The University of Kansas

Information and Telecommunication Technology Center

Technical Report

Innovative Active Networking Services: Final Report

Gary J. Minden, Joseph B. Evans, Ed Komp,
Ravi Shankar, Vishal Zinjuvadia, Suresh Krishnaswamy,
Magesh Kannan, and Sandeep Subramaniam

ITTC-FY2004-TR-19740-12

September 2003

Project Sponsor:
Defense Advanced Research Projects Agency
U.S. Air Force Research Laboratory
Contract: F30602-99-2-0516

Innovative Active Networking Services

Table of Contents

Innovative Active Networking Services

Figures

Innovative Active Networking Services

Tables

Innovative Active Networking Services
Final Report

1 Project Introduction

Traditional networking implementations follow a layered protocol model that implements a well-defined protocol stack. Major functionality is built into the software and hardware of end-point hosts and the network switches and routers. The entire complex of end-point host system software, network switch software, and networking services evolves slowly due to need to reach consensus prior to widespread changes and improvements in the protocols. Active Networking was developed to bring robust, rapid, flexibility to developing and deploying new network services and protocols.

Our research project covered three areas:

(1) Create an architecture and implementation for Innovative Active Networking Services based on Composite Protocols,

(2) Establish criteria and evaluation measures for Active Network performance, and

(3) Implement Active Networking services on end-point host systems based on the Windows NT and Linux operating systems.

1.1 Active Networking Services Architecture

Our architecture for Innovative Active Networking Services (IANS) is based on three concepts:

(1) Composite Protocols — we identify functional units of conventional, existing, and proposed protocols that are easily combined with new application oriented protocols.

(2) Global Memory — we define a functional interface for the controlled sharing of information among independent protocols that collectively perform a network service.

(3) Robust, Distributed Network Services — we identify new mechanisms to easily distribute network services in a reliable and robust manner.

The problem of dynamic, on-the-fly customization of the intermediate network nodes is addressed through Active Networking. However, while the basic concept of Active Networking allows for end-users to design their own "boutique" protocols, we expect that most innovative protocols will encompass many functional components with known networking protocol properties. For example, we expect that most protocols will require some mechanism of assured delivery. We expect that many protocols will require sequential data delivery. We expect that many protocols will require custom routing mechanisms. And, many protocols will require authentication and privacy.

Previous work on protocol composition such as, the x-kernel [Hutchinson1991], Horus [vanRenesse1995], and Kanga [Burke1996], focused on decomposing protocols into modules implementing small pieces of functionality, and the composition of these modules into flexible, modular protocol frameworks. However, the activation of individual protocol modules is based on the flow of protocol data units through the established protocol stack. One therefore needed to decide the order of functional modules, i.e. do privacy functions come before or after assured delivery? In addition, these early efforts could not count on an Active Networking infrastructure and had to assume a fixed internet services structure between end-point hosts. Active Networking breaks these assumptions.

Composite Protocols will be built around the following concepts:

- Express network communication functions in terms of single-function protocol components, with a uniform interface, enabling complex protocols to be built by composing selected protocol components. Protocol components will be described with a set of properties to enable describing protocol compositions.
- Develop a composition operator to combine multiple protocol modules. Our approach uses a very simple and regular stacking of protocol modules leading to a standard encapsulation operator.
- Develop formal functional specifications for protocol components. Formal specification explicitly states the properties of a protocol component and its interaction behavior with external components. Explicit specification enables automatic evaluation of composite protocol properties.
- Develop and evaluate a set of Composite Protocols based on an initial set of protocol components that includes: error detection, packet retransmission (several types), segmentation and reassembly (e.g. TCP PDUs and packet ordering), next hop routing, credit-based flow control, simple privacy and authentication protocols.

For the implementation of specialized endpoint protocols, communication is confined to the protocol components comprising a specific protocol. We have carefully defined the lines of communication among the components in a stack, in order to aid in verification of implementation correctness. However, when considering network services communication among protocol components becomes much more complicated, and in many current implementations largely ad hoc. A network service, such as multicast delivery, requires communication among protocol components in different protocol stacks. We have defined another level of memory access, global memory, for protocol components. A global memory is defined by a purely functional interface. The implementation and execution monitor for a global memory are completely independent of any specific protocol that requires access to the shared information.

As new services become available within the Active Network, there is the requirement to establish robust, survivable distributed services. Proxy information servers [Kulkarni1999] are one example. We envision survivable servers with the following initial functionality:

- When launched into the Active Network, they attempt to identify near-by cohorts,
- If other cohorts are found, they join the server group and enter either an operational or standby state. If no other cohorts are found, the server replicates itself at another Active Node (within defined constraints),
- Operational servers announce themselves via an "information routing beacon" to nearby Active Nodes. End-point host service requests will be routed to nearby operational servers. Standby servers periodically monitor their nearby operational servers, and if the operational server is running, no action is taken. If the operational server does not respond, the standby server switched to operational status.

These kind of robust, survivable and distributed services should be easy to construct given the Composite Protocols and Protocol class libraries described above.

2 Project Tasks

Our project was divided into two major tasks. The first task focused in implementing a NodeOS and execution environment that would be simple to check for security properties and demonstrate reasonable performance. The second task focused on designing, implementing, and testing a composite protocol architecture. These tasks are described below. Details of this work are described in the technical reports listed below. These reports are available from The University of Kansas Information and Telecommunications Center and have been submitted to the sponsor.

ITTC-FY2003-TR-19740-05	Magesh Kannan, Ed Komp, Gary Minden, and Joseph Evans	Design and Implementation of Composite Protocols	February 2003
ITTC-FY2003-TR-19740-06	Stephen Ganje, Ed Komp, Gary Minden, and Joseph Evans	Stack Local Packet Memory Interface Requirements	February 2003
ITTC-FY2003-TR-19740-07	Ravi Shankar, Gary Minden, and Joseph Evans	Implementation and Performance of an Integrated OCaml-Based Active Node	February 2003
ITTC-FY2003-19740-08	Suresh Krishnaswamy, Joseph B. Evans, and Gary J. Minden	A Prototype Implementation for Dynamically Configuring Node-Node Security Associations using a Keying Server and the Internet Key Exchange	February 2003
ITTC-FY2003-TR-19740-09	Vishal Zinjuvadia, Gary Minden, and Joseph Evans	Designing a Framework for Dynamic Deployment of Network Services in an Active Network Domain	February 2003
ITTC-FY2003-TR-19740-10	Yoganandhini Janarthanan, Gary Minden, and Joseph Evans	Enhancement of Feedback Congestion Control Mechanisms by Deploying Active Congestion Control	February 2003
ITTC-F2004-TR-19740-11	Sandeep Subramaniam, Ed Komp, Gary Minden and Joe Evans	Building a Reliable Multicast Service Based on Composite Protocols	July 2003

Table 1 lists the IANS Technical Reports

Task 1: Design and Implement an Integrated Active Switching Node

Task 1 focused on integrating a number of Active Networking components into a "Standard Active Switching Node" (SASN). Our goal was to combine a NodeOS, execution environment, and active networking services on a stock personal computer. The intent was that the NodeOS would be simple enough to determine security properties by inspection and the combined software and hardware would facilitate distributing active nodes to other institutions.

We assembled a Standard Active Switching Node from the following components:

1. Utah OS Toolkit [Ford1997]: The OS Toolkit offered a simple operating system with small memory footprint;
2. OCaml [Remy1999]: OCaml is an implementation of the ML programming language and runtime system. OCaml offers a programming environment that can be formally analyzed;
3. University of Pennsylvania PLAN [Hicks1998]: PLAN uses OCaml to implement a rigorous active networking execution environment; and
4. Experimental active networking services.

The Section 2.1 describes our work in implementing a SASN. Security was also a capability we built into our Active Network. Section 2.2 discusses our security architecture.

Project Task 2 focused on developing innovative active networking services. Section 2.3 describes our composite protocol components. The work of composite protocols is applied to the reliable multi-cast problem and described in Section 2.4.

2.1 An Integrated Active Node

The PLAN system is part of the SwitchWare [Alexander1998] active networking project at the University of Pennsylvania. Its architecture contains two basic components. It defines a high level interoperable layer wherein lie the active packets based on a new language called PLAN. Below this layer, exists a much stable layer that provides the node-resident services. These node resident services can be written in a general purpose programming language such as Java or OCaml. A typical PLANet (PLAN Active network) node would look as in Figure 1.

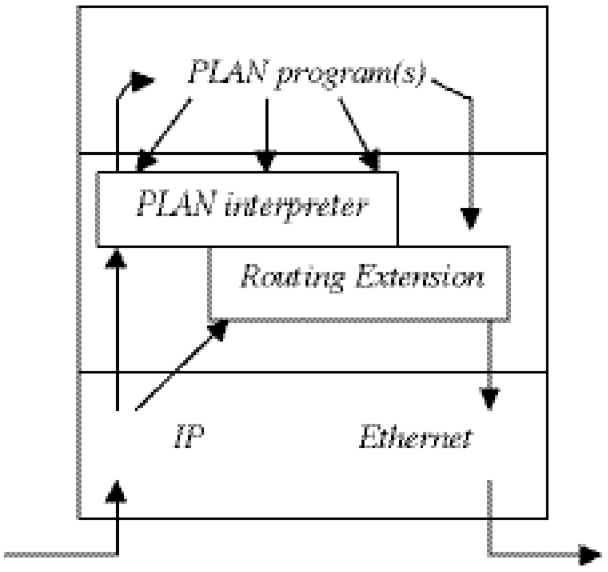


Figure 1 A typical PLAN Active Node

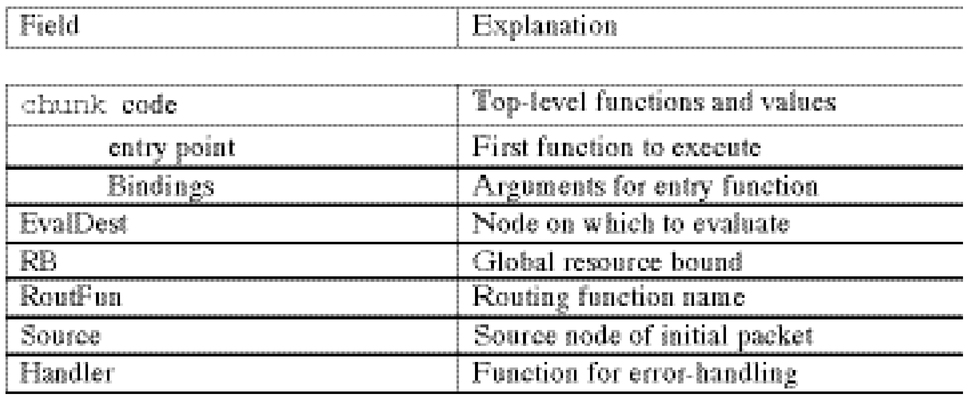
The PLAN architecture [Hicks1998] is designed to provide programmability at two levels. It can support both programmable (or active) packets and downloadable router extensions. This basic structure, as discussed below, follows a model of distributed computing that is based on remote evaluation. This sort of communication is asynchronous and unreliable. The extendible router infrastructure is used to provide support to heavyweight operations.

These extensions can be dynamically installed, but are not mobile once installed.

2.1.1 The PLAN language

An active networking approach must tread a fine line among the following issues: *flexibility*, *safety and security*, *performance* and *usability*. Increased flexibility is the primary motivation for active networks. PLAN does not have to be too general because it adopts a two-level approach. This is because the service language helps in providing general-purpose functionality. Hence PLAN has been able to express itself in 'little' programs and acts as glue between router resident services. By safety we mean reducing the risk of mistakes or unintended behavior, and security encompasses the concept of privacy, integrity and availability in the face of malicious attack. To address some of these issues, PLAN was made a functional, stateless, strongly typed language. This ensures that PLAN programs are pointer-safe and concurrently executing programs do not interfere with each other. Since network operations involve changing the state of nodes in some way, some sort of authentication is required. However, packet-based authentication is very costly and hence PLAN pushes these features on to the node-resident services. PLAN programs are statically typed and are guaranteed to terminate as long as they use services which terminate. Basic error-handling facilities are also provided. These help in improving the usability of PLAN programs.

PLAN design is based on remote evaluation, rather than on remote procedure call. Specifically, child active packets may be spawned and asynchronously executed on remote nodes. Each packet may further create packets provided it adheres to a global resource bound. The PLAN packet format is shown in Figure 2.



Field	Explanation
chunk code	Top-level functions and values
entry point	First function to execute
Bindings	Arguments for entry function
EvalDest	Node on which to evaluate
RB	Global resource bound
RoutFun	Routing function name
Source	Source node of initial packet
Handler	Function for error-handling

Figure 2 illustrates the PLAN packet.

The primary component of each packet is its *chunk* (code *hunk*), which consists of code, a function to serve as an entry point, and values to serve as bindings for the arguments of the entry function. The `EvalDest` field specifies the active node on which this packet is to be evaluated. The packet is transported to the `EvalDest` active node by means of a routing function specified by the field `RoutFun`. The resource bound field specifies the total number of hops the packet and its subsequent child packets can take before evaluation. The source and handler fields represent the source node of initial packet and the handler function for error handling respectively. A host application constructs the PLAN packet. It than injects it into the local PLAN router via a well-known port, say 3324. Remote execution is achieved by making calls to network primitives such as `OnRemote` or `OnNeighbor`. These are services written at the lower level in a general-purpose programming language. A PLAN program can be better explained by a simple *Ping* example.

```
fun ping (src:host, dest:host) : unit =
  if (not thisHostIs(dest)) then
    OnRemote(|ping|(src,dest), dest, getRB(), defaultRoute)
  else
    OnRemote(|Ack| (), src, getRB(), defaultRoute)

fun ack() : unit = print("Success")
```

Figure 3 lists the PLAN code for ping.

This program is placed in an active packet that executes ‘ping’ at the source. The arguments for the ping function include the source and destination active nodes. The program then proceeds to execute as follows: If the packet is not at the destination, the OnRemote call is activated which creates a new packet and sends it over to the destination using the defaultRoute, which is RIP here. Once the packet is on the destination, it further invokes another OnRemote that executes an 'ack' function at the source. This completes the required operation for ping.

The PLAN language is characterized by the semantic basis provided by the theory of lambda calculus. However, in order not to compromise on security, PLAN does not include many features common to functional languages. In keeping with this idea, PLAN has simple programming constructs: statement sequencing, conditional execution, iteration over lists with *fold* and exceptions. The lack of recursion and unbounded iteration (as well as the monotonically decreasing resource bound) ensure that all PLAN programs terminate. Its type system is strong and statically typed but is dynamically checked. This arises from the necessity of distributed programming wherein static checking is necessary for debugging purposes whereas dynamic checking ensures safety of the code. It also helps that PLAN does not provide user-defined mutable state, although some aspects of PLAN, such as the resource bound, maintain state. In addition to the general exception based error handling mechanism, PLAN also provides an `abort` service that allows the program to execute a *chunk* on the source node. A major feature of PLAN is that chunks can be encapsulated in one another providing for protocol layering within PLAN.

Another issue, which we have been digressing from, is the choice of implementation language for PLAN at the service level. Such a language must to able to provide services to make code dynamically loadable. To enable such modules to work on heterogeneous types of machines, it must be portable. Thirdly, in order to provide guarantees for safe execution and termination it must be a safe language. PLAN has been implemented in OCaml [Remy1999] and the Pizza extension to Java. Our efforts at an integrated node revolve around the use of OCaml as a safe language. Hence we will be discussing it alone.

2.1.2 OCaml

OCaml provides several of the design goals required for a service level language. Some of these have been outlined above. It has been developed at INRIA, Rocquencourt within the "Cristal project" group. The OCaml language system is an object-oriented implementation of the Caml dialect of ML.

All programming in OCaml is dominated by the use of functions. These first-class functions can be passed to other functions, received as arguments or returned as results. A powerful type system is another inherent feature of OCaml. It comes along with parametric polymorphism and type inference. Functions also may have polymorphic types. It is possible to define a type of collections parameterized by the type of elements, and functions operating over such collections. For instance, the sorting procedure for arrays is defined for any array, regardless of the type of its elements.

OCaml provides several common data-types such as int, float, char and string. Other data-types include records and variants that are standard features of functional languages. OCaml's type system is extendable by using user-definable data-types. New recursive data-types can be defined as a combination of records and variants. More importantly, functions over such structures can be defined by pattern matching: a generalized case statement that allows the combination of multiple tests and multiple definitions of parts of the argument in a very compact way.

OCaml is a safe language. The compiler performs many type checks on programs during compilation. That's why many programming errors such as data type confusions, erroneous accesses into compound values cannot happen in Caml. The compiler carefully verifies all these points, so that data accesses can be delegated to the compiler code generator, to ensure that data manipulated by programs may never be corrupted. The perfected integrity of data manipulated by programs is hence granted for free in Caml. These features are extremely important for an active network where mobile code generated at one end gets evaluated on other machines.

Another feature that is of big importance to the active networking community would be the strong type safety provided by OCaml. OCaml is statically type-checked, hence there is no need to add type information in programs (as in Pascal, or C). Type annotations are fully automatic and handled by the compiler.

The OCaml compiler frees the programmer of all memory management. All memory allocations and de-allocations are handled automatically by the compiler. This way the programs are much safer, since spurious memory corruption never occurs. The memory manager works in parallel with an application, thereby improving the efficiency of execution of OCaml bytecodes.

In addition to these features, OCaml provides an expressive class-based object oriented layer that includes traditional imperative operations on objects and classes, multiple inheritance, binary methods, and functional updates. Error handling is achieved by an exception mechanism as in other object-oriented languages.

The OCaml distribution comes with general-purpose libraries that facilitate arbitrary precision arithmetic, multi-threading, and graphical user interfaces, etc. It also offers Unix-style programming environment including a replay debugger and a time profiler. Objective Caml programs can easily be interfaced with other languages, in particular with other C programs or libraries. This feature has been extensively used while building our wrapper around the Moab C libraries.

2.1.3 OSKit

The Flux research group at the University of Utah has developed OSKit. It provides a set of modularized libraries with straightforward and documented interfaces for the construction of operating system kernels, servers, and other core OS functionality. It is not an OS in *itself* and does not define any particular set of “core” functionality, but merely provides a suite of components from which real OS’s can be built directly on hardware. The OSKit is considered self-sufficient in that it does not use or depend on existing libraries or header libraries installed on the host system.

Building the OSKit is no different than any other user-level system. Installing the OSKit causes a set of libraries to be created in a user-defined location. They can then be linked into operating systems just like ordinary libraries are linked into user-level applications. The most important goal of the OSKit was to be as convenient as possible for the developer to use. This has led to its modular structure. It is also highly separable in that inter-module dependencies are very thin and managed through “glue” layers that provide a level of indirection between a component and the services it requires. The structure of the OSKit is shown in Figure 4.

