latency is defined as the total time taken by the packet from the sender application to the receiver application. Before taking timing measurements, all machines have to be synchronized, so that the results reflect the correct values. NTP[18] was used to synchronize the machines. For each measurement the receiver and sender NTP offsets are also noted and are used while computing the net end-to-end one way latency. One-way latencies measurements were made for different message sizes and also by changing the number of hops. The following test set-up was used to measure one-way latencies upto 6 network hops.

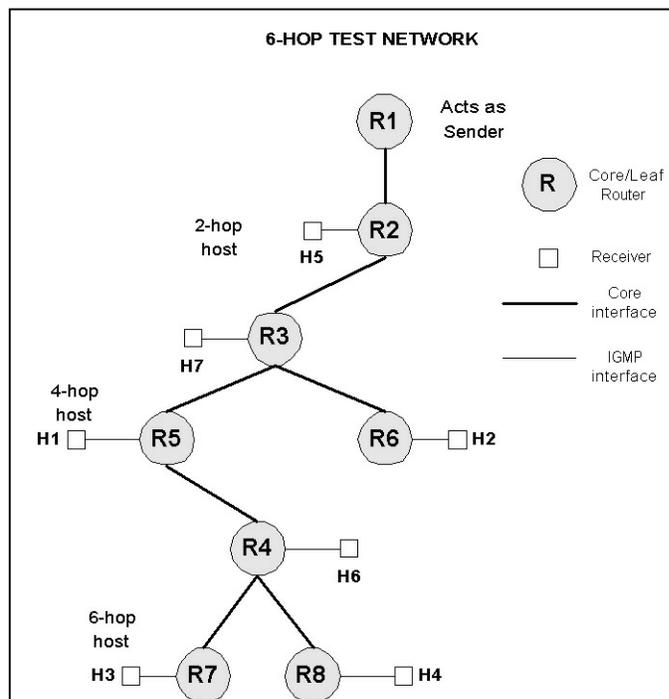


Figure 8: 6-Hop Test Network

Measurements are made at the sender R1, 2-hop host H5, 4-hop host H1 and the 6-hop host H3.