For usability, it is critical that OSKit components have clean, well–defined interfaces. To provide this sort of abstraction, the Flux project adopted a subset of the Component Object Model as a framework in which to define the OSKit’s component interfaces. COM is a language-independent protocol that allows software components within an address space to rendezvous and interact with one another efficiently, while retaining sufficient separation so that they can be developed and evolved independently. The obvious advantage of COM is that it makes the OSKit interfaces more consistent with one another. The other major technical advantages it brings about are implementation hiding and interface extension. COM helps us to define interfaces independently of the interfaces that implement them. Hence, several different implementations of the same interface can exist together. COM also allows for interface extension and evolution. An object can export any number of COM interfaces, each of which can be defined independently by anyone with no chance of accidental collisions. Given a pointer to any COM interface, the object can be dynamically queried for pointers to its other interfaces. This mechanism allows objects to implement new or extended versions of existing interfaces safely. One of the other abstraction features of COM that is made use of in OSKit is that interfaces can be completely “standalone” and do not require any common infrastructure or support code that the client OS must use in order to make use of the interfaces. This is markedly different from traditional operating systems like BSD or Linux. Consider the networking stacks of these systems. Though they are highly modular, each of their interfaces depend on a particular buffer management abstraction which are `mbufs` and `skbufs` respectively. The OSKit’s corresponding interfaces, on the other hand, rely on no particular common implementation infrastructure.

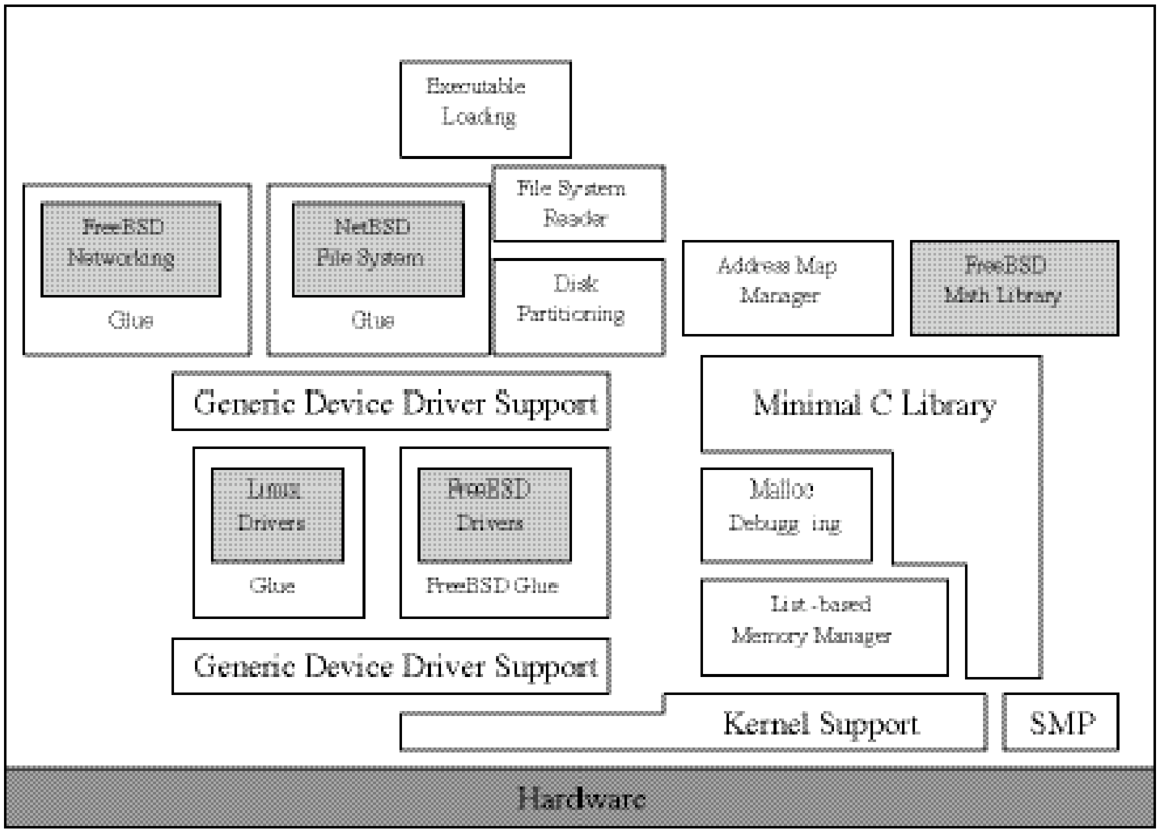


Figure 4 illustrates the OSKit software structure.

2.1.4 The OSKit Structure

Much of the OSKit code is derived directly or indirectly from existing systems such as BSD, Linux, and Mach. The OSKit uses a two pronged approach towards legacy code. For small pieces of code, that aren't expected to change much in the original source base, it is simply absorbed into the OSKit source tree, modifying it as necessary. The OSKit uses an encapsulation method towards large blocks of code. These include code borrowed from existing systems such as device drivers, file systems, and networking protocol stacks. The OSKit defines a set of COM interfaces by which the client OS invokes OSKit services. The OSKit components implement these services in a thin layer of *glue* code, which in turn relies on a much larger mass of encapsulated code, imported directly from the donor OS largely or entirely unmodified. The glue code translates calls on the public OSKit interfaces such as the `bufio` interface into calls to the imported code's internal interfaces. It also in turn translates calls made by the imported code for low-level services such as memory allocation and interrupt management into calls to the OSKit's equivalent public interfaces. Although tricky to implement this simplifies the task of making modifications to each block of code. Modifications to blocks of code remain insulated from other components. For example, the OSKit's Linux device driver set has already tracked the Linux kernel though several versions, starting with Linux

1.3.68. The encapsulation technique described above has made this relatively straightforward.

Most operating systems come with their own boot loading mechanisms, which are largely incompatible with those used by other systems. This diversity is attributed to the fact that not much time is spent on designing boot loaders which are relatively uninteresting when compared to actual OS's itself. The OSKit, on the other hand, subscribes to the Multiboot standard. The Multiboot standard provides a simple but general interface between OS boot loaders and OS kernels. Hence any compliant boot loader will be able to boot any compliant OS. Using the OSKit it is very easy to create kernels that support a variety of existing boot loaders that support the Multiboot standard. Another key feature of the Multiboot standard is its ability of the boot loader to load additional files in the form of boot modules. The boot loader does not interpret the module in any way at the time of loading the kernel. It however provides the kernel with a list of physical addresses and sizes of all the boot modules. It is up to the kernel to interpret these modules in the way it seems fit.

The primary purpose of the OSKit's kernel support library is to provide easy access to the raw hardware facilities without adding obscuring the underlying abstractions. Most of the definitions and symbols defined here are highly specific to supervisor-mode code. This is in contrast to the most other OSKit libraries that are specific to user-mode code. Another difference worth to note is that that most of the code here is architecture specific. No attempt has been made to hide machine-specific details so that the client OS may directly manipulate these. Other OSKit libraries build open these machine-specific details and provide higher architecture-neutral interfaces to higher layers. However these machine-specific details remain directly accessible.

The default behavior of the kernel support library is to do everything necessary to get the processor into a convenient execution environment in which interrupts, traps and other debugging facilities work as expected. The library also locates all associated modules loaded with the kernel and reserves the physical memory in which they are located. The client OS need only provide a main function in the standard C style. After everything is setup, this library calls the client OS with any arguments passed by the boot loader.

Memory management implementation typically used in user space, such as the malloc implementation in a standard C library, is not suitable for kernels because of the special requirements of the hardware on which they run. For example, device drivers need to allocate memory with specific alignment properties and space constraints. To address these issues the OSKit includes a pair of simple but flexible memory management libraries. The list-based memory manager, or LMM, provides powerful and efficient primitives for managing allocation of either physical or virtual memory, in kernel or user-level code, and includes support for managing multiple "types" of memory in a pool, and for allocation with various type, size and alignment constraints. The address map manager, or AMM, is designed to manage address spaces that don't necessarily map directly to physical or virtual memory. It provides support for other aspects of OS implementation such as the management of process address spaces, paging partitions or free block maps. Although these libraries can be used in user space, they are specifically designed to satisfy the needs of OS kernels.

The OSKit provides two different C libraries – one a minimal C library native to OSKit and another imported from the FreeBSD C library. The OSKit's minimal C library is designed around the principal of minimizing dependencies rather than maximizing functionality and performance. For example the standard I/O calls don't do any buffering, instead relying directly on underlying read and write operations. Dependencies between C library functions are minimized. This approach is followed since the standard C library running on a full-function OS, such as Linux, makes too many assumptions to be reliable in a kernel environment.

The FreeBSD C library provides an alternative the OSKit's minimal C library so that sophisticated applications can be built using it. In addition to the standard single threaded version of the library, a multi-threaded version is also built which relies on the pthread library to provide the necessary locking primitives. Like the minimal C library, the FreeBSD C library depends on the POSIX library to provide mappings to the appropriate COM interfaces. For example, fopen in the C library will chain to open in the POSIX library, which in turn will chain to the appropriate oskit_dir and oskit_file COM operations.

The OSKit provides the developer with a full source-level kernel-debugging environment. The OSKit's kernel support library includes a serial-line stub for the GNU debugger, GDB. The stub is a small module that handles traps in the client OS environment and communicates over a serial line with GDB running on another machine, using GDB's standard remote debugging protocol.

One of the most expensive tasks in OS development and maintenance is supporting the wide variety of available I/O hardware. The OSKit avoids direct maintenance by leveraging the extensive set of stable, well-tested drivers developed for existing kernels such as Linux and BSD. The OSKit uses the technique of encapsulation discussed earlier to integrate these various code bases into the OSKit. Currently, most of the Ethernet, SCSI and IDE disk device drivers from Linux 2.2.14 are included. Eight character device drivers which manage the standard PC console and the serial port are imported from FreeBSD in the same way.

The OSKit provides a full TCP/IP network protocol stack. It incorporates the networking code by encapsulation. The TCP/IP stack is borrowed from the FreeBSD system which is generally considered to have much more mature network protocols. This demonstrates another advantage of using encapsulation. Two different systems, namely Linux device drivers and FreeBSD TCP/IP can coexist with one another. With this approach, it is possible to pick the best components from different sources and use them together.

However this approach is also fraught sometimes with inefficiency. The networking stack is an excellent example of this. When a packet arrives at an OSKit node, it is initially picked by the Linux device drives and represented as the Linux packet buffer, skbuff. The OSKit represents all packets as COM bufio objects. Due to contiguous nature of Linux packet buffers, they can be directly passed to the FreeBSD TCP/IP stack as bufio objects. The FreeBSD code internally repackages them as mbufs, which is the FreeBSD abstraction for packet buffers. However, the situation reverses in the case of outgoing packets. mbufs consist of multiple dis-contiguous buffers chained together. Hence when they are passed to the Linux driver code as bufio objects, the Linux code has to resolve these into contiguous buffers. This mismatch sometimes requires extra copying between on the send path.

The OSKit incorporates standard disk-based file system code, again using encapsulation, this time based on NetBSD's file systems. The choice of NetBSD was influenced by the fact that it had one of the best separated interfaces of the available systems. FreeBSD and Linux file systems are more tightly coupled with their virtual memory systems.

2.1.5 Active Networking NodeOS

The NodeOS interface defines the boundary between the EE and the NodeOS. Generally speaking the NodeOS is responsible for multiplexing the node's resources among the various packet flows, while the EE's role is to offer AAs a sufficiently high-level programming environment. The design of the NodeOS is influenced by three major considerations:

1. The interface's primary role is to support packet forwarding. Hence the interface is designed around network packet flows: packet processing, accounting for resource usage, and admission control are all done on a per-flow basis. No single definition is attributed to the flow.
2. All NodeOS implementations need not export the same set of interfaces. Some NodeOS implementations can have advanced features such as hardware transfer of non-active IP packets. However, these features must be exported to EEs so that they may make use of it. The NodeOS may also be extensible. Exactly how a particular OS is extended is an OS-specific issue.
3. Whenever the NodeOS requires a mechanism that is not particularly unique to active networks, the NodeOS interface borrows from established interfaces, such as POSIX.

The NodeOS defines five primary abstractions: thread pools, memory pools, channels, files and domains. The first four encapsulate a system's four types of resources: computation, memory, communication, and persistent storage. The domain abstraction encapsulates all the above and is used to aggregate control and scheduling of the other four abstractions.

The domain, Figure 5, is the primary abstraction for accounting, admission control, and scheduling in the system. A domain typically contains the following resources: a set of channels on which messages are received and sent, a memory pool, and a thread pool. Active packets arrive on an input channel (inChan), are processed by the EE using threads and memory allocated to the domain, and are then transmitted on an output channel. One can think of a domain as encapsulating resources used across both the NodeOS and an EE on behalf of a packet flow. Domain creation is hierarchical, see Figure 6. This is used solely to constrain domain termination. A domain can be terminated by the domain itself, or by the NodeOS because of some policy violation. The domain hierarchy is independent of resource allocation. That is, each domain is allocated resources according to credentials presented to the NodeOS at domain creation. These resources are not deducted from the parent domain.

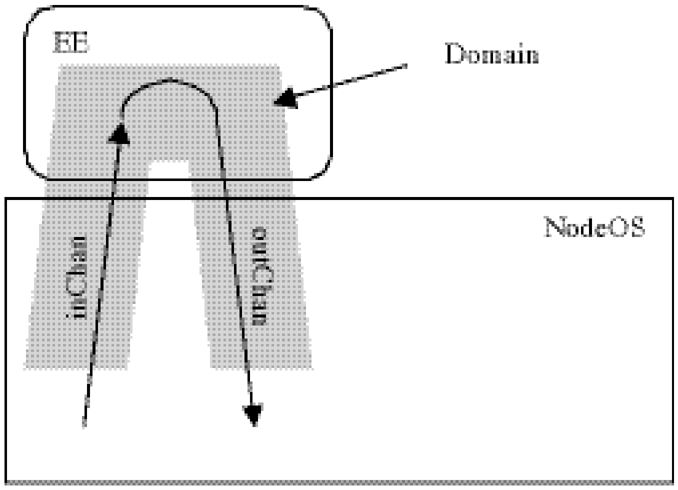


Figure 5 illustrates a NodeOS Domain

Thread pools are the primary abstraction for computation and exist for accounting purposes. A thread pool is initialized at the time of domain creation and threads run "end-to-end", i.e., to forward a packet they typically execute input channel code, EE-specific code and output channel code. Threads in the pool are implicitly activated and scheduled to run in response to certain events like message arrival, timers firing and kernel exceptions. The entire domain is terminated if a thread misbehaves. There is no explicit operation of killing a thread.

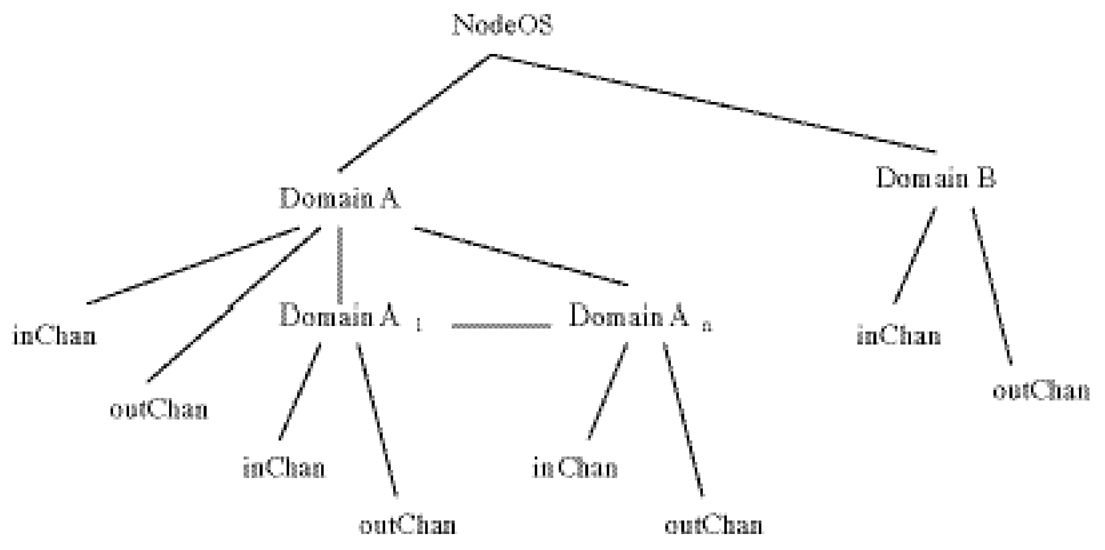


Figure 6 a NodeOS Domain Hierarchy

The memory pools are the primary abstraction for memory. It is used to implement packet buffers, the NodeOS abstraction for network flow buffers. A memory pool combines the memory associated with the domains. This many-to-one mapping between domains and memory pools accommodates EEs who would like to manage memory resources themselves. Memory pools have an associated callback function that is invoked by the NodeOS whenever the resource limits of the pool have been exceeded. The corresponding domains are terminated by the NodeOS if the EEs do not handle resource violation in a timely manner. Memory pools can be arranged hierarchically though this is not used to control the propagation of resources.

Domains create channels to send, receive and forward packets. Some channels are anchored in an EE i.e. they are used to send packets from an EE to the underlying physical layer and vice versa. They can hence be classified into two types-inChan and outChan. When creating an inChan, a domain must specify the following things: (1) which arriving packets are to be delivers on this channel; (2) a buffer pool that queues packets waiting to be processed by the channel; and (3) a function to handle the packets. Packets to be delivered are described by a protocol specification string, an address specification string, and a de-multiplexing (demux) key. On the other hand the requirements for an outChan include (1) where the packets are to be delivered and (2) how much link bandwidth the channel is allowed to consume. Another type of channel known as cutChan is defined which forward packets through the active node without being processed by an EE. This might correspond to a standard forwarding path that the NodeOS implements very efficiently. It can also be created by concatenating an existing inChan to an existing outChan.

A packet is de-multiplexed by specifying the protocol and addressing information. For example, the protocol specification “if0/ip/udp” specifies incoming UDP packets tunneled through IP. The address specification defines destination addressing information like the destination UDP port number. The NodeOS designers realized that simply specifying the protocol and addressing information is insufficient when an EE wants to de-multiplex multiple packet flows out of a single protocol. Hence a demux key is used which is passed on to the inChan. It can specify a set of (offset, length, value, mask) 4-tuples. These tuples are compared in an obvious way to the “payload” of the protocol.

Files are provided to support persistent storage and sharing of data. The file system interface loosely follows the POSIX specification and is intended to provide a hierarchical, namespace to EEs that wish to store data in files.