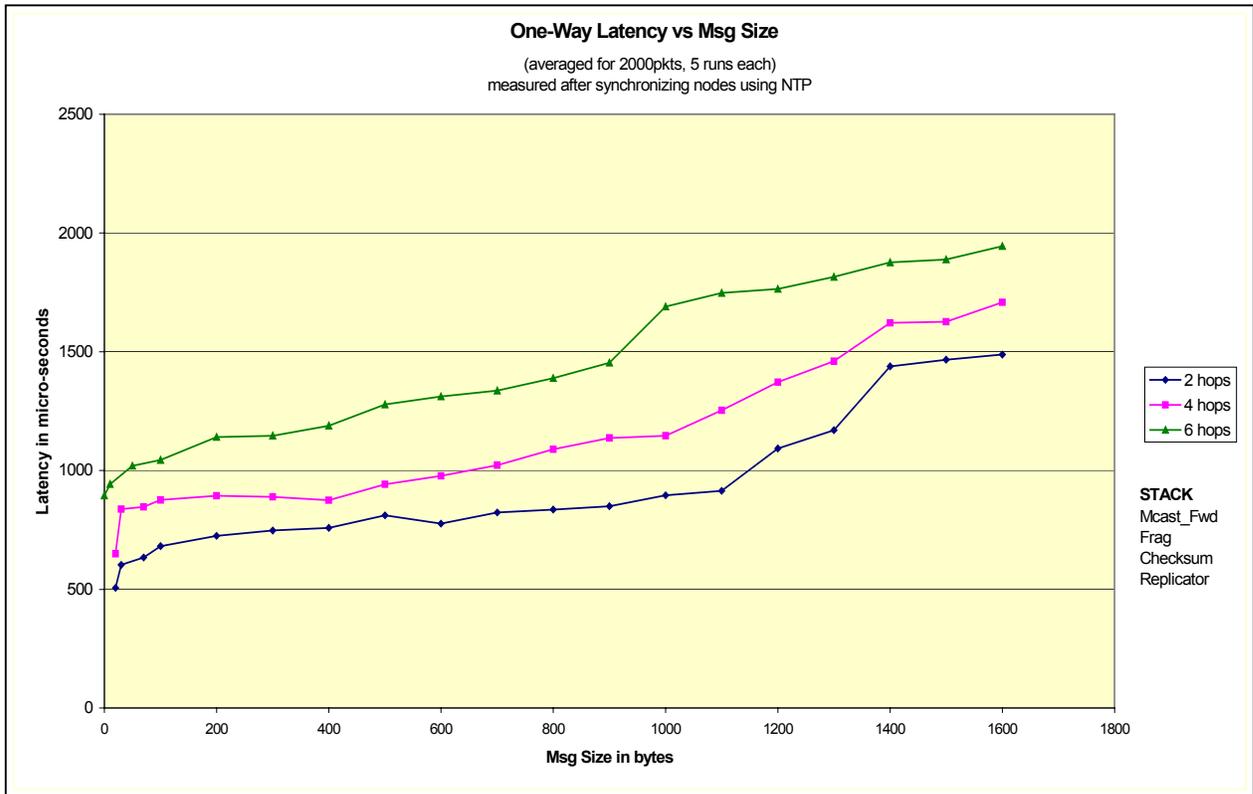


Figure 9: Variation Of One-Way Latency With Message Size

The plot shows how one-way latency varies with message size and number of hops.

As expected the end-to-end latency values increase with increase in message size and increase in number of hops. The values are tabulated as under.

Msg	End-to-End Latency		
Size	(in microseconds)		
(in bytes)	2-hop	4-hop	6-hop
20	506	650	895
30	603	837	943
70	634	847	1020
100	682	876	1044
200	725	894	1141
300	748	889	1147
400	759	875	1189
500	811	942	1278
600	777	978	1312
700	824	1023	1336
800	836	1089	1389
900	850	1137	1454
1000	896	1147	1690
1100	915	1254	1748
1200	1093	1372	1765
1300	1170	1460	1815
1400	1439	1622	1876
1500	1467	1626	1889
1600	1489	1708	1945

Table 10: Variation Of End-To-End Latency With Hops

The message sent from the sender consists of a 20-byte timestamp followed by a variable length message field. So the minimum message size is 20-bytes.

5.2.3 Test 4: Measurement of end-to-end throughput

End-to-end throughput refers to receiver throughput, which is defined as follows:
Throughput in bits/sec = (No of bytes received * 8) / (T_{last} - T_{first}) secs, where, T_{last} is the time when the last packet is received and T_{first} is the time when the first packet is received. End-to-end throughput values were measured for 2 stack combinations, a stack with only MCAST_FORWARD and REPLICATOR and for the basic multicast stack.

The throughput values were measured at 4 receivers H1, H2, H3 and H4 each 4 hops away from the multicast source R1, values obtained are averaged. As we do not

have a flow control component the sender needs to be slowed down if the receiver is not able to sustain the sender rate. A sender slow-down factor of 70 was used for all the measurements.