2.1.6 The Moab NodeOS

The Moab is a C implementation of NodeOS developed at the University of Utah. It comprises of a multi-threaded fully pre-emptable single address-space operating environment implementing the NodeOS abstractions. Moab is not an operating system in the strict sense of the word. This is because invocations of NodeOS functions are direct function calls and do not “trap” into the OS.

Moab is built using the OSKit. This helps it to leverage many of the components such as the device drivers, a networking stack, and a thread implementation, as well as a host of support code for booting and memory management. The following paragraphs describe the Moab implementations of three NodeOS that help us to understand the advantages and disadvantages of using OSKit:

The implementation of NodeOS threads directly leverages the POSIX thread library. This was possible because of the similarity between the NodeOS and POSIX APIs. This direct mapping between NodeOS and POSIX threads caused some performance problems. The NodeOS’ thread-per-packet model of execution led to creation and destruction of pthreads, which imposed a lot of overhead. This was avoided by creating and maintaining a cache of active pthreads in every thread pool.

The OSKit made it easy to track the memory allocated and freed within its components such as the networking stack. However, it was difficult identifying the correct user to charge the memory. Right now, the Moab charges the memory allocated to the “root flow”. The alternative would be to charge it to the thread doing the allocation. The OSKit’s memory interfaces are being modified to bring it inline with the NodeOS’ needs.

Channels provide the path necessary for execution of a packet flow. Anchored channels, namely inChan and outChan, are implemented in two ways depending on their protocol specification. Raw interface (“if”) channels are directly implemented over the stock Linux device drivers. All the other types of protocols use the OSKit's socket interface and its network stack to deliver UDP/TCP packets to Moab. This only partially implements the NodeOS API requirements since direct "IP" packet delivery is not supported. Cut-through channels are implemented as un-optimized concatenation of NodeOS inChan/outChan pairs and perform no additional protocol processing.

2.1.7 Implementation of PLAN Router on OSKit

The OSKit provided us with a platform wherein single address-space based kernels could be directly executed on bare hardware. It is with this idea that we proceed to build a PLAN router that operates directly over the OSKit. The OSKit provided all the essentials required for the router to operate. These include a C library, a TCP/IP protocol stack and a pthread library. The OSKit's modular design helps us in choosing the components we would like the router to be integrated with. This modularity also implies configurability. Hence, given a choice, we could link the PLAN router with a different set of components at system build time.

The PLAN router is built in a vertically integrated fashion over the OSKit in a single address space. In a traditional operating system, such as Linux, we tend to differentiate between the user space and the kernel space. This is done to operate the two levels with different levels of trust. This again is due to the fact that programs loaded and unloaded at runtime should be operated only with certain privileges. When a single application such as a PLAN router is intended to run on a system, it is no longer necessary to differentiate between processes. Hence, the distinction between kernel-space and application becomes fuzzy. In our implementation we will study the effect of OSKit's components on our PLAN router implementation.

2.1.7.1 Porting OCaml to OSKit

The first task involved in the implementation of a PLAN active node on OSKit was the porting of the OCaml language to OSKit. A normal install of OCaml on Linux would look like in Figure 7.

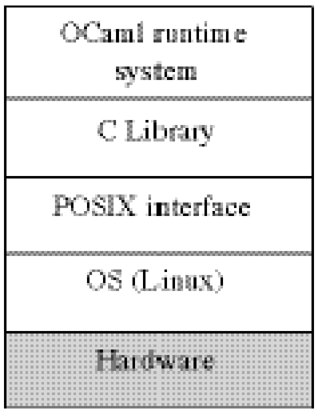


Figure 7 show an OCaml installation on Linux.

In order to integrate OCaml with OSKit, the interfaces required by OCaml from the underlying operating system have to be figured out. It is also necessary to examine the interfaces provided by the OSKit in order to match them with those required by OCaml. In this process, the dependencies of the libraries linked to provide the interfaces have to be taken care of. The sequence of development of a custom OCaml kernel over the OSKit is shown in Figure 8.

A regular OCaml system is installed over Linux, in order to build the various OCaml tools such as ocamlc (OCaml bytecode compiler), ocamlopt (OCaml native code compiler), etc. The OCaml-on-Linux compiler (`ocamlc` or `ocamlopt`) thus built is used to custom-compile OCaml sources to generate a C object file instead of compiled object bytecode executables. The objective Caml runtime system comprises three main parts: the bytecode interpreter, the memory manager, and a set of C functions that implement the primitive operations. In the default mode, the Caml linker produces bytecode executables for the standard runtime system, `ocamlrun`, with the standard set of primitives. In the "custom runtime" mode, we may generate a C object file that contains the list of C primitives required or an executable file that contains both the runtime system and the bytecode for the program.

The OCaml runtime system was built using OSKit libraries. We then use the C object file generated above and link it against a runtime system built using OSKit components. If we are using the native code compiler, the `-custom` flag is not necessary, as it can directly produce a C object file with the help of the `-output-obj` option.

An OSKit interface file is then compiled which initializes the OCaml component. This is linked with the OSKit libraries and the modified OCaml runtime system to form an executable. Using the OSKit's image generating tools, we generate a multi-boot image of this kernel. This image is then booted as a kernel using a multi-boot loader such as GRUB (Grand Unified Boot-loader).

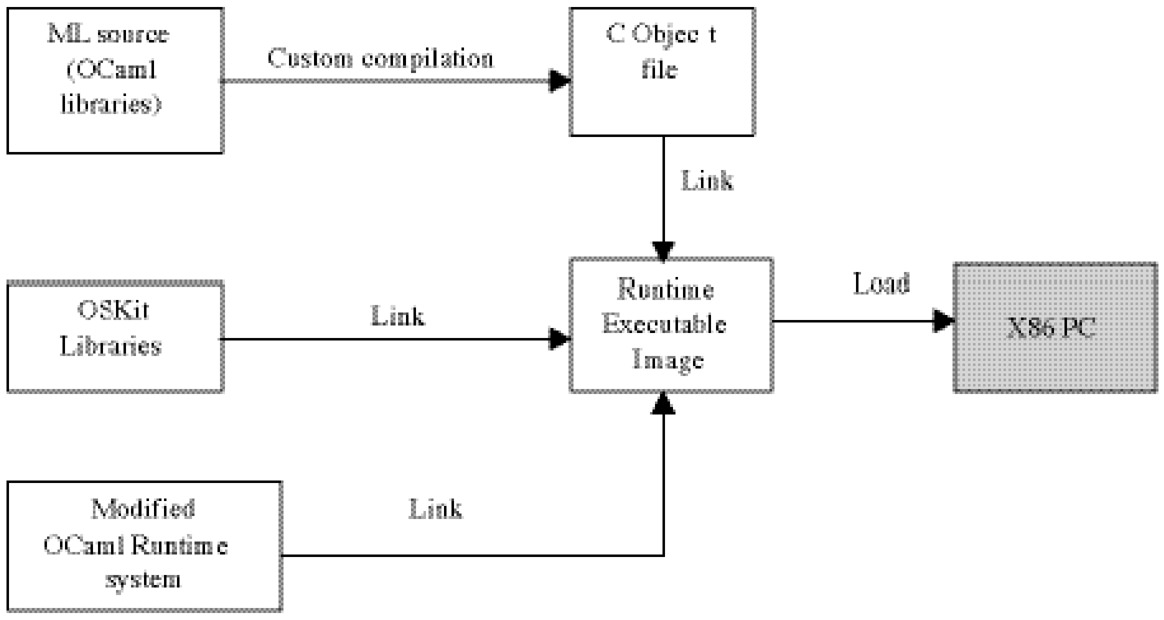


Figure 8 shows the Ocaml on OSKit software build process.

2.1.7.2 Integration of PLAN with OSKit

The integration of PLAN with OSKit involves matching all the OCaml interfaces used by PLAN with those provided by OSKit. Fixes were required in areas such as file operations and usage of loopback interfaces. To fix this, it was decided to convert all file-based operations to string operations. Most of these files were configuration files and DNS files. PLAN operates its own DNS system which makes use of a file similar to the standard '/etc/hosts' file on Unix systems. The PLAN code thus modified is then custom compiled to generate object code. This is then linked against OSKit libraries to form an executable that in turn is used to generate a multiboot image.

The PLAN router protocol graph as used on OSKit is shown in Figure 9.

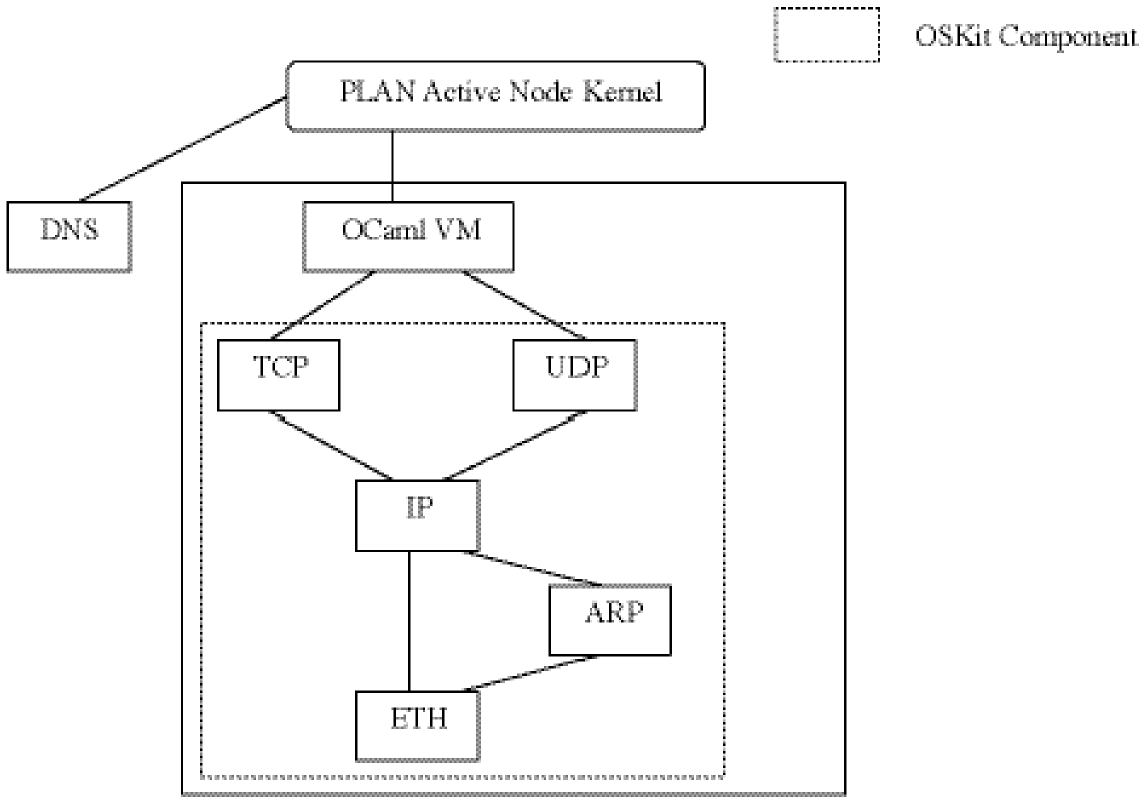


Figure 9 illustrates the PLAN protocol stack.

The DNS shown in Figure 9 is implemented by PLAN in a user-level file that maps host names with their IP addresses. The OCaml VM is the OCaml runtime system linked against OSKit libraries that can load and interpret bytecode modules. The TCP, UDP and IP modules of OSKit are derived from the FreeBSD TCP/IP stack of OSKit. The Ethernet module uses the device drivers of Linux to probe and operate network cards. Such a system is not highly optimized for network operations. We shall further investigate this during our discussion of our results. Minor fixes were required to the Pthread library to ensure that it works correctly.

2.1.7.3 Integration with the Moab NodeOS

As described earlier, the Moab NodeOS is a C implementation of the NodeOS API. The Moab environment is first setup by means of the interface an_moab_setup. This function initializes the Moab by first initializing the memory associated with the NodeOS flow. The various hardware devices are probed and file systems initiated. The networking framework is initialized and can be configured manually. The Moab now reaches a state where its bare setup is done. The NodeOS root flow is now started in this context. The parameters passed to it include the root flow thread count and the root flow stack size. The function for flow initialization and it's associated arguments is also passed. The various NodeOS components such as resources, credential objects and threads are initialized. The complete setup of Moab is now complete.

With the initialization of Moab complete, the root flow is then fired off as a thread in this context. There is no callback associated with the root flow. It can be killed only by an explicit call of ani_moab_shutdown. New credentials are created in the root flow that may correspond to child flows. Since we have only one flow subordinate to the root flow, all the root credentials are transferred to the new flow. A new flow is now created in the context of root flow. The flow initialization function and the corresponding arguments are now passed to this flow. This flow is used to startup the OCaml virtual machine. A resource specification is also passed to the flow that specifies the number of threads and the stack size allocated for each thread. A thread is then fired off which starts up the Caml flow.

2.1.7.4 Testing and Discussion of Results

The following test framework was used to evaluate our implementation. Figure 10 describes a linear topology used to test the performance of OSKit as a node operating system for an OCaml based active node.

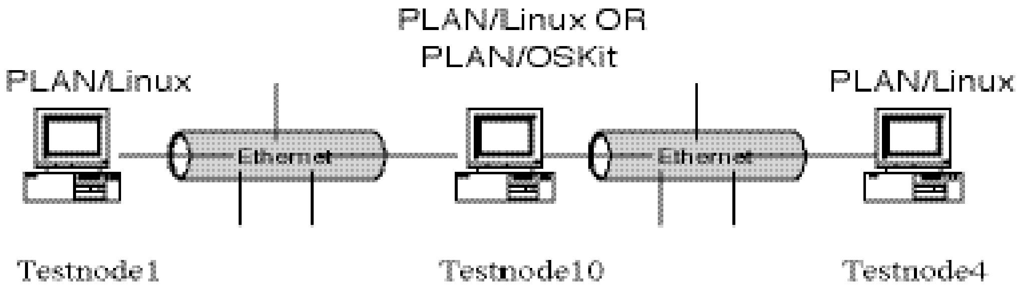


Figure 10 show the basic setup for testing.

The machines used in the above tests were equipped with 530 MHz Pentium III processors and 128 MB RAM. Their network interfaces were connected to 100 Mbps Tulip cards. The Caml benchmarking tests were done on a single machine featuring both Linux and OSKit versions. The table compares Linux 2.2.13 and OSKit version 20010214. The tests have been performed both with the bytecode and native code versions of OCaml. All measurements are made using the `rdtsr` function provided for *i386* machines.

Most of the benchmarks are self-explanatory. We observe that the speedup provided by OSKit as compared to Linux is not heavy.

Benchmark Test	**OCaml-bc/ Linux (ms)**	**OCaml-nat/ Linux (ms)**	**OCaml-bc/ OSKit (ms)**	**OCaml-Nat/ OSKit (ms)**
Array Access 1000000 times in a tight loop	876.25	215.992	851.441	199.035
Array Access 2 1000000 times after unrolling of loops	757.45	215.891	742.894	199.167
Fibonacci Series N = 32	1,159.94	140.16	1,151.54	149.053
Hash Access With 80000 entries	1,320.33	692.093	1,084.01	518.831
Heap Sort of 80000 randomly created entries	1,876.98	138.048	1,914.63	135.223
Various List operations for 16 lists each of size 10000	991.441	184.252	990.477	171.724
Matrix Multiplication of 2 matrices of size 30x30	5,645.93	198.179	5,824.22	197.17
1000000 Method Calls on the same object	1,429.35	129.09	1,388.04	129.192
Loop overhead for 16 nested loops	3,861.07	169.358	3,668.71	170.669
Thread synchronizations between 2 producer/consumer threads using a mutex and a condition variable	1,804.36	1,740.73	1,186.81	1,132.77
Generated 900000 Random number	1,085.74	164.336	1,089.89	164.168
Sieve of Eratosthenes Counts primes from 2 to 8192 , 300 times	3,202.76	145.933	3,170.74	155.087
String concatenates 40000 strings(using `Buffer.add_string`)	84.231	11.286	83.338	11.173
String Append (using the string concatenation operator)	44,959.89	43,684.61	41,592.80	41,274.74

Table 2 compares the performance of four implementations of OCaml on Linux and the OSKit

2.1.7.4.1 Baseline Performance Evaluation

We study the performance of our implementation of the PLAN active network node on OSKit using the above test framework and compare it with PLAN running over Linux. Specifically, we measure the latencies observed during the execution of the PLAN ping program.

The PLAN ping experiment will serve to demonstrate the difference in latencies between Linux and OSKit. It consists of a simple application program, which injects a PLAN packet that contains the ping code described earlier. This packet is injected in Testnode1. The packet is evaluated at its destination (Testnode11), that is across a single hop and sent back to the source.

2.1.7.4.2 Results

For the above PLAN ping experiment, the results are as shown in Figure 11. The PLAN ping times have been measured as the average round trip time taken by the packets sent from a Linux host to an OSKit router. Each individual test involved 100 round trip times. It illustrates the performance of this active protocol both on Linux and OSKit. To enable us to compare these figures with a standard benchmarks, the ICMP and C-level ping times are also shown. The C level ping consisted of a simple UDP client which sends a packet to a UDP server that returns the packet back to the client. It is also noticed that there is perceptible difference between the bytecode and native version of PLAN/OSKit. The Linux version does not show such a large difference between the two versions.

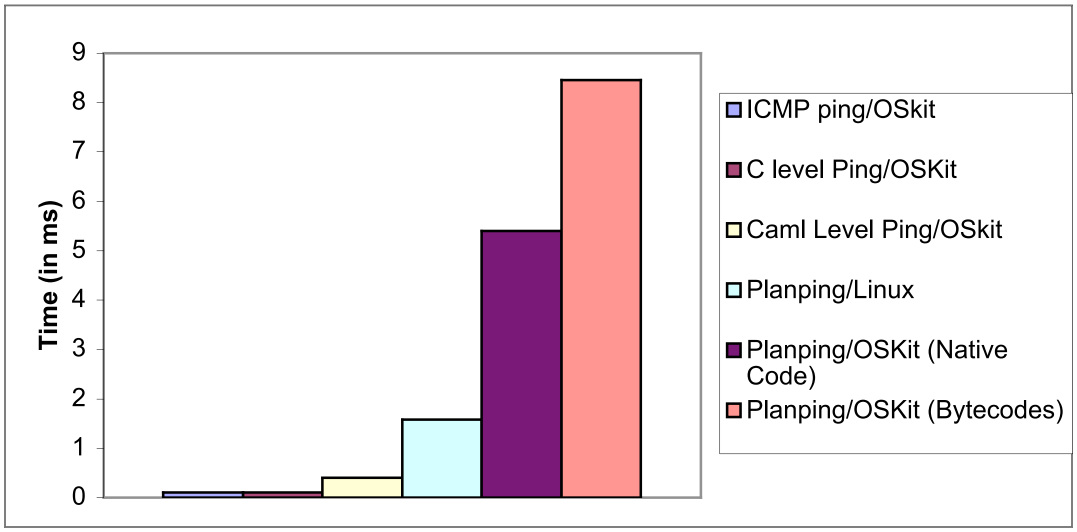


Figure 11 Latency measurements between endnodes using test setup in Figure 10

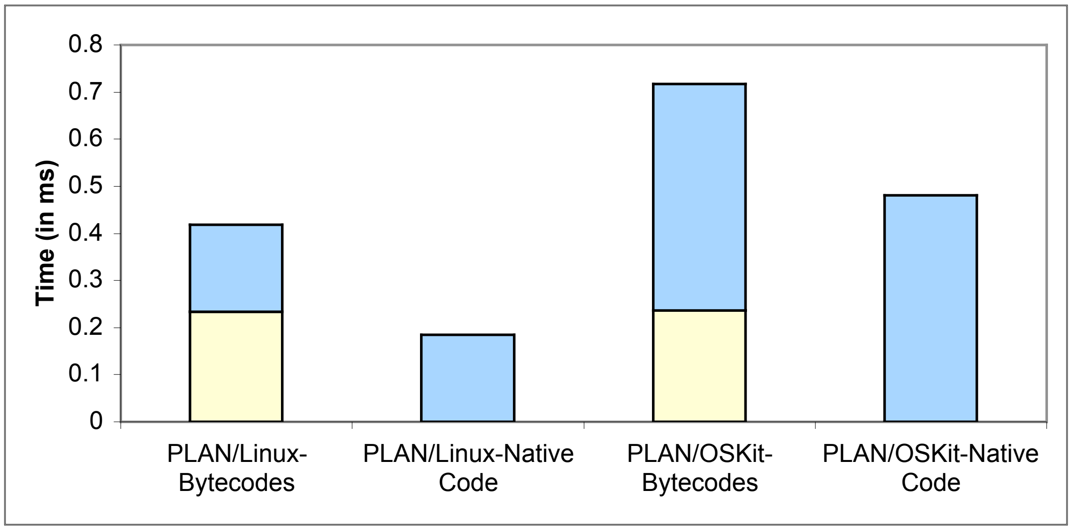


Figure 12 PLAN ping packet evaluation overhead.

The time taken for packet evaluation is shown in Figure 12. The packet concerned contains the PLAN ping packet code. The above chart measures the time taken by a PLAN Active node to un-marshall the packet, interpret the PLAN code and send it over to is next destination which, in our case, is the source. The processing overhead is considerably larger for PLAN/OSKit when compared to PLAN/Linux. However this account for only a little of the total delay.

In order to further understand the delays seen, we ran an un-optimized version of PLAN/OSKit. This version forks an extra thread that waits for packets from the local PLAN port. It was noticed that the delays experienced increased considerably in this case. These results are tabulated in Table 3. This is probably due to poor performance of the OSKit Pthread library.

Switch	**Optimized PLAN**	**Un-optimized PLAN**
PLAN/OSKit – Native Code	5.4 ms	14.3 ms
PLAN/OSKit – Bytecodes	8.45 ms	37.91 ms

Table 3 lists the execution times for two implementations of PLAN and two runtime environments.

2.1.7.4.3 Throughput Tests

The following test framework measures the forwarding capacity of a PLAN node. We make use of a linear back-to-back connectivity between 3 nodes in order to compare the forwarding performance of a PLAN switch. The test topology is shown in Figure 13.

In order to do these tests, PLAN routers are set up on the three nodes. We then make use of a Caml program that generates load of a given payload on the sender side. This program injects these packets into the local PLAN router through its PLAN port. The destination of these packets is on the far end of the topology, which is Testnode 4. These packets are initially evaluated at Testnode1 and routed through the active network by Testnode11. The PLAN router on Testnode 4 later receives them and hands over the packets to a receiver program running on Testnode 4 which waits for these packets on its PLAN port. A total of 80,000 packets were sent from Testnode1 through Testnode11 onto Testnode4.

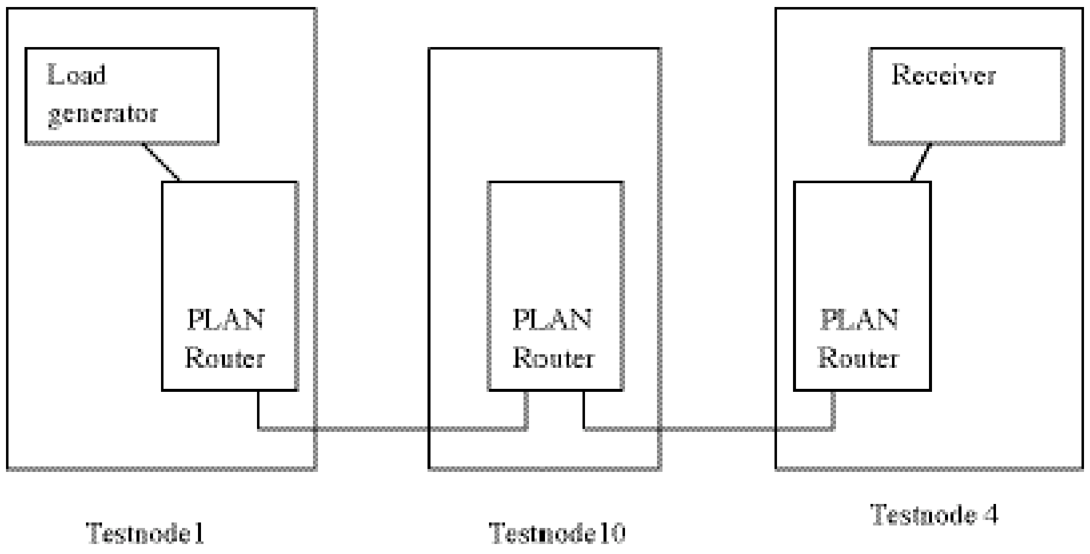


Figure 13 Illustrates the setup for router-level throughput tests.

The results of these tests are as shown in Figure 14. As seen from the figure the PLAN/OSKit shows significant loss of packets as compared to the PLAN/Linux switch. This could be attributed to a less tighter integration of PLAN over the OSKit framework.

The above tests do not entirely isolate our problem of interest. The PLAN router communicates with a local application by means of PLAN ports that are implemented using Unix domain sockets. In addition to the forwarding overhead of the switch used on Testnode11, these tests also carry the overhead of transfer of packets between PLAN routers and the active applications, namely the load generator and the receiver. The tests shown in Figure 14 reflect the performance achieved by such application.

2.1.7.4.4 Router-level Exchange

The tests described here measure the performance achieved by a PLAN networking system. It makes use of an inbuilt “bandwidth service” provided with the PLAN router. Hence packets are dispatched directly to and from the PLAN routers, avoiding the costs of copying to and from the application. The test framework is shown in Figure 13.

The sender service makes use of a script provided with the PLAN code to initiate the service. The sender is made to send 100000 packets with a delay of 100 busywaiting iterations of an empty loop between each send. The sender initially sends out a connection to the destination (testnode4) that starts a receiver thread there. The receiver counts the number of packets arrived and reports the bandwidth depending on the payload of the packet. The results of the test are shown in Figure 14.

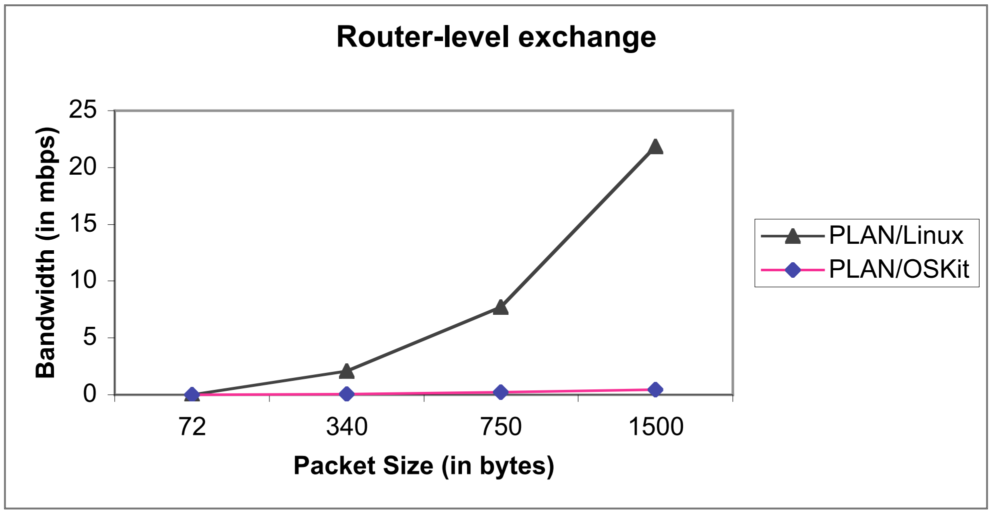


Figure 14 Comparison of routing performance of PLAN

The performance of a PLAN router is significantly affected. In the PLAN/Linux case, for an Ethernet packet size of 750 bytes, only 9.4 % of the packets made it to the receiver. The figures for the PLAN/OSKit case were worse. Of the 100000 packets dispatched from Testnode1, the receiver reported only around 370 packets. The above tests involving PLAN were conducted using a version that makes use of queues between its network layer and link layer. In order to reduce the synchronization overhead associated with queue, the router-level exchange tests were repeated with a new configuration.

The topology for the tests remains the same, as shown in Figure 13. The new configuration of PLAN uses direct upcalls between its network layer and link layer. This helps in removing a lot of thread synchronization overhead. The following results, shown in **Error! Reference source not found.**, verify this position.

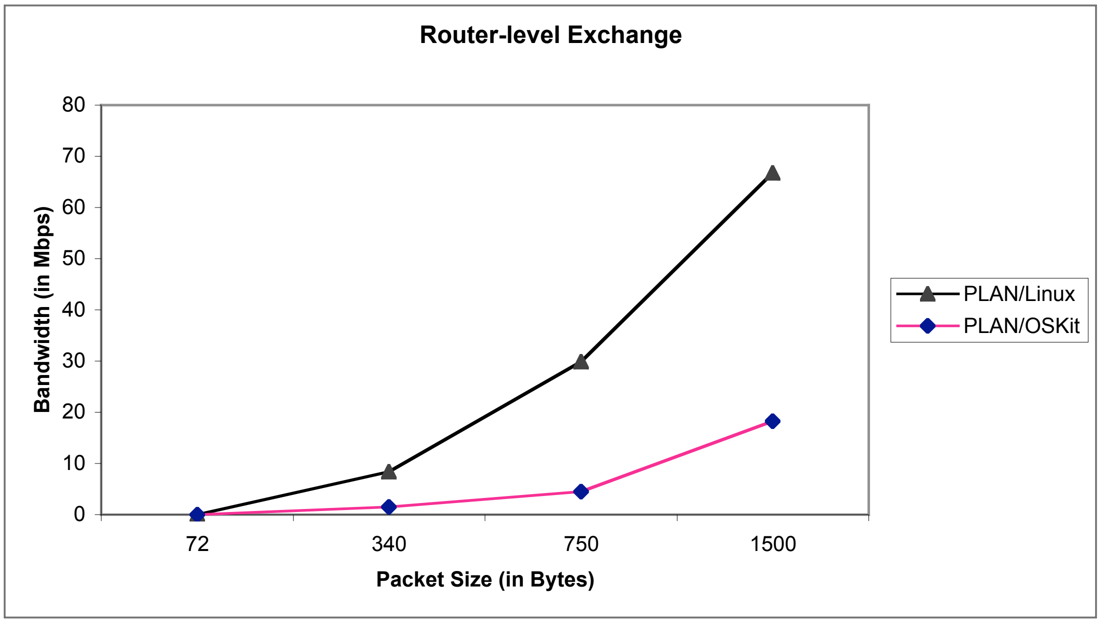


Figure 15 Comparison of routing performance of PLAN using upcalls.

Error! Reference source not found. illustrates the performance advantage gained by operating PLAN using upcalls. The numbers of packets that make it to the receiver also register a significant increase in both the cases. In the case of PLAN/Linux, the number of packets received at Testnode4 increases by 4 times. The improvement in performance is more marked in the PLAN/OSKit. As many as 5,600 packets make it to the receiver which is a 14-fold increase over earlier numbers.

While we were able to assemble an active networking node from these components, the complexity and performance did not meet our goals. The OS Toolkit used Ethernet drivers and a standard protocol stack from the Linux distribution. This reduced the work needed to implement network drivers and internet protocols. However, the "glue" programs to link the Linux drivers to the OS Toolkit APIs reduced the performance of active networking services. Further, the OS Toolkit did not implement all the operating services required by the OCaml runtime system. The details of this work are reported in ITTC-FY2003-TR-19740-07.

2.2 Distributed Security Policies

We also developed a mechanism to distribute security policies through an active network. The details are reported in ITTC-FY2003-19740-08 and below.

We explain our design by first giving an overview of our framework. Each module within this framework is described and specifics of its implementation are then detailed.

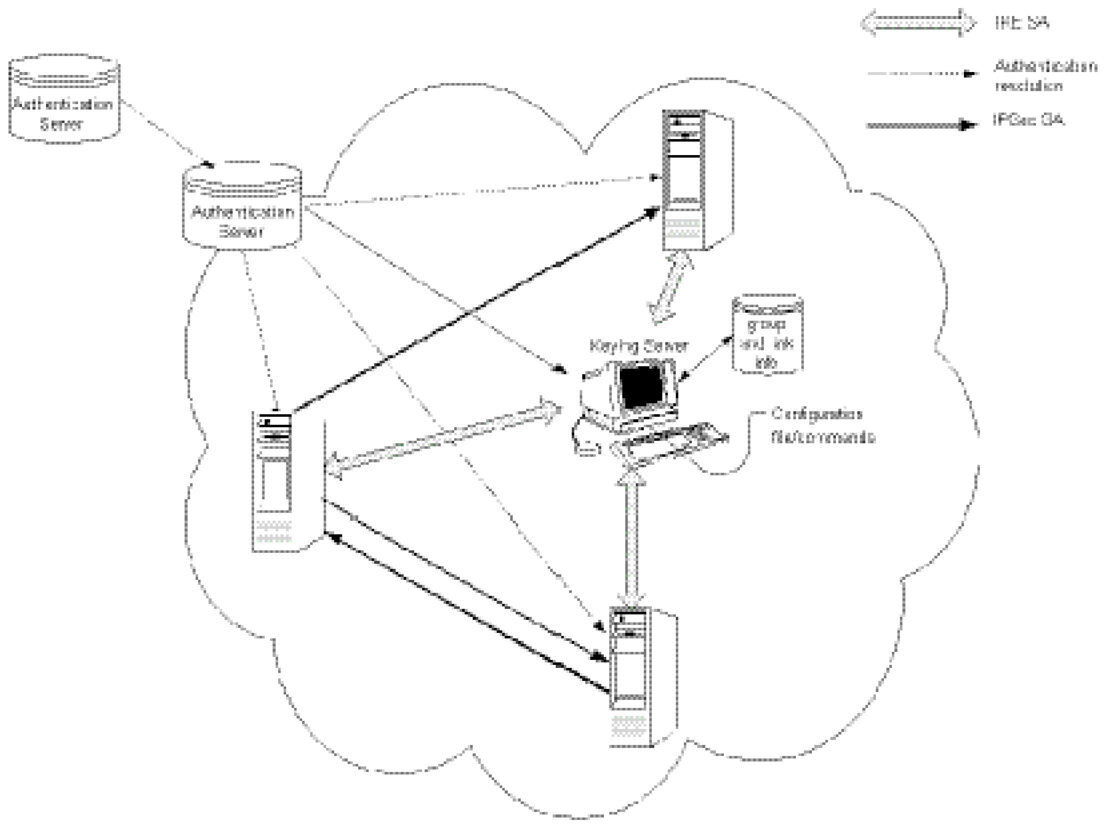


Figure 16 illustrates the design of the keying framework.

Our keying infrastructure consists of three main components - the Keying Server (KSV), a Key Management Module (KMM) within every node, and an Authentication Server (ASV). The keying server forms the central entity in our framework.

The secure topology is specified in the form of links and groups, where the links correspond to unidirectional Security Associations and the groups, the different multicast groups. This topology is configured into the KSV either statically, when the server comes up or dynamically using configuration commands. We define the “trusted set” in a KSV as the collection of all those nodes that are either configured as one end of a secure link or a member of a secure group inside a particular KSV.

The KMM runs on every node that wants to be part of the KSV’s trusted set including the KSV itself. In order to set up the relevant security associations, a node needs to register itself with the KSV. Either the KSV or the trusted node can initiate this registering process. If initiated by the node, we can think of this being done automatically when the node boots up. A node can register with multiple servers; we do not however define how to resolve any conflicts that may arise out of the same link being defined at two different servers. Errors in configuration are indicated to the appropriate servers when a node detects that such a mismatch in configuration has occurred. Policy can additionally be used to define which KSV a node is allowed to register with in the first place. We however suggest this last area as future work.

During the node registration process and any time the configuration at the KSV affecting this particular node has changed, the KMM at the KSV sends all the pertinent information for both links as well as groups to this node. This link and group information is then used to set up the security associations within the node.

Information exchanged between the KSV and the trusted nodes are authenticated so as to discourage spoofing of messages, sent either with the intent of breaking security or simply in order to perform some Denial of Service attack at the KSV. Since data between the KSV and the trusted node is sent on a more-than-rarely basis, we do not use direct public key mechanisms as summarized in Section 2.2. Instead we use a shared key approach. We however require some infrastructure in order to authenticate the shared key created. The Authentication Server provides this service in our framework. The ASV maintains authentication information for all those nodes that it considers as being a part of its domain of control. We define "domain of control" for a particular Authentication Server as all those nodes for which it directly maintains authentication information. ASVs are organized hierarchically and perform some trust-chaining mechanism to authenticate nodes not lying directly in its domain of control.

We define our security services at the network layer using IPSec as our reference framework. Rather than defining a separate module for the KMM we decide to extend the services provided by IKE itself so as to seamlessly integrate IPSec services within our framework.

Finally, since we authenticate different entities from a network-layer perspective, DNSSEC comes up as a good candidate for providing these authentication services. The ASV can now be treated simply as a DNSSEC server with the aforementioned trust-chaining mechanism being natively provided by it. DNSSEC-aware resolvers can check authenticity of retrieved DNS records by simply verifying the SIG records sent as part of the query responses.

In the following sections we describe the basic design of the various sub-components, the capabilities if any, that we need additionally from them, and finally how we actually provide these capabilities.

2.2.1 Integrating IKE

We use the "pluto" implementation from Freeswan [Freeswan] as our IKE module. Pluto runs as a daemon on a Linux Network node. This base implementation though incomplete with respect to some features of IKE is still sufficient in order to inter-operate with other pluto implementations and many other IKE implementations. Commands to pluto are given using a control interface to the daemon, called “whack”.