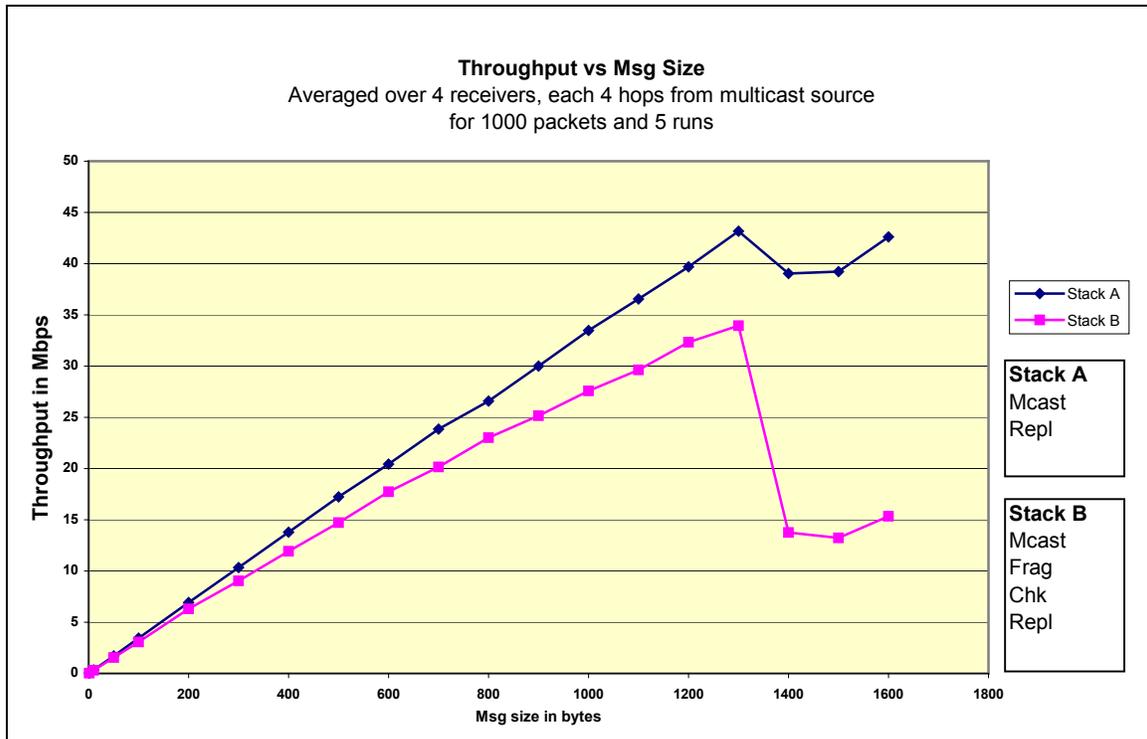


Figure 10: Variation Of Throughput With Message Size

We find that for both curves the throughput increases with increase in message size from 1 byte to 1300 bytes. Stack A does not have our *FRAGMENT* component, so IP fragmentation comes into effect after 1300 bytes. Stack B has the *FRAGMENT* component in it. We find a steeper drop in Stack B curve compared to Stack A curve after 1300 bytes. This is due to the difference in performance of our fragment component and IP fragmentation. We find that addition of Checksum and Fragment in Stack B has resulted in a decrease in throughput. We achieve the highest throughput of 43.17 Mbps

for 1300-byte sized message for Stack A and a highest throughput of 33.9 Mbps at 1300 bytes for Stack B. The increase in throughput for both the curves is also very consistent.

The individual values are tabulated as under:

Msg size (in bytes)	Throughput (in Mbps)	
	Stack A	Stack B
1	0.035	0.033
10	0.347	0.306
50	1.7	1.539
100	3.433	3.087
200	6.93	6.298
300	10.357	9.041
400	13.789	11.938
500	17.241	14.716
600	20.437	17.74
700	23.861	20.151
800	26.579	23.017
900	30.003	25.157
1000	33.484	27.568
1100	36.546	29.622
1200	39.678	32.343
1300	43.172	33.935
1400	39.051	13.748
1500	39.237	13.242
1600	42.596	15.349

Table 11: Variation Of Throughput With Message Size

5.2.4 Test 5: Measurement of throughput for reliable multicast

The reliable multicast stack consists of 7 components viz. Mcast_Reliable, Mcast_Forward, Ucast_Forward, Fragment, Checksum, Replicator and Random Drop.

Throughput for the reliable multicast stack was measured by varying link error rates using the Random Drop component. The values were measured at receivers H1, H2, H3 and H4 which are 4-hops from the multicast source. 1000 packets were transmitted from source each with packet size of 1000 bytes. A 1% error probability implies that out of 1000 packets, 990 packets are reliably transmitted and 10 are re-transmitted from the

source. NACK status packets if any, are sent from all receivers every 10ms. Re-transmissions at the sender also take place every 10ms. A daily timer interval of 30s is used. The multicast re-transmission threshold was set at 2 i.e. if 2 or more receivers request a packet to be re-transmitted it will be multicast on the network, else re-transmissions are separately unicast back to each receiver.

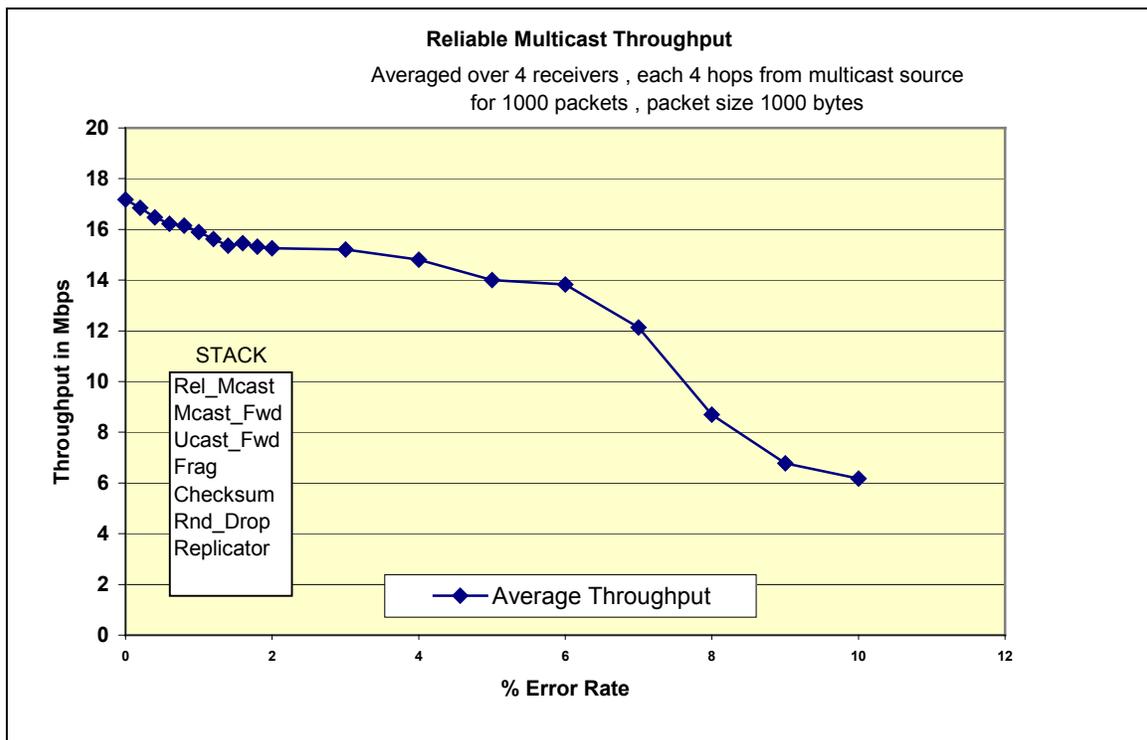


Figure 11: Variation Of Reliable Multicast Throughput With Error Rate

For a 0% link-error probability (ϵ), a throughput of 17.18 Mbps is achieved. For 1000 bytes the basic multicast stack gave a throughput of 27.57 Mbps (previous test result). This reduction can be attributed to the addition of 3 more components in the stack and buffering operations at the reliable component’s sender. No reverse ACK flow occurs here and there are no-retransmission too. The throughput only decreases gradually from 17.18 Mbps to 13.82 Mbps at an error probability of 6%. A 6% error probability in the

link leads to about 60 retransmissions from source. There is a much steeper decrease from 6% to 10% and the throughput drops to 6.18Mbps. On the whole, the throughput values are good even for high error rates. The individual values are tabulated as under:

Error	Throughput
%	(Mbps)
0	17.17
0.2	16.84
0.4	16.47
0.6	16.23
0.8	16.15
1	15.9
1.2	15.62
1.4	15.35
1.6	15.46
1.8	15.33
2	15.26
3	15.2
4	14.8
5	14.01
6	13.83
7	12.17
8	8.69
9	6.78
10	6.18

Table 12: Variation Of Reliable Multicast Throughput With Error Rate

5.2.5 Test 6: Measurement of join and leave latency

Join Latency is defined as the time taken for a receiver host to start receiving data from the source after it has joined the corresponding group. Join Latency can be controlled by adjusting the values of the query timer and the graft timer and it is also dependent on prune depth (how far the tree is pruned).

The following sequence of operations occur after a host joins a group:

- The local group cache is first updated, a report packet is sent to the leaf router on receiving a query and the global memory group table gets updated at the leaf router. Let the time taken for this sequence be T1.
- On expiry of the graft timer, the grafting component sends a graft message upstream, which then grafts all interfaces till either an un-pruned branch is reached or till the source is reached. Let this time be T2.
- Then, data has to flow from that node back to the receiver. Let this time be T3.

The join latency is the sum $T1 + T2 + T3$ approximately.

Join latency was measured for 3 cases, for prune depth of 1, 2 and 3.

The prune timer, graft timer and the query timer were all set to 100ms. The sender data rate was set to 10 packets/sec. The following figure shows all the 3 cases:

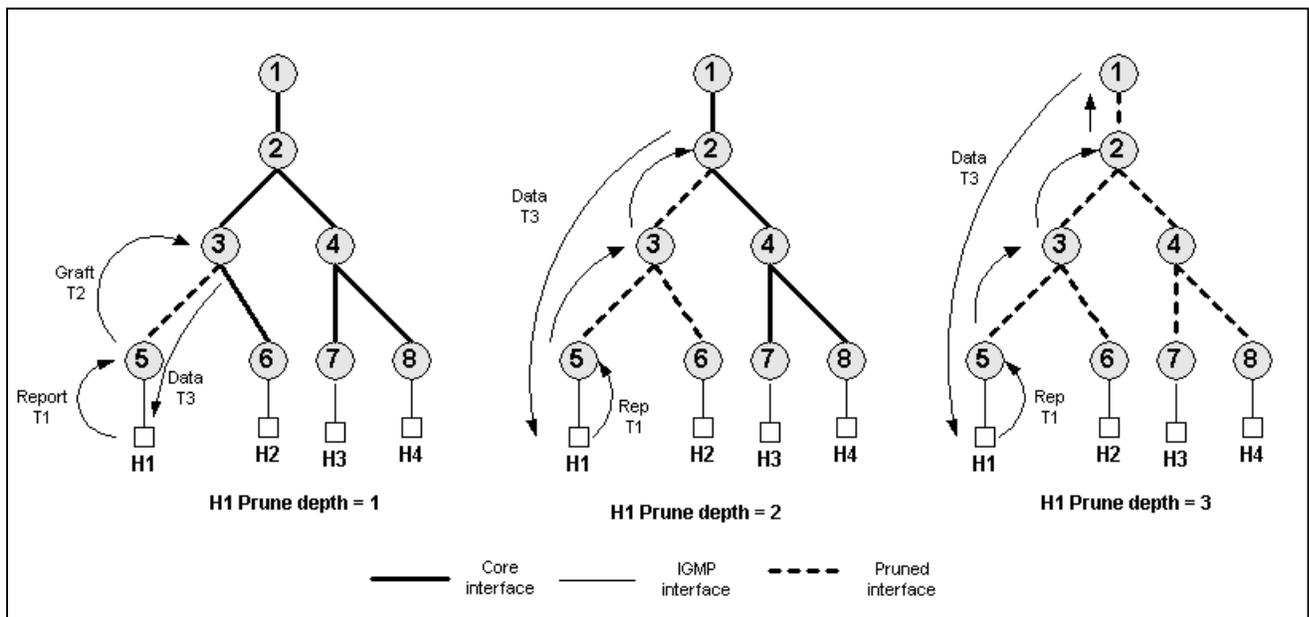


Figure 12: Prune Depth Of A Multicast Tree

The join latency test results are tabulated as under:

Prune Depth	Average Join Latency (in milli-seconds)
1	405
2	458
3	535

Timer	(seconds)
Query	0.1
Graft	0.1
Prune	0.1

Table 13: Variation Of Join Latency With Prune-Depth

As expected we find that join latency increases with increase in prune depth. However, it should be noted that join latency is very controllable and can be affected due to change in any of the above timer values. Making the timers expire more frequently will definitely improve join latency but will also increase the amount of traffic in the links because more number of query, prune and graft messages will be sent.

Leave latency is defined as the time taken for the receiver to stop receiving data after it has left the corresponding group. Leave latency just depends on the query timer interval. For a query timer interval of 100ms, a leave latency of 146ms was obtained.

Leave latency can also be improved by increasing the query timer frequency at the cost of more link traffic. Both leave and join latency valued reported above are averaged over 5 runs.

We have thus described the functionality tests and performance tests that were performed on the multicast composite protocols.

5.3 Comparison with Linux IP Multicast

The throughput values attained by the composite protocol implementation are compared with those using Linux IP multicast. Mrouded[19], the Linux IP multicast implementation for DVMRP was used on the same test network. Mrouded was installed on all router (R1 to R8). Iperf [20] was used to measure the end-to-end multicast throughput.

Throughput measurements were made for varying packet sizes ranging from 10 to 2000 bytes. The sender is made to send at a maximum possible data rate, so that there is no receiver loss. 1000 packets are sent in each throughput measurement test. The throughput increases from 2.81 Mbps for a 10-byte packet to 95.8 Mbps for 1400 byte packet. There is a sheer drop of throughput at around 1500 bytes due to IP fragmentation. Figure x illustrates the end-to-end throughput performance of Linux IP multicast and the basic Composite Protocols multicast data stack.