Pluto uses either shared secrets or RSA signatures to authenticate peers during the IKE phase-1 exchange. This database of pre-shared secret keys and private keys required during the authentication process is maintained in the file /etc/ipsec.secrets. Public keys for different hosts are normally specified using whack commands but can also be queried-for using DNS. In order to use the IKE, pluto needs to be started using administrator privileges. Once the daemon has been fired up, it enters a quiescent state and awaits further instructions from whack. Pluto may optionally be started using a configuration file in which case it immediately initiates setting up of the different security associations specified therein.

Internally, pluto maintains the following data structures – a connection list and a state hash table. A single connection maintains all such information necessary in order to create an SA having these characteristics. The connection structure typically maintains information such as the two SA end-points, the lifetime of the SA, a reference to the physical interface, and information linking it to the SA actually instantiated.

During the IKE exchange, the information about the negotiation is represented in a state object. Each state object reflects how far the negotiation has progressed. Once the negotiation is complete and the SA established, the state object persists to represent the SA. When the SA terminates, the state object is also discarded. Each state object is given a serial number and this is used to refer to it in logged messages. At any given time there might be many different states corresponding to the same connection. Once the SA for a connection has been created, generally two types of states exist – one ISAKMP state and one or more IPSec states. Pluto destroys all its states once it shuts down.

Every state object is also associated with an event. An event fires when one of the following occurs:

The SA corresponding to this state has expired

The SA corresponding to this state has to be replaced

The data packet associated with this state has to be retransmitted

Some secret needs to be refreshed

The state object has to be discarded

Events are maintained as simple entries in a queue and fire whenever the timers associated with them expire.

A connection (and consequently its state) also maintains information about the policy for this SA. Policy is simply a collection of certain flag values that denote what kind of SA is to be created. Thus we have values such as POLICY_AUTHENTICATE for authentication, POLICY_ENCRYPT for encryption, POLICY_TUNNEL for tunnel mode, etc. The state transitions after an IKE SA depends on what flags are set in this policy. For example, IPSec SAs are created only if either the POLICY_AUTHENTICATE or the POLICY_ENCRYPT flags are set.

2.2.2 Interaction between the node and the Keying Server

Once an IKE SA between the Keying Server and the node has been set up, keying information needs to be sent from the former to the latter. We define a new policy POLICY_KEYSERVER to indicate that IKE SA creation for this connection needs to be followed by a keying update.

Since the registering node needs to set up IPSec-SAs without any direct IKE negotiation with the peer, the KSV needs to send all such information required by the registering node in order to describe the connection and SA between the latter and its peer accurately. We investigate the different fields that need to be included in the message sent by the KSV in order to achieve this.

In order to set up a Security Association, the first thing that needs to be known is the type of security service (transformation type) – AH or ESP being offered by this SA. We specify a list of policy groups in the KSV configuration file where each policy group contains the following information:

The transformation type : e.g. (TRANS-TYPE = AH) or (TRANS-TYPE = ESP)

The lifetime in seconds for this SA: e.g. LIFE-SEC = 100

The lifetime in Kbytes of information transformed by this SA: e.g. LIFE-KBYTE = 10000

Every link or group specified in the configuration file is associated with a particular policy. In case no policy is specified, a default value of (TRANS-TYPE = AH, LIFE_SEC = 28800, LIFE-KBYTE = 86400) is assumed.

Another value that we need to add artificially is the SPI value. The SPI as mentioned before is used to identify a particular SA at the receiving end. Since this value is receiver specific, the KSV creates an SPI for every node that it maintains as part of its trusted set. By ensuring that the SPI generation process does not create duplicate values, the KSV can make sure that no two hosts use the same SPI to send data to the same peers. In order to conserve SPI space, the (SPI, sender) tuple can be used instead of just the SPI to identify the SA uniquely at the receiver. However in our application, we use the SPI value directly.

Security Associations can be either inbound or outbound. Each of these SAs can exist independent of each other and can also have separate policies. Links at the KSV are specified with some directionality. In our configuration file we specify outbound links using the symbol “ =>”. In order to define inbound and outbound links with symmetrical policy, we also specify a bi-directional link using the “<=>” symbol. Examples of configuring links with such directionality are given in section 2.2.3.

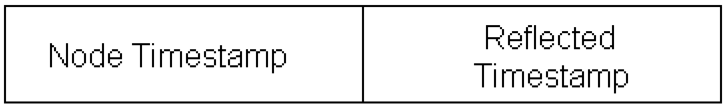


Figure 17 Fields in ACK Message

The KSV can specify some additional operations to be performed as soon as the given set of SAs is installed at the receiver. An indication of which operation is to be performed is passed as flags in the register message.

REFRESH: The KSV has specified this set of SAs as a complete set and not an incremental update. Any older SAs that had been created as part of earlier registration or update processes have to be deleted.

ACK_REQD: The KSV has asked the node to send back a confirmation about having received and installed the SAs. Since IKE messages are sent using the unreliable UDP protocol, this is particularly useful in some instances where we want to guarantee that a particular SA was successfully installed. A timestamp returned by the node can be used by the KSV to determine the order in which SAs were installed at the node if required. In order to identify which out of a number of possible states an ACK value corresponds to, the client also reflects back the server's timestamp value in its ACK message. The state corresponding to the (server, node, timestamp) tuple is guaranteed to be unique.

The format of the ACK message is specified in Figure 17. We summarize the information contained in a registration message in Figure 18 below:

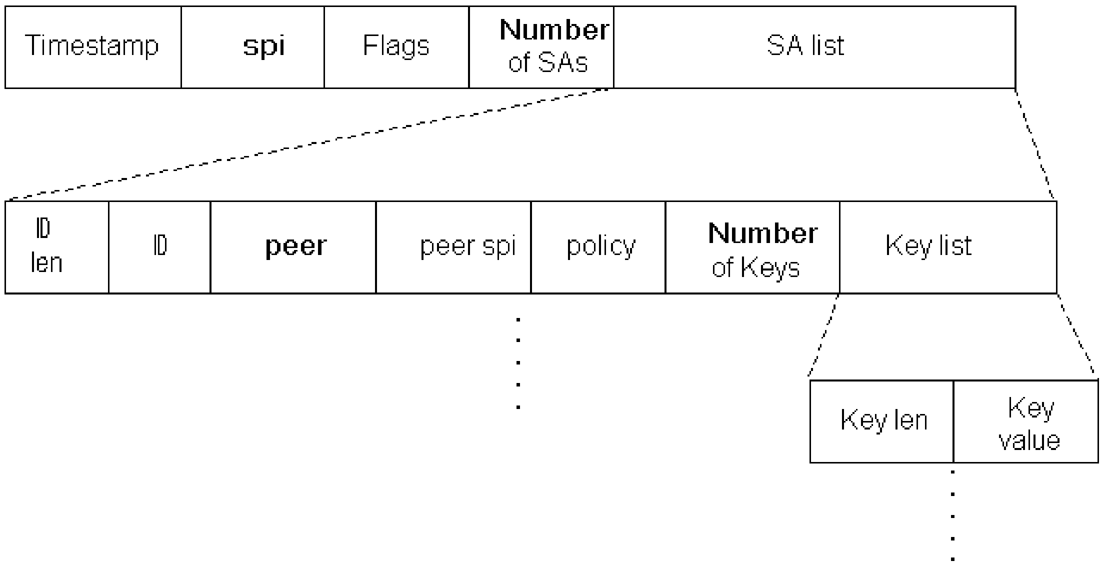


Figure 18 illustrates the fields in the Registration Message.

A few of the fields defined in Figure 18 are used in the context of multicast groups and will be explained in section 2.2.2.6. Nodes that do not belong to any multicast group have the "ID len" and "Number of Keys" fields set to zero.

Once the information necessary to form an SA is received by the node, it goes ahead and instantiates the same. No negotiation between the two nodes or costly DNS lookups is required because KSV has already performed this operation for it. We note that since all SAs are created artificially, in most cases there is no existing connection corresponding to this source and destination present at either the KSV or the node. Hence we define a new operation to create a connection "on the fly". The connection name used to index this connection in the connection database is also created dynamically by simply concatenating the IP addresses of the two SA endpoints. For example an on-the-fly connection created between 192.168.1.1 and 192.168.1.2 created on the fly is named " _192.168.1.1_192.168.1.2_".

2.2.2.1 SA Deletion

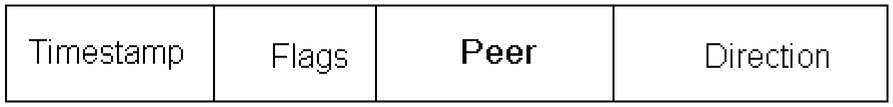


Figure 19 illustrates the fields in the Delete Message.

The process of deleting an SA is very similar to that of creating one using the keying server. However, we only need to send enough information to identify the SA or more specifically, the peer and the SA directionality. Figure 19 specifies the format of fields required in the delete message. We also specify the flag and timestamps fields for reasons mentioned in section 2.2.2.

When a node receives a "DELETE" message from the keying server for a particular SA, it identifies and deletes all the states associated with the same. Deletion of these states automatically cleans up any connection that may have been created on the fly also described in section 2.2.2.

2.2.2.2 Alarm Indication

Messages that are sent by the Keying Server do not fail silently if they are not able to perform their expected behavior. The affected node sends back some error indication, which the server can simply log or use directly in order to rectify the problem. We identify the following types for alarm indications:

SA_EXPIRED

MALFORMED_PACKET

SERVER_CONFLICT

SIGN_FAILED

Node Timestamp	Alarm type	**Peer**

Figure 20 illustrates the fields in the Alarm Message.

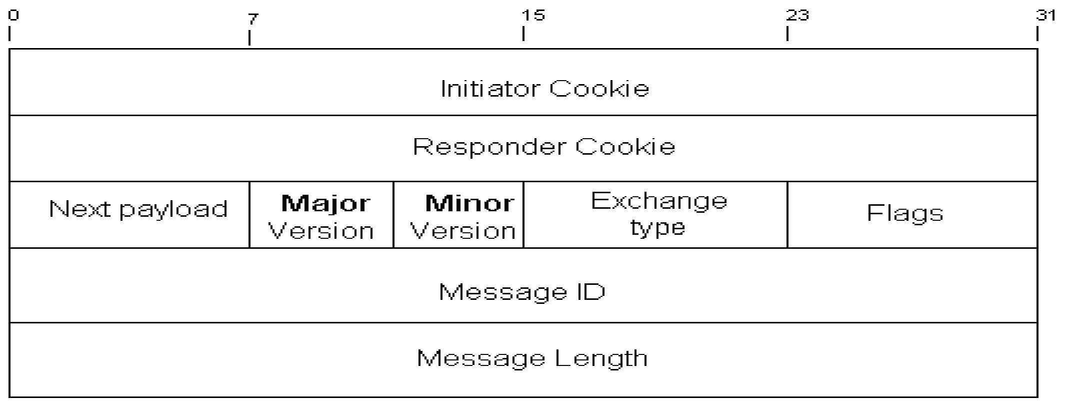
Security associations have a limited lifetime. We need to re-key a particular SA before the latest one expires so that the link security is never compromised. In case a new key is not registered before the latest one expires, the node continues to use the existing SA, renewing its lifetime to the default value. The alarm "SA_EXPIRED" is however sent to the keying server to indicate that such an event has taken place.

A "MALFORMED_PACKET" indicates that some packet created by the server was not understood properly by the node. This is different from the malformed packet at the network layer, which we handle by retransmission mechanisms at the TCP level or drop entirely in the case of UDP. Reception of such an alarm at the KSV is usually an indication of either someone attempting (unsuccessfully) to spoof or replay messages from the server or simply a loss of synchronization between the keying server and this node. In the case of the latter, the server can simply perform a refresh operation (Section 2.2.2.3) to bring the node back into a state of sanity.

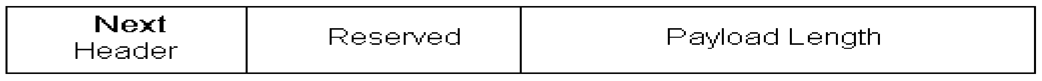
Some configuration errors such as two keying servers independently specifying the same node to be a part of the same multicast group can be detected at the node. If so, the node sends the "SERVER_CONFLICT" message to both the servers it has registered with.

The SIGN_FAILED alarm is used in the context of secure multicast groups, which we describe in section 2.2.2.6.

2.2.2.3 Information Packaging



(a) The ISAKMP header



(b) The ISAKMP Generic Header

Figure 21 illustrates the ISAKMP header fields.

In order to integrate the key registration process with IKE, we need to send the information in a form that IKE understands. IKE uses the ISAKMP protocol to specify the message formats sent between the two peers during various exchanges.

The details of the ISAKMP protocol itself are given in RFC 2408. Messages exchanged in an ISAKMP-based key management protocol are constructed by chaining together ISKMP payloads to an ISAKMP header (Figure 21(a)). There are thirteen distinct payloads that all begin with the same generic header (Figure 21(b)). Payloads are chained together in a message using the "next payload" field in the generic header. The ISAKMP header describes the first payload following the header and each payload describes which payload comes next. In our framework, we use the ISAKMP notification payload (Figure 22) within an Informational Exchange (Exchange type = 5) to send keying information from the KSV to the node.

The notification payload is generally used to send status data from a process managing the SA database to a peer process. ISAKMP defines two blocks of Notify Message codes, one for errors and one for status messages. It also allocates a portion of each block for private use within a particular Domain of Interpretation.

We define the following notification message types for use within the keying server framework with the values for these types taken from the private section of the status message codes:

KEYEXCHANGE_ACK = 32768

KEYEXCHANGE_REGISTER = 32769

KEYEXCHANGE_DELETE = 32770

KEYEXCHANGE_ALARM = 32771

The values within the notification payload are as specified in RFC 2408 except for the notification data field, which is decided by notification message type of this payload. For the notification types defined in our system - KEYEXCHANGE_ACK through KEYEXCHANGE_ALARM we define the notification data exactly as shown in Figure 22.

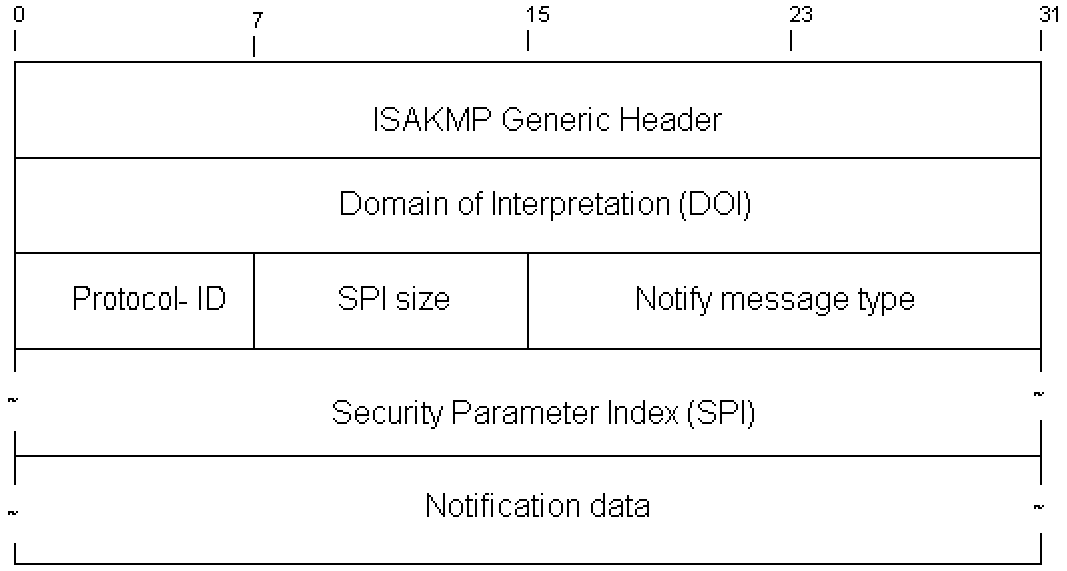


Figure 22 illustrates the ISAKMP Notification Payload.

The Informational Exchange messages are protected using the IKE SA that has already been created. This is done by chaining the notification payload with a hash payload and encrypting the entire ISAKMP message, except for the fixed part of the header. Only the IKE peers are able to read or modify the notification message contents.

2.2.2.4 Installing IPSec Security Associations

The Freeswan KerneL IPsec Support (KLIPS) implementation provides us with a mechanism for installing and maintaining Security Associations within a node. KLIPS hooks into the routing code in a Linux kernel. Traffic to be processed by an IPsec SA must be directed through KLIPS by routing commands. Pluto implements ISAKMP SAs itself and after it has negotiated the characteristics of an IPsec SA, it directs KLIPS to implement it.

Destination	Gateway	Mask	Flg	Iface
129.237.126.215	0.0.0.0	255.255.255.255	UH	eth0
129.237.126.212	0.0.0.0	255.255.255.255	UH	ipsec0
192.168.1.0	0.0.0.0	255.255.255.0	U	eth1
129.237.120.0	0.0.0.0	255.255.248.0	U	eth0
129.237.120.0	0.0.0.0	255.255.248.0	U	ipsec0
127.0.0.0	0.0.0.0	255.0.0.0	U	lo
0.0.0.0	129.237.127.254	0.0.0.0	UG	eth0

Table 4 Routing table containing an ipsec route.

KLIPS is implemented as follows. Every physical network interface is associated with a corresponding virtual interface. These virtual interfaces are named ipsec0, ipsec1, etc and each of them corresponds to one physical interface. Whenever a new SA is to be installed either manually or using the pluto IKE daemon, a new route referencing this virtual interface is installed in the kernel routing table. We observe in Table 4 that packets sent out of the source 129.237.126.215 to the destination 129.237.126.212 have been routed through the virtual ipsec0 interface instead of the physical eth0 interface.

The SA corresponding this connection is itself added to the SADB, which is maintained as a hash table in kernel memory. IP packets belonging to a particular SA now pass through the ipsec virtual interface as per the new route entry rather than any physical interface. IPSec transforms are applied based on the security association retrieved from the SADB and the keys corresponding to this SA.

In order to inform KLIPS about the SA-profile and its associated secret keys the KLIPS implementation uses the PF_KEY generic key management API. PF_KEY is a socket protocol family used by trusted privileged key management applications to communicate with an operating system's kernel-mode implementation of IPsec. Without going into too much details of PF_KEY sockets, we note these sockets provide a generic interface for key management applications (such as IKE, GKMP) for inserting and retrieving security association information inside the kernel using a set of predefined messages. This eventually makes the keying applications more portable.

In our prototype, we use the KLIPS implementation as is.

2.2.2.5 Extending the IPSec security associations for multicast data

IPSec is inherently a point-to-point protocol. One side encrypts and the other side decrypts using some shared key either statically configured into it or dynamically generated using IKE. Securing multicast data is a totally different paradigm since there are multiple recipients of a single packet and often, many senders to the same multicast address.