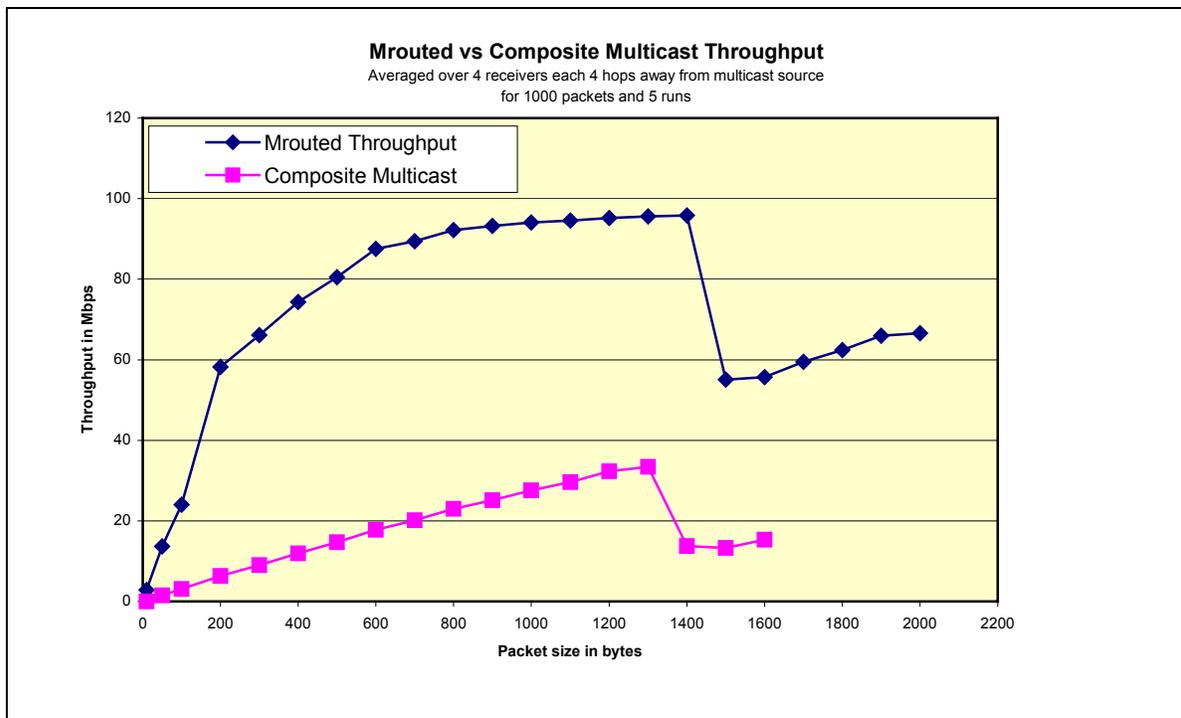


Figure 13: Comparison With Linux IP Multicast Throughput

The composite multicast achieves a highest throughput of 34 Mbps compared to its Linux counterpart, which achieves about 95 Mbps for packet sizes of 1300 bytes. The fact that the composite protocol implementation is about 2-3 times slower is not surprising.

Msg size (in bytes)	Throughput (in Mbps)	
	Linux IP Multicast	Composite Protocols
10	2.81	0.306
50	13.7	1.539
100	24	3.087
200	58.2	6.298
300	66.1	9.041
400	74.3	11.938
500	80.5	14.716
600	87.5	17.74
700	89.4	20.151
800	92.2	23.017
900	93.2	25.157
1000	94.1	27.568
1100	94.6	29.622
1200	95.2	32.343
1300	95.6	33.935
1400	95.8	13.748
1500	55.1	13.242
1600	55.7	15.349

Table 14: Comparison With Linux IP Multicast

Given the constraints imposed by the specification methodology and limitations of the current implementation, this is a reasonable performance penalty to pay. A few reasons are:

- Executing a component's state machine incurs a non-trivial amount of overhead, which the in-kernel implementation in Linux does not.
- There are no well-defined boundaries between layers in the Linux implementation with respect to memory access and all layers operate on a common instance of a socket buffer. Linux protocol software can afford to perform pointer arithmetic on socket buffers and minimize memory copies. The strict layering enforced by the composite protocol framework makes it impossible to access the local memory of another component.

- Moreover, Ensemble is a user-level program and hence incurs further overhead in sending and receiving messages compared to the Linux in-kernel implementation.
- Finally, the Linux implementation has matured over many years of use and improvement, whereas only limited time could be spent so far in optimizing the current implementation of composite protocols.

6. Summary and Future Work

This thesis presents a novel approach of building network services from composite protocols consisting of single-function protocol components. It demonstrates the applicability of the composite protocol approach to wider-range of network protocols and services, both data-oriented/data plane and control-oriented/control plane protocols can be built and composed into stacks using this approach. This thesis addresses one of the main challenges in building network services, inter-stack and cross-protocol communication that is addressed through use of global memory objects.

As a case study, a reliable multicast service is built using three composite protocol stacks and 5 global memory objects. A multicast data stack for reliable replication of data in the network, a multicast routing stack for dynamically creating and maintaining neighbor tables, routing tables, spanning trees in the network and a group-membership stack for members to join/leave multicast groups in an ad-hoc fashion. The global memory objects are implemented as part of shared memory which link to the stacks at run-time. They provide a functional interface and simultaneous access to them is controlled using semaphores.

The reliable multicast service is also tested for both functionality and performance on a medium sized 12-node test network. The functionality tests confirm the expected behaviour of the stacks, including dynamic pruning and grafting of stacks. Performance tests measured end-to-end throughput, one-way latency, reliable-multicast throughput and individual per-component send and receive latencies. The performance of composite reliable multicast is also compared to Linux IP multicast.

6.1 *Future Work*

This section suggest possible improvements and enhancements to this thesis and to the area of composite protocols and services in general and identifies scope of future work in this area.

- The multicast service designed and implemented here supports only point-to-multipoint data transfer used in applications like file-transfer and audio streaming. This can be extended to support multi-point to multi-point multicast which can be used in applications like video-conferencing.
- Complex multicast protocols like MOSPF and PIM can be implemented using this approach.
- More composable services can be built , security protocols ,network management protocols can be built to test the feasibility, demonstrate component re-use and expand the library of components.
- The main focus of this thesis was to focus on demonstrate the feasibility of the composite protocol approach to design and implement network services, performance was not the major focus. A lot of work can be done to improve and optimize the performance of these composite protocol stacks and make them come into speed with IP based implementations.
- Deployment of composable services on an active network is another big challenge.
- Automating the process of verifying specification of components, tools to automatically translate from specification to implementation, a Property-In Protocol Out conversion tool are also possible areas of improvement.

Bibliography

- [1] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In IEEE OPENARCH, April 1998.
- [2] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In Proceedings of the International Conference on Functional Programming Languages. ACM, 1998.
- [3] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, S. Sheth, H. Pindi, F. Wah-hab, A. Gopinath, and A. Nagarajan. Implementation of a Prototype Active Network. In OPENARCH '98, 1998.
- [4] G. J. Minden, E. Komp et al, "Composite Protocols for Innovative Active Services", DARPA Active Networks Conference and Exposition (DANCE 2002), San Francisco, USA, May 2002.
- [5] ISO, "Information Processing Systems - OSI Reference Model - The Basic Model", ISO/IEC 7498-1, 1994.
- [6] T. Pusateri, "DVMRP version 3," draft-ietf-idmr-dvmrp-v3-10, August 2000.
- [7] J. Moy. Multicast Extensions to OSPF. Internet Requests For Comments (RFC) 1075, Mar. 1994.
- [8] Deering, Estrin, Jacobson et al, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Motivation and Architecture" draft-ietf-idmr-pim-arch-01.ps , Internet Draft.
- [9] W. Fenner, "Internet Group Management Protocol, Version 2", RFC 2236, Xerox PARC, November 1997.

- [10] Yuri Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms," ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.
- [11] M. Hayden, "The Ensemble system", Ph.D. dissertation, Cornell University Computer Science Department, January 1998.
- [12] J. C. Lin and S. Paul, "RMTP: A reliable multicast transport protocol," in Proc. IEEE Infocom, pp. 1414--1425, March 1996.
- [13] C.Hedrick. Routing Information Protocol. RFC 1058, June 1988.
- [14] J. Moy, OSPF Version 2, Internet Request for Comments, RFC 2178, July 1997.
- [15] X. Leroy, "The Objective Caml system, release 3.04", Documentation and user's manual, INRIA, France, December 2001.
- [16] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Third International Symposium on Programming Language Implementation and Logic Programming, number 528 in Lecture Notes in Computer Science, pages 1-13, Passau, Germany, August 1991.
- [17] Mills, D. L. Network Time Protocol (version1) specification and implementation. DARPA-Internet Report RFC-1059, DARPA, 1988.
- [18] B. Fenner. "The multicast router daemon - mrouterd,"
<ftp://ftp.parc.xerox.com/pub/net-research/ipmulti>.
- [19] Distributed Application Support Team, "Iperf",
<http://dast.nlanr.net/Projects/Iperf>