Some aspects of IKE fail when viewed in a multicast context. The SPI value, which is used by the destination to uniquely identify the sender of a given IPSec packet, has a problem with multiple recipients. It is not possible for an SPI to be unique at all destinations in this multicast group without making the negotiation process too complex. Even if this was possible we now have the problem of the entire SPI space being shared by all multicast nodes rather than each of them maintaining their own individual SPI space. A better approach and the one that matches our prototype well, is for the multicast server (the KSV in our prototype) to define and distribute the SPIs for each multicast group itself.

Another aspect that fails in a multicast scenario is that of source authentication and replay protection. There is no way to synchronize the anti-replay counters when all nodes sharing the same secret can send data on the same multicast address. In our implementation we simply turn off replay protection and limit source authentication to lighter forms of "one within the group" rather that stricter definitions.

In order to have multicast support within KLIPS, we need to make a few modifications.

KLIPS is modified to treat any multicast packets, as destined to itself. Multicast packets are only dropped if no application has been registered on this node to read multicast packets sent on this address.

As of this writing KLIPS is incapable of handling IP packets with a header size greater than 24 bytes as in the case of IGMP. We modify this behavior to allow the latter to go through

We also note that it is not possible to have an inbound and outbound SA for a given multicast address. This is because for both cases, the destination address and the SPI remain the same. We do not know the sender beforehand and so cannot remove the SA ambiguity using the source address. Hence we install only an outbound SA. The incoming packets use this same SA in order to decrypt packets sent to this multicast address

In order to allow any side to be able to find the SA for a multicast packet uniquely, each node now uses the common group SPI as both the sending as well as the receiving SPI. Multicast SAs defined in this manner are always bi-directional which, intuitively, seems to be perfectly reasonable.

2.2.2.6 Integrating the keying framework for multicast groups

An important issue in secure multicasts is that of member revocation. We use the LKH approach in our prototype implementation for multicast group member revocation. We decide to use this approach as against other methods with the reasoning that other implementations are simply variants of this basic approach. Building the basic approach helps us to understand and incorporate any newer approaches much more easily.

We note the following important property of the hierarchical tree - all the members are located at the leaf. Hence we construct the hierarchical tree as a B+ tree. We however do not link the leaves as we would normally do in a B+ tree simply because it serves us no useful purpose in the context of our implementation.

Nodes are added by comparing an "index" with the "current" node-index and traversing the tree based on the comparison result. The index would be a value that uniquely defines the host or any member value. We could think of maintaining this index as an IP address type, but in order to leave our implementation more generic we define this value as a character string.

Each node within in the B+ tree has the following contents:

A value specifying the number of entries in that particular node

The node key and its length

An array of values corresponding to every index maintained by this node

A shared key associated with each node in case this node is a leaf

An flag indicating whether the latest keying information for this node has been sent out

Pointers to the child-trees

The KSV defines a multicast group, which it uses to send key update messages to all members defined within one of its groups. This address is either known beforehand to each of the clients or is sent as part of the first register message sent by the KSV. In our implementation we have this value well known by all the clients. Clients that lie in the trusted set of a particular KSV need to bind itself to this group in order to receive the key update messages sent by its KSV. We however need to take care of a few issues.

There needs to be some way for the server to inform the clients about the key it is using to encrypt the current key-update message. A level indicator can identify this value – however there can be multiple nodes at the same level and hence each with the same level-indicator. For example in Figure 23, both keys 12 and 34 have a level-indicator as 2 (considering the root as level 0). In order to remove this ambiguity, we define a new method for identifying the encrypting key.

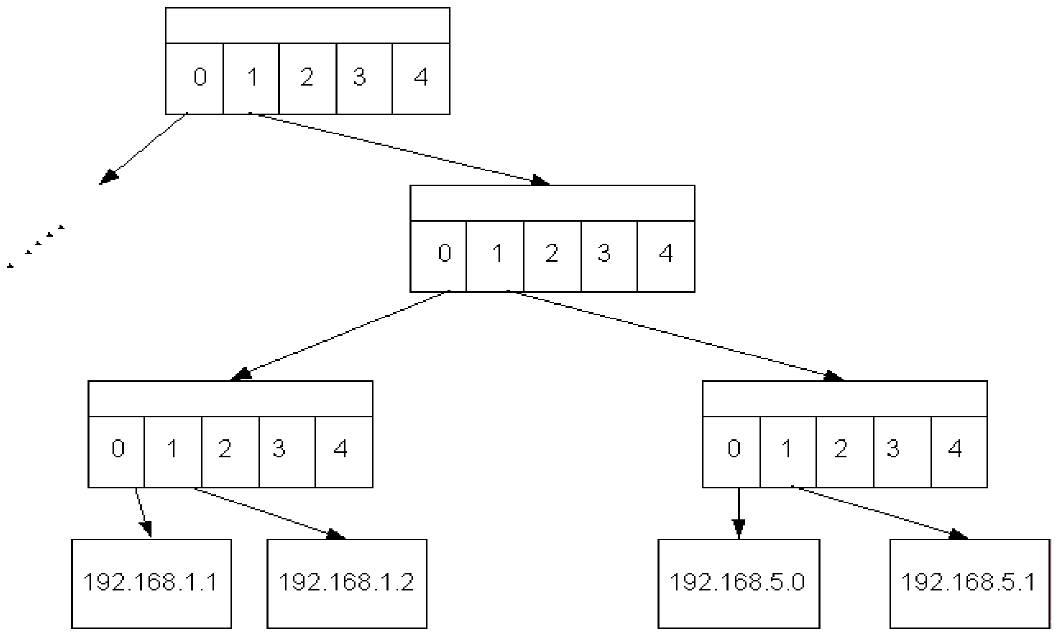


Figure 23 illustrates generating group member IDs.

We define a new ID for each member within a group by encoding the branch indices as an integer value, taken, in order to traverse the hierarchical tree from the root to this member. Hence the ID corresponding to node 192.168.1.1 in Figure 23 is "X100" while the ID for node 192.168.5.0 is "X110". The leading "X" identifies the particular group in the list of all the groups maintained by this KSV. Intermediate key IDs are simply subsets of the leaf IDs – hence to denote the parent node of indexes 192.168.1.1 and 192.168.1.2 we simply specify its ID as "X10". These ID values and their associated lengths are sent in the node registration message (Figure 18) in order to identify a group member completely.

We note that having long IDs could potentially lead to too many bytes being consumed just for defining the ID field. We optimize this slightly by defining each index as a byte instead of an integer, in effect limiting the maximum allowable order for the B+ tree. This seems to be a reasonable balance because bytes themselves have a range of 256 values and our B+ tree is not expected to be of such large orders in any case.

In order to use the multicast key update channel reliably, every message that is sent by the KSV needs to be signed. We see that this is true because using shared key approaches, it is always possible for a node containing the shared key to spoof messages from the KSV and hence confuse any revocation attempt by the KSV.

We summarize the protocol for multicast group member revocation using the update channel.

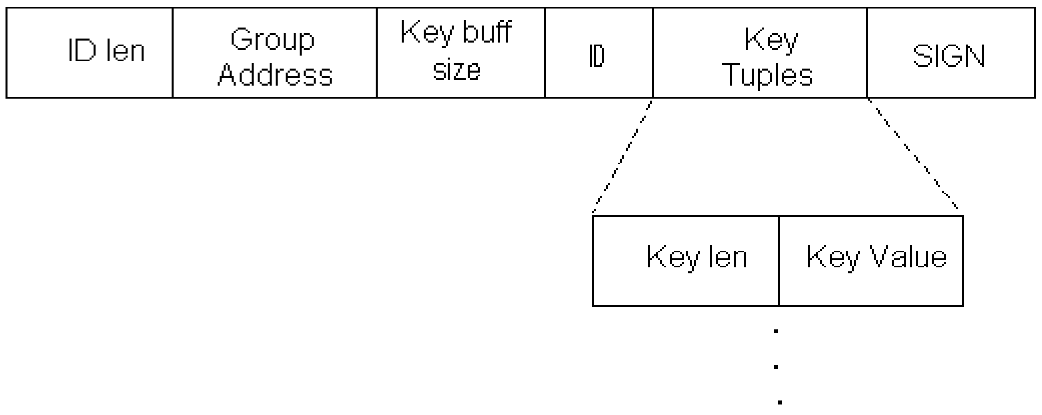


Figure 24 shows the multi-cast key-update message format.

Nodes that are defined to be a part of the KSV's trusted set are allowed to register with it. During this registration process, the KSV hands out the IDs corresponding to every group that this node belongs to. A client maintains its ID information for every server it has registered with and for every group it belongs to. In case it finds itself configured by two different KSVs as belonging to the same multicast address, it sends a SERVER_CONFLICT message to each of those servers.

Whenever the hierarchical tree at the KSV changes, an update message (Figure 24) is sent. The receiving node is able to identify without any cryptographic procedures if the update message belongs to it by simply checking the ID in the received message. Update messages that do not contain a valid signature are ignored and a SIGN_FAILED alarm (Section 2.2.2.2) is sent back to the KSV. In order for the group update process to work reliably we need some guarantees from the multicast channel itself. The algorithm will work incorrectly if packets are lost in the multicast channel. We however treat this issue as one inherent of multicast communication and beyond the scope of our problem.

Adding a new member to a multicast group can be a very costly process in this approach. This is because, in order to keep the B+ tree balanced, ID values of different nodes may change when a new member is added. New ID values for all affected nodes then need to be sent separately using register messages. It may be possible to optimize this process by delaying additions to the B+ tree until the next delete operation but we do not explore this option further.

2.2.2.7 Integrating DNSSEC

We use the BIND 9.1 implementation for providing DNSSEC services. The operations needed to set up the Secure DNS server are detailed in Appendix B. In this section we describe how we integrate the liblwres library for invoking DNSSEC services from our DNSSEC server.

The lwresd program provided in the BIND distribution, is a daemon for lightweight DNS resolvers. This daemon provides name lookup services to clients that use the liblwres library. It is essentially a stripped-down, caching only name server that answers queries using the BIND 9 lightweight resolver protocol rather than the DNS protocol. Incoming lightweight resolver requests are decoded by lwresd, which then resolves them using the DNS protocol. When the DNS lookup completes, lwresd encodes the answers from the name servers in the lightweight resolver format and returns them to the client that made the original request.

The liblwres library provides an API "lwres_getrrsetbyname( )" in order to get a set of resource records associated with a hostname, class and type. After a successful call to this function, a list of resource records and potentially another list containing SIG resource records are filled up. The lwres daemon automatically checks these signatures for validity and indicates success or failure by setting or clearing the RRSET_VALIDATE flag respectively.

Using the "lwres_getrrsetbyname( )" API, the KSV as well as nodes belonging to its trusted set are able to query for public key RRs and know if they are authentic simply by testing the RRSET_VALIDATED flag. These public keys can then be used to authenticate messages during the IKE phase-1 exchange or in the multicast key-update channel.

2.2.2.8 Integrating the packet filter

This security framework defined in its present form yet has a serious problem – spoofing is still very easily possible. To see how this is possible we again allude to the Freeswan IPSec implementation.

```
up_host:
        ipchains -A input -d $PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET -p 50 -j ACCEPT
        ipchains -A input -d $PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET -p 51 -j ACCEPT
       ipchains -A input -d $PLUTO_MY_CLIENT -s
$PLUTO_PEER_CLIENT_NET -p icmp -j ACCEPT
        ipchains -A input -d $PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET -i ! $PLUTO_INTERFACE  -j DENY

down_host:
        ipchains -D input -d $PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET -p 50 -j ACCEPT
        ipchains -D input -$PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET -p 51 -j ACCEPT
       ipchains -D input -d $PLUTO_MY_CLIENT -s
$PLUTO_PEER_CLIENT_NET -p icmp -j ACCEPT
        ipchains -D input -d $PLUTO_MY_CLIENT  -s
$PLUTO_PEER_CLIENT_NET –i ! $PLUTO_INTERFACE -j DENY
```

Figure 25 lists the ipchains rules for enforcing inbound policy.

The sending end of a Security Association ensures that packets that are sent are afforded all the necessary transforms. At the receiver end, packets with the correct transforms, using the correct keys are accepted, while those failing this step are considered to be bad. There is nothing however, which checks if the sender has simply abstained from IPSec processing altogether. The sender or any third party can change the source address on the packet without adding any IPSec headers and the receiver would just go ahead and treat this as a normal packet. This problem arises basically due to the lack of an inbound policy check in the Freeswan implementation. We can however provide an external policy check using a simple packet filter such as IPChains.

The pluto implementation has a mechanism whereby system commands can be executed as soon as the connection from or to a particular host comes up or goes down. This interface which is actually a script file provides a very easy way to set the packet filter rules at the appropriate places. Using IPChains the script would have the format as shown in Figure 25. The –d option specifies the destination while -s specifies the source. These values are passed as arguments to the script. The values - 50 and 51 in Figure 25, signify the ESP and AH protocols respectively while the –j options specifies the action to be taken when such a packet is received. The above rules state that when a connection comes up for this source and destination, only the packets containing either an ESP or an AH protocol header should be accepted. This rule is rescinded as soon as the connection goes down.

2.2.2.9 This framework and the ABONE Hop-By-Hop Security Architecture

The fundamental difference between our prototype and the ABONE Hop-by-Hop framework is that while we define security services at the network layer, the latter defines all the security services at the ANEP layer. We summarize a few differences between the two approaches in Table 5.

The Keying Server Prototype	ABONE Hop-by-Hop framework
Defines Security Associations in the Network Layer	Defines Security Associations in the ANEP layer
Identifies the SA using the receiver IP address and the SPI	Identifies the SA using the sending interface and the key identifier
Uses existing sequence numbering and replay protection within the IPSec implementation	Needs to define this separately since replay protection is done at the ANEP layer
Do not have a clear picture of the active packet's message boundaries	Clearly knows the message boundaries, hence can perform services such as non-repudiation more effectively
Need to secure every packet from the source to the destination irrespective of whether it is an ANEP packet or no	Can be more efficient in terms of speed because ANEP packets as opposed to individual IP packets are being secured
Defines a key management protocol to distribute authentication keys among peering nodes	No such protocol has been defined in this framework as yet although key management interfaces have been defined

Table 5 lists a comparison between the Keying Server and the ABONE Hop-by-Hop security frameworks.

2.2.3 Defining a secure topology

In order to use pluto as a keying server, we need to run the daemon as follows:

```
pluto --keyserver <config file>
```

The configuration file is used to set up the keying server using the trusted set given in the configuration file.

2.2.3.1 Configuration file Grammar

We specify the grammar for the configuration file used while running pluto as a keying server using Backus-Naur Form notation.

```
/* The configuration File consists of three sections – Policy,
Group and Link */

<Conf_File>:: <Policy_Section> <Group_Section> <Link_Section>
 | <Group_Section> <Link_Section>
 | <Policy_Section> <Group_Section>
 | <Policy_Section> <Link_Section>
 | <Group_Section>
 | <Link_Section>;


/* Specification for the Policy Section */

<Policy_Section>:: <Policy>
 |  <Policy_Section> <Policy>;

<Policy>:: “POLICY” <Policy_Name> <Policy_List> “END_POLICY”;

<Policy_List>:: <Policy_Element>
 | <Policy_List> <Policy_Element>;

 <Policy_Element>:: "LIFE-SEC="{Decimal number of at least one
digit}
 |"LIFE-KBYTE="{ Decimal number of at least one digit}
 | "TRANS-TYPE=” “AH | ESP”;


/* Specification for the Group Section */

<Group_Section>:: <Group>
 | <Group_Section> <Group>;

<Group>:: “GRP” <Mcast_Addr> “,” “POLICY=” <Policy_Name> “MEMBERS”
<Member_List> “END_GRP”;
```

```
<Member_List>:: <Member_Addr>
| <Member_List> <Member_Addr>;

<Mcast_Addr>:: <Valid_Name>;
<Member_Addr>:: <Valid_Name>;


/* Specification for the Link Section */

<Link_Section>:: <Link_Set>
| <Link_Section> <Link_Set>;

<Link_Set>:: “LINK” <Link_List> “END_LINK”;

<Link_List>:: <Link>
| <Link_List> <Link>;

<Link>:: <Source> <Link_Type> <Destination> “,” “KEY=”
<Key_String> “,” “POLICY=” <Policy_Name>;

<Link_Type>:: “=>” | “□”; /* directionality of the link */

<Key_String>:: {Key_Len} “:” {base 64 encoding of the key}| {hex
encoding of the key}| {text value of the key};

<Key_Len>:: { One or more digits;}
<Source>:: <Valid_Name>
<Destination>:: <Valid_Name>
<Policy_Name>:: <Valid_Name>;

<Valid_Name>:: {Any valid IP address OR Any string starting with
a-z, A-Z followed by any number of a-z, A-Z, 0-9, minus or period
characters};
```

2.2.3.2 Setting up a sample secure topology

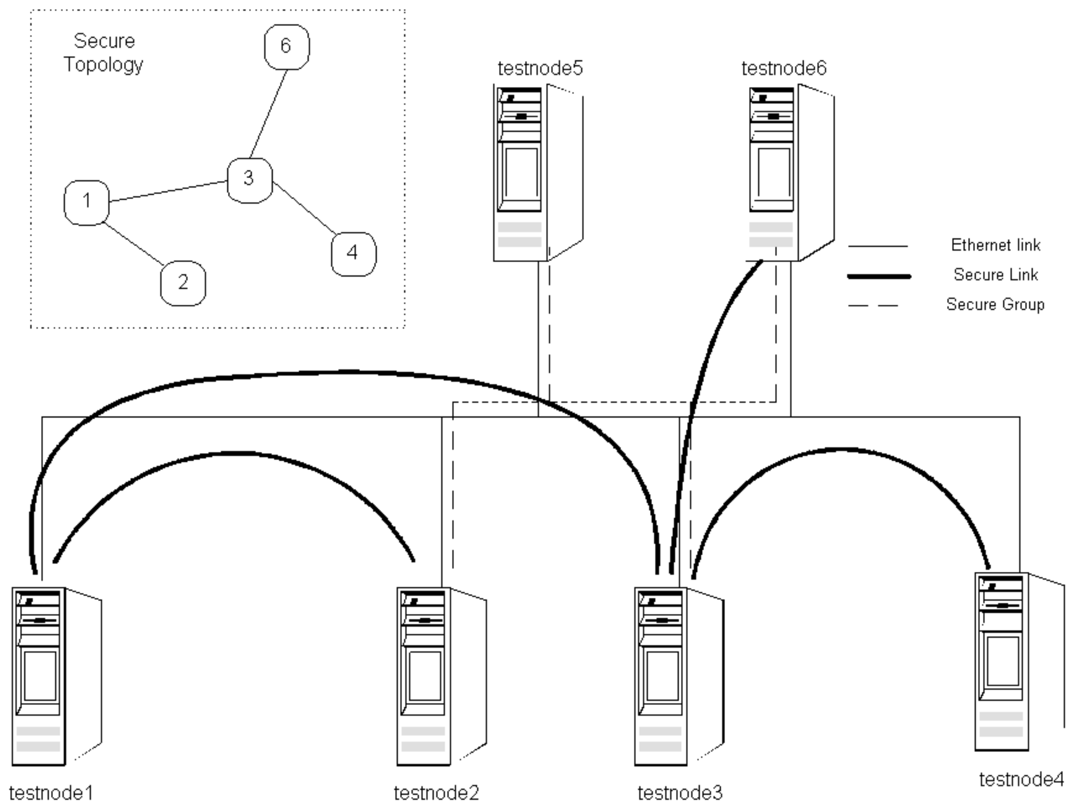


Figure 26 illustrates overlaying a secure topology on Ethernet.

Consider the Ethernet shown in Figure 26. Since Ethernet has a shared bus topology, a packet sent from testnode1 to testnode2 above would be broadcast to all other nodes on the same segment. Suppose we wish to overlay a secure topology over Ethernet in the following manner:

Any packet sent from testnode1 to testnode3 and from testnode3 to testnode4 should be authenticated in both directions

Any packet sent from testnode1 to testnode2 should be authenticated while packets from testnode2 to testnode1 should be encrypted.

Any packet sent from testnode3 to testnode6 should be authenticated but there should be no security processing in the reverse direction.

testnode2, testnode3 and testnode5 should send authenticated data when communicating on the multicast group 224.0.1.5

Such a topology can be easily defined using the configuration file shown in Figure 27.

```
POLICY auth
    TRANS-TYPE=AH
    LIFE-SEC=100
    LIFE-KBYTE=10000
END_POLICY
POLICY enc
TRANS-TYPE=ESP
    LIFE-SEC=100
    LIFE-KBYTE=10000
END_POLICY
GRP 224.0.1.5, POLICY=auth
MEMBERS
    testnode2
    testnode3
    testnode5
    testnode6
 END_GRP
LINK
        testnode1 <=> testnode3, KEY=0:0, POLICY=auth
        testnode3 <=> testnode4, KEY=0:0, POLICY=auth
        testnode1 => testnode2, KEY=0:0, POLICY=auth
        testnode2 => testnode1, KEY=0:0, POLICY=enc
        testnode3 => testnode6, KEY=0:0, POLICY=auth
END_LINK
```

Figure 27 lists a policy specifying a secure topology.

2.2.3.3 Configuration Commands

We provide various commands in order to interact with the Keying Server and define the secure topology. Before performing any of these commands, the keying server and clients should be started using the following commands respectively:

```
pluto --keyserver <config file>
pluto --keyclient
```

We integrate configuration management with “whack” [FreeSwan] as specified in Section 2.2.3.3. Hence our configuration commands have similar syntax as that of the native whack commands. We define the following commands for managing the keying server:

```
whack  --showconf
```

Since the configuration at the Keying Server is dynamic due to various links being added and deleted, we provide a mechanism to query the Keying server for its current configuration.

```
whack --keyinit-here <my-ip> --keyinit-there <peer-ip>
```

This command initiates the IKE key exchange between the nodes specified as <my-ip> and <peer-ip>. This command can be initiated by either the keying server or the client. This is so because there is no guarantee of which one out of the server and the client will come up first. The client could execute this command directly on startup or we could alternatively imagine the server doing this step if it has restarted with a new configuration file for example.

```
whack --add-link <server-ip> --host <source> --dir < uni | bidir >
--host <dest> [--policy <policyname>]  [--keymat <keylen:key>]
```

This command creates a new link from <source> to <dest>. Specifying the direction as "bidir" creates a symmetric pair of SAs. In case no policy is specified, the link will use the default policy. The key manager can manually specify the key material used for this link; in case keylen is 0, a key for the link is automatically created.

```
whack --delete-link <server-ip> --host <source> --dir < uni |
bidir >  --host <dest>
```

This command is used to delete an existing link between <source> and <dest>. The exact link specification has to be described. Thus, specifying "bidir" does not delete two links created using "uni".

```
whack --add-group <server-ip> [[--host <group-member> --to] --host
<group-addr>]  [--policy <policy-name>]  [--keymat <keylen:key>]
```

This command can be used either to create a new multicast group or to add a member to an existing group. <group-addr> specifies the group multicast address.

```
whack --delete-group <server-ip> [[--host <group-member> --to] --
host <group-addr>]
```

Using this command we can delete existing members from a group or delete the entire group itself if it is empty. As before, group-addr specifies the group multicast address.

```
whack --refresh --host <server-ip> --to --host <client-ip>
```

At Keying Server may at any time send all keying-information pertinent to a particular client. This may be necessary when, for example, this client has missed any keying updates on the multicast channel. This command can then provide synchronization between the Keying Server and the affected client.

2.2.4 Testing and Evaluation

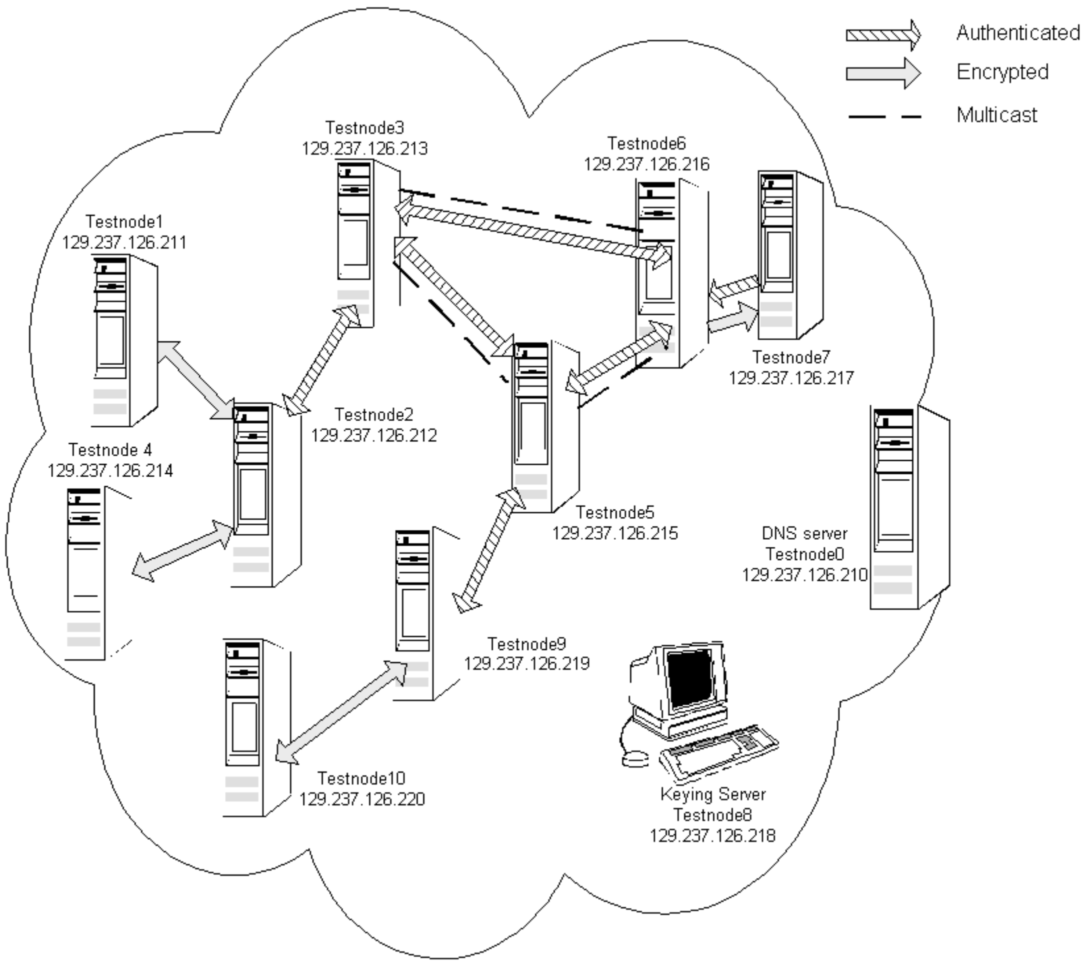


Figure 28 illustrates the security framework test configuration.

We use the topology shown in Figure 28 to test our Keying Server implementation. Each node in our topology is a 533 MHz Pentium III machine and is connected to 100Mbps Ethernet. We can define such a topology using the configuration file given in Figure 29 below:

```
POLICY auth
    TRANS-TYPE=AH
    LIFE-SEC=28800
    LIFE-KBYTE=86400
END_POLICY

POLICY enc
    TRANS-TYPE=ESP
    LIFE-SEC=28800
    LIFE-KBYTE=86400
END_POLICY
```

```
GRP 224.0.0.5, POLICY=default
MEMBERS
    testnode3
    testnode5
    testnode6
END_GRP

LINK
        testnode1<=>testnode2, KEY=0:0, POLICY=enc
        testnode4<=>testnode2, KEY=0:0, POLICY=enc
        testnode2<=>testnode3, KEY=0:0, POLICY=auth
        testnode3<=>testnode5, KEY=0:0, POLICY=auth
        testnode3<=>testnode6, KEY=0:0, POLICY=auth
        testnode5<=>testnode6, KEY=0:0, POLICY=auth
        testnode6=>testnode7, KEY=0:0, POLICY=enc
        testnode7=>testnode6, KEY=0:0, POLICY=auth
        testnode5<=>testnode9, KEY=0:0, POLICY=auth
        testnode9<=>testnode10, KEY=0:0, POLICY=enc
END_LINK
```

Figure 29 lists the configuration file for the test topology.

We divide the testing process into two parts:

Functionality Testing: To ensure that all our SAs have been set-up correctly

Timing Evaluation: Which places some of our results in perspective

2.2.4.1 Testing for Functionality

We observe the behavior of the DNSSEC authentication framework after inserting an unsigned KEY-Resource-Record corresponding to testnode5 in the DNS server (testnode0).

In this case, the IKE set-up process between the KSV and testnode5 fails with the following message: "Could not authenticate response from DNS server". A similar test with an incorrect SIG record also produces similar results. Hence we conclude that the DNSSEC framework has been set-up correctly.

```
Testnode6 # ipsec eroute
129.237.126.216/32 -> 129.237.126.213/32 =>
ah0xe69ad84f@129.237.126.213
129.237.126.216/32 -> 129.237.126.215/32 =>
ah0xe69ad84f@129.237.126.215
129.237.126.216/32 -> 129.237.126.217/32 =>
esp0xe69ad84f@129.237.126.217
129.237.126.216/32 -> 224.0.0.5/32       => ah0xe69ad84c@224.0.0.5
```

```
testnode6 # ipsec spi
ah0xe69ad85d@129.237.126.216 AH_HMAC_SHA1: dir=in
src=129.237.126.217 ooowin=64 alen=160 aklen=160
life(c,s,h)=add(8,0,0)
esp0xe69ad84f@129.237.126.217 ESP_3DES: dir=out
src=129.237.126.216 iv_bits=64bits iv=0x2529fcbc1469c51f ooowin=64
eklen=192 life(c,s,h)=add(8,0,0)
ah0xe69ad84e@129.237.126.216 AH_HMAC_SHA1: dir=in
src=129.237.126.215 ooowin=64 alen=160 aklen=160
life(c,s,h)=add(8,0,0)
ah0xe69ad84d@129.237.126.216 AH_HMAC_SHA1: dir=in
src=129.237.126.213 ooowin=64 alen=160 aklen=160
life(c,s,h)=add(8,0,0)
ah0xe69ad84f@129.237.126.215 AH_HMAC_SHA1: dir=out
src=129.237.126.216 ooowin=64 alen=160 aklen=160
life(c,s,h)=add(8,0,0)
ah0xe69ad84f@129.237.126.213 AH_HMAC_SHA1: dir=out
src=129.237.126.216 ooowin=64 alen=160 aklen=160
life(c,s,h)=add(8,0,0)
ah0xe69ad84c@224.0.0.5 AH_HMAC_SHA1: dir=out src=129.237.126.216
alen=160 aklen=160 life(c,s,h)=add(9,0,0)
```

Figure 30 lists the ipsec eroute and ipsec spi command results.

We define a "correctly setup" Security Association as one in which peers are able to reach one another after applying the proper IPSec transforms. We trust the verity of the underlying IPSec implementation with respect to providing us with the correct AH and ESP transformations. Thus, simply knowing that packets arrive with the proper encapsulation is sufficient for us to determine that the SAs have been set up correctly. The commands "ipsec eroute" and "ipsec spi" can be used to observe the different Security Associations that have been set up. The results after executing the above commands on testnode6 are shown in Figure 30.

"ipsec eroute" shows us that IPSec processing must be done for packets destined to testnode3, testnode5, testnode7 and the multicast address. Each eroute also points to a corresponding SA in the SADB.

"ipsec spi" displays the contents of the SADB. The fields displayed for a particular Security Association are as follows:

- The SA identifier
- The protocol used for the IPSec transformation
- The directionality of the SA – inbound or outbound
- The source with which to associate this SA with
- The Replay window size
- Initialization Vectors in the case of ESP
- Authentication or encryption key length
- Lifetime set for this Security Association

Thus in order to send a packet to testnode5 (129.237.126.215) for example, testnode6 performs the AH transform using the SA ah0xe69ad84f@129.237.126.215. The entry for this SA in the SADB confirms AH processing using the HMAC-SHA1 algorithm with a key-size of 20 bytes.

We check connectivity between the hosts Testnode6 and Testnode5 by performing simple application-level "ping" tests. We simultaneously check if the packet at the other end was received with the proper encapsulation by inserting the logging option, "-l", in an appropriate filter rule. We log all our results in /var/log/kern.log.

In our test scenario, packets are logged when any of the following conditions are met:

ACCEPT: A packet from testnode6 has either of the AH or ESP in the transport protocol field

DENY: A packet from testnode6 does not have the above transforms

If any log entry corresponding to rule 2 shows up, it means that some IPSec transformations was not applied correctly.

```
testnode5#  tail  -f   /var/log/kern.log
Jun  3 20:42:44 testnode5 kernel: Packet log: input ACCEPT eth0
PROTO=51 129.237.126.216:65535 129.237.126.215:65535 L=108 S=0x00
I=13789 F=0x0000 T=64 (#1)
Jun  3 20:42:45 testnode5 kernel: Packet log: input ACCEPT eth0
PROTO=51 129.237.126.216:65535 129.237.126.215:65535 L=108 S=0x00
I=13791 F=0x0000 T=64 (#1)
```

Figure 31 lists packet filter logs.

The absence of DENY entries in Figure 31 above, confirms that packets sent from testnode6 to testnode5 have the proper IPSec-AH encapsulation. Testing other links in a similar manner assures us of the fact that all the Security Associations have been set up accurately.

Multicasting, at the time of this writing suffers from a curious problem. We tried testing multicast using a simple reader-writer application on a multicast address. While security associations set up correctly, packets were still not detected at the readers. One observation we made was that the IGMP join messages themselves were not being sent out of the readers, which led us to believe that there was some filtering happening at the Ethernet layer. This suggested that packets routed through an "eroute" causes this behavior. We could not confirm this fact however. We supply a temporary fix at this stage by modifying the Ethernet driver code to accept all multicast packets seen on the Ethernet bus. The readers and writers now function as expected.

We also check the operation of our LKH implementation as given below.

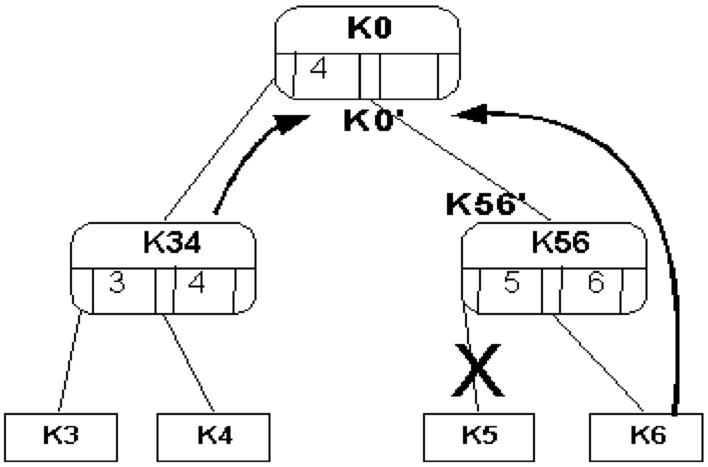


Figure 32 illustrates a LKH tree for our sample secure topology.

We first add testnode4 to our multicast group using the command:

```
testnode5# whack  --add-group 129.237.126.218 --host
129.237.126.214 --to --host 224.0.0.5
```

Our LKH tree now has the form shown in Figure 32. We observe the key-update behavior using LKH by revoking testnode5 from our multicast group:

```
testnode5# whack  --delete-group 129.237.126.218 --host
129.237.126.215 --to --host 224.0.0.5
```

The following key updates can be detected:

An update message with ID = 111 corresponding to keys K56′ and K0′ encrypted using the key K6

An update message with an ID = 10 corresponding to key K0′ encrypted using the key K34

This observation agrees well with the requirements of the LKH mechanism.

2.2.4.2 Summary and Timing Evaluation

From our tests in Section 2.2.4.1 we have verified and demonstrated the following:

Correct operation of the DNSSEC authentication framework

Our ability to successfully set-up and configure node-to-node security associations using a keying server.

Correct operation of the multicast key update channel during a member revocation.

DNSSEC processing typically takes about 2.5ms on average. In order to investigate the latency imposed by IPSec encapsulation, we run a simple application-level "ping" and examine the round-trip overhead. Expectedly, IPSec processing introduces a significant overhead. While the average RT-time between two nodes without a Security Association between them is 0.58 ms, the same, using either IPSec ESP or AH processing is 1.2 ms.

A more interesting problem, however, is comparing the time taken for explicit keying between every KSV-client pair and hierarchical keying using the LKH mechanism whenever a member is revoked. We perform this test as follows:

Configure a multicast group containing all our testnodes except the DNS server and our Keying Server

Register every node with the KSV so as to maximize the number of keys we may have to send during a re-key

Remove all our nodes from the multicast group beginning from testnode10 incrementally so as to trigger any keying updates on our multicast channel

Perform the same operation this time using individual KSV-client re-keys for every node currently registered at the KSV

Repeat steps 1 through 4 for different orders of the LKH tree

Table 6 and Figure 33summarize our comparison results for explicit keying (EK) and hierarchical keying using the LKH approach.

Revoke this testnode =>		10	9	7	6	5	4	3	2	1
Order = 1	EK	4.4	3.8	3.3	2.8	2.3	1.7	1.2	.6	-
	LKH	236.2	235.3	158.1	157.0	89.6	166.1	78.5	78.6	-
Order = 2	EK	4.4	3.8	3.2	2.7	2.2	1.7	1.1	.6	-
	LKH	336.5	235.6	157.0	235.3	156.9	78.5	156.8	85.3	-
Order = 3	EK	4.4	3.9	3.2	2.7	2.2	1.7	1.1	.6	-
	LKH	414.5	314.0	235.2	163.1	90.0	244.9	164.0	78.5	-

Table 6 Time in ms taken for Explicit (EK) and LKH keying for different Orders of the LKH tree

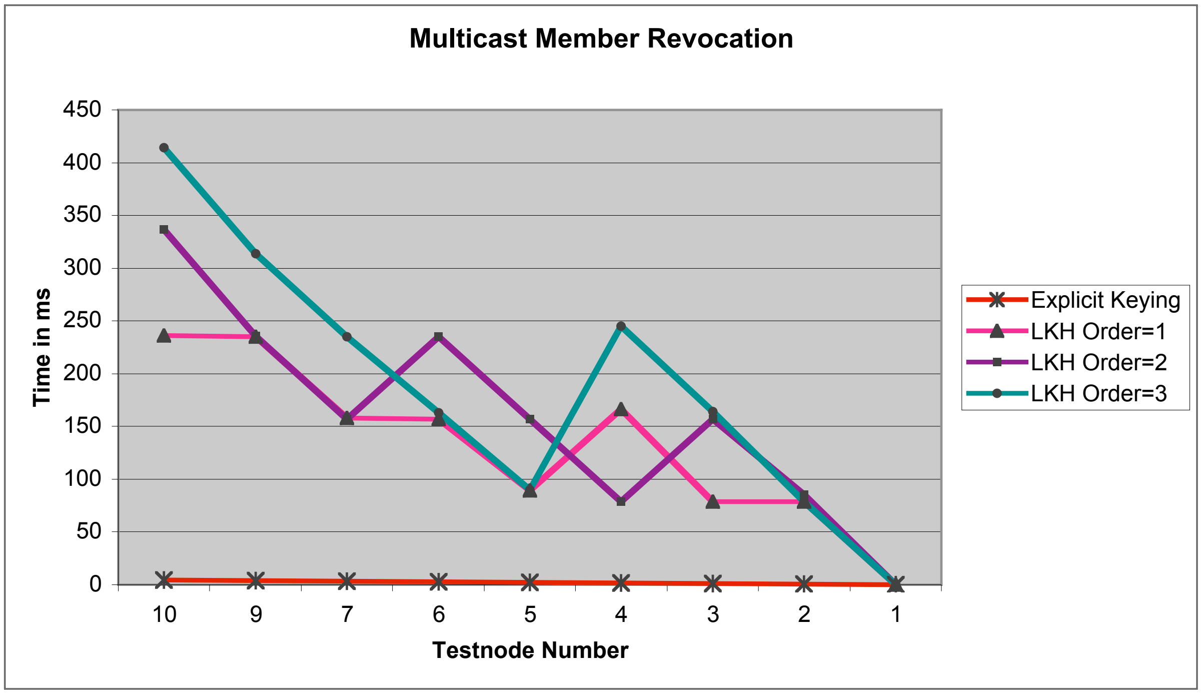


Figure 33 shows a comparison between Explicit Keying and LKH for multicast member revocation.

To investigate the behavior of the LKH approach, we need to see how the LKH trees are created in the first place. Figure 34 depicts this information.

We observe that the timing behavior for the LKH trees directly depends on the number of public-key operations required to be performed during the revocation process. Thus for the LKH tree with order=1, the number of public-key operations required when testnode10 is revoked is 3 – one for every sibling node - (5,6) and (7,9), and one for the adjacent branch. Revocation of testnode9 also has the same number of public-key operations – one for testnode7, one for its sibling node – (5,6) and finally one for the adjacent branch.

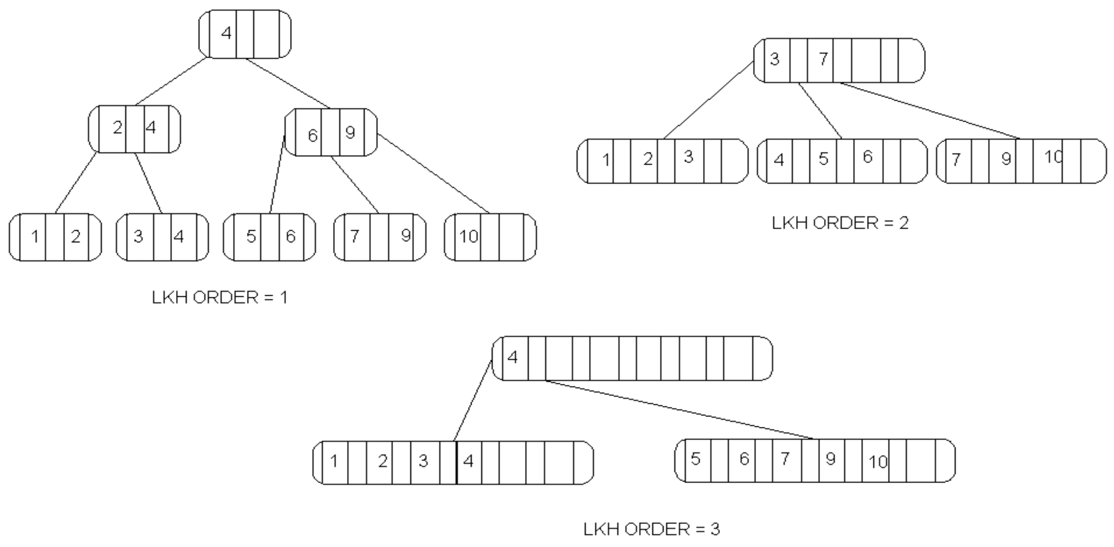


Figure 34 illustrates a LKH tree for tree orders of 1, 2, and 3.

This result is interesting because it tells us that simply increasing the order of the LKH tree does not necessarily improve the key-update time. On an average, the update time increases because there are more number of key transmissions required for larger orders. Even though a broader LKH tree requires a lesser number of keying updates *across* branches for propagating key revocations, the delay overhead caused by sending an update for every sibling in the current node is significant.

Finally, we observe that the LKH mechanism is more than an order of magnitude slower than its explicit key-exchange counterpart due to the expensive public key operations required in its implementation.

In summary, it is clear that for small number of nodes the advantage of the LKH algorithm is amortized by the overhead of the public key operations required in its implementation. Explicit keying for every peer also does not scale well and becomes comparable to the LKH mechanism as the number of nodes increase. An interesting future enhancement could be combining Explicit Keying with the LKH approach in order to reduce the overall latency during group member revocation.

Task 2: Innovative Active Networking Services

We believe that active networking capabilities fundamentally changes the dynamics of network protocol design and implementation. In traditional networks, relatively few protocols are available. These protocols must be widely accepted and therefore general to satisfy a wide range of demands. These protocols are implemented and maintained by a few specialized engineers with a strong emphasis on performance.

Active networking should allow a much wider community of users to design and implement protocols that are specialized for specific application demands and/or expected traffic patterns. To realize this potential, however, the environment for designing and implementing network protocols must change dramatically. We have identified several necessary features of this environment:

1. Safety and correctness. With many users deploying network protocols on shared resources, these key features of network protocols take on even greater importance. The user wants to deploy his/her protocol quickly, without requiring extensive testing in an isolated test environment. He/she may not have access to adequate test environments, and certainly does not want to expend this additional time and effort. Since network protocols operate on shared hardware there will be very grave concerns among the non-users of newly deployed protocols that they do not adversely affect their usage of these shared resources – either maliciously or unintentionally.
2. Minimize time and effort for implementation. The benefit of designing a specialized protocol will be weighed against the effort required for its implementation. If an application or environment can make marginal advantage of a specialized protocol, it will be implemented only if the required expertise and costs are small, relative to the potential gains.
3. Reusability. This is almost a corollary of the previous two items, but sufficiently important to list separately. Broadly speaking, network protocols perform a small collection of functions, such as forwarding, filtering, error checking and/or correction, although there are many alternatives and variations for each general function. Very often, new protocols will be a variation of or specialization of one such function within an existing protocol stack. The ability to reuse all components but the one to be modified has obvious advantages for both correctness and minimization of effort.
4. In order to implement new network services, protocol definition environments must provide facilities for the communication among independently developed and deployed protocols, in order for them to cooperate and provide the desired service.

We have designed and implemented an environment for developing composite protocols that emphasizes the features described above. The key features of this environment are:

1. Very fine decomposition. Each protocol component should perform a single, well-defined function. Verification that an individual component is safe and correct is much more tractable when the component is small and performs a single function. Additionally, a designer is able to maximize the usage of pre-existing, and tested components when they are defined at a fine level of granularity.
2. Each protocol component has a uniform external interface. Therefore, components can be physically connected in arbitrary order. There may be semantic dependencies that restrict connectivity to create valid protocol stacks, but these dependencies are explicitly expressed elsewhere. This also allows designers of individual components to work independently without agreeing on interface issues. This uniform interface is realized by providing other formal mechanisms to describe and implement inter-component communication rather than the traditional parametric interface.
3. A unique component composition operator is implemented in the framework.
4. Each protocol component is defined abstractly via a finite state machine and a set of memory dependencies. This abstract definition is amenable to formal analysis to detect inconsistencies and verify and specific functionality is accomplished. The actual implementation is automatically derived from the abstract definition.
5. Every external dependency of a protocol component is explicitly identified via a memory dependence. Making external dependencies explicit allows us to identify inconsistencies in network protocol definitions. It encourages component designers to minimize these dependencies. Finally, this focuses attention on the most likely areas for protocol errors – non-local interactions.

Technical details on the system are presented below and in the technical report, "Design and Implementation of Composite Protocols," (ITTC-FY2003-TR-19740-05) and [Minden2002].

2.3 Introduction

Active Networking anticipates an environment with rapid proliferation of new network protocols. Application-specific protocols can be injected into the network in advance of application data. In such an environment, it is no longer necessary to achieve global agreement on protocol details and wait for the acceptance and implementation of the protocol throughout the network, but one is enabled to quickly design, test, and deploy innovative protocols and network services. This encourages the development of a variety of specialized protocols by a wide range of programmers, rather than a few specialized protocol designers.

This environment places greater demands on ensuring the correctness of the protocols deployed. A faulty protocol used by a single application may have wide effect on the network infrastructure.

Our work addresses both of these issues, a methodology to more quickly design and implement a new protocol and at the same time provide increased assurance that the new protocol will have no negative impact on the network. We propose to define network protocols through the composition of simpler, reusable components, which are formally specified.

Partitioning a problem into smaller units, at least some of which can be reused, was one of the first techniques employed for software development, and as such, is not a new idea. However, it has typically not been used in the traditional implementation of network protocols. There are two dominant reasons for this:

1. Performance has always been of critical concern for protocol implementations. Clever programming tricks that could save a few instruction cycles and/or bytes of memory have been very highly valued.
2. Relatively few protocols have been implemented, and once implemented change very slowly over time. Therefore, reusability and ease of modification have been of relatively lower importance.

We believe that active networking significantly changes these priorities. Performance will remain an important issue, but correctness will become the most critical concern. We anticipate the development of a wide variety of protocols with only minor adaptations or specializations from existing protocols for specific applications. The ability to reuse the common components will significantly reduce the time and effort needed to develop specialized protocols, and will reduce the effort required to demonstrate correctness. A variety of other researchers agree and there is a growing body of research in the area of composite protocols.

2.3.1 Structure of Protocols and Services

Protocols are the major building blocks of network and distributed systems. However, rigorous definition of what constitutes a protocol is rare. Generally, protocols are described as having some mutually agreed upon set of messages and used to exchange information between peers. We first seek a definition of protocol that is: (a) sufficient to recognize a protocol when you see one, (b) suitable for rapid definition of (simple) protocols for implementation and experimentation, (c) suitable for machine manipulation, e.g. formal methods, and (d) allows extensions to more exotic formalisms. We provide such a definition for a protocol component in Section 2.3.2.

In our architecture, protocol components implement simple communications functions that are easily reused. Example functions are in-order delivery, reliable delivery, and message integrity. Protocol components are composed into protocols. Composition can take many forms. In our current implementation, protocol components are linearly connected. Protocols, i.e. a linear collection of protocol components, are what we normally deal with in networks and distributed systems. Primary examples are the Internet Protocol (IP), the Transmission Control Protocol (TCP), and Real-Time Protocol (RTP) among many others. Protocols can be collected into larger configurations to provide a network service. The implementation of a network services requires the implementation and deployment of two or more protocols. An example of a network service is multicast which requires protocols to join and leave a multicast group, a (possible) reliable delivery protocol, a routing protocol to link multicast servers, and a tunneling protocol to communicate between servers.

The remainder of this section focuses on the definition of protocol components and a straightforward method of composing protocol components into a protocol. The composition is called a protocol stack.

2.3.2 Essential Elements of a Protocol Component

We think of protocol components as layers in a protocol stack, similar to traditional networking layers, but at a much finer detail. On transmission, packets are accepted from a higher layer, operated upon, and sent to a lower layer. On reception, packets are accepted from a lower layer, operated upon, and sent to a higher layer. Further, protocol components transfer information between a local protocol component and remote peer protocol component over a communications channel. A formal definition of the communications channel is outside the scope of this paper.

A protocol component has the following elements:

1. Two finite state machines, one for transmission and one for reception,
2. A set of local functions,
3. A local memory, and
4. A set of properties.

The tokens are used to communicate state transition information from one end of the communication channel to the remote end of the communication channel. The best example of a token set is the TCP set containing SYN, ACK, and FIN. The most common, and many times explicit, token is 'PacketArrival'.

The finite state machines implement an orderly control sequence of messages and actions. In our current implementation we use augmented finite state machines to minimize the state space and facilitate formal analysis. The augmented structure supports an input token and guard expression on each machine transition. The augmented finite state machine is described below.

The local memory is used for temporary storage by the protocol component. One can think of the transmission buffers necessary for reliable delivery or the receive buffers required for in-order delivery.

The local functions operate on the local memory, packet contents, and cause the packet to transit up or down the stack.

The set of properties describe important attributes and requirements of the protocol components. Examples are 'Provides in-order delivery,' 'Provides message integrity,' or 'Requires reliable delivery.' These properties can be used to insure a particular composition of protocol components meets a global objective.

In addition, protocol components exchange information with other protocol components in the protocol stack, with global memory (used to communicate between protocols), and with a peer protocol component at the remote end of the communications channel. The interaction of these elements is described below.

2.3.2.1 Augmented State Machine Model

The augmented state machine model [Gurevich2000], which we use, consists of a finite set of states with a finite set of transitions from one state to another. We define the augmented state machine by transitions. Each transition is defined by the current-state, the next-state, an event, a guard-expression, an action function and a local-memory update.

1. Events trigger a transition from one state to another.
2. Guard expressions are Boolean expressions that conditionalize the transition from one state to another. A transition can only be activated by its corresponding event if the guard expression evaluates to true.
3. Action functions describe the response of the protocol component to the associated event. Action functions typically would consist of executing defined framework functions. Action functions should be limited to simple sequences of non-branching statements through proper use of guard expressions, synchronous transitions, and synchronous states.
4. The local memory update function modifies the component's local state after the execution of the action function.

The state machine may have synchronous states and transitions. A synchronous transition is one which does not have any event associated with it and is selected based solely on its guard expression. A state is said to be synchronous if all of its outgoing transitions are synchronous. Synchronous transitions are used to sequence actions within a protocol component.

2.3.2.1.1 State Machine Execution

The execution of a state machine is as follows. For any given non-synchronous state, the state machine changes its state only in response to an event. When an event occurs, a single transition (from the set of all outgoing transitions) is activated. The action function is executed, the local-memory is updated, and the next state is entered.

To determine the transition to be activated from the set of all transitions, first the transitions for which the event does not match are filtered out. Next, the guard expressions for each of the remaining transitions are evaluated. One and only one of these guard expressions must evaluate to true and the rest must evaluate to false. If none of the guard expressions are true, then we have an error condition of the finite state machine remaining in the same state. If more than one of the guards evaluates to true, then we have an ambiguous transition error condition. Because the guards are Boolean expressions, an algorithm can determine that one and only one transition is activated for each event.

A synchronous state transition occurs when all outgoing transitions are synchronous. In this case, all the outgoing transitions are selected as if they had matched an event, and their guard statements are evaluated through the same process as above.

The action function corresponding to the activated transition is executed. This is an atomic operation, as it is not pre-empted by the occurrence of another event in the state machine. In actual implementation, an event occurring during the execution of an action function shall be queued in order and sent to the state machine.

The local memory corresponding to the activated transition is then updated and the next state is set as the destination of the matching transition.

2.3.2.1.2 Protocol Component use of Augmented Finite State Machines

A protocol component defines two Augmented Finite State Machines (AFSMs): a Transmit State Machine (TSM) and a Receive State Machine (RSM). Protocol component initialization defines the initial state of each machine and the initial state of the local memory.

We currently use two events for our components:

PacketArrival

- PacketArrival (from above) occurs only in the TSM
- PacketArrival (from below) occurs only in the RSM

Timeout events

- Occur only in those SMs in which the corresponding timer was created.

The designer can define additional events, however we keep the number of events small to facilitate defining complete AFSMs and automatic checking of the protocol components and its interactions within the protocol stack. This limited set of event types does not, however, limit the expressiveness in our augmented state machines. In addition, to an activation event, each transition is further qualified by a Boolean guard expression that can refer to fields within the header of the arriving packet. Therefore, from one state, several distinct and unambiguous transitions may be defined each triggered by the same event type, using different guard expressions. This approach does complicate the verification of state machine completeness, because we must now determine that the set of Boolean guard expressions for the transitions from a state with the same activation event are both disjoint and exhaustive (that is, exactly one guard expression evaluates to true for the event). The approach does allow, however, one to define the complete behavior with a minimum number of distinct transitions.

2.3.2.2 Action Functions

A transition's action function would typically consist of invoking one of the following framework functions:

3. PktSend (PktComponentMemory)
   Send the packet downwards towards the "network wire" (i.e., the next layer down, or the network). Occurs after the TSM has received a packet from above that needs to be sent and has completed its processing in order to send it. Therefore it can be called only from TSM.
4. PktDeliver ():
   Passes the packet upward towards the "application" (i.e., the next layer up, or the application). Occurs after the RSM has received a packet and has finished processing it. Therefore it can only be called from the RSM.
5. NewPktSend (PktComponentMemory)
   Send a newly created packet downward towards the "network wire." This creates a peer-to-peer message. May be called from either the TSM or the RSM.
6. NewPktDeliver ():
   Sends newly created (or re-constructed) protocol information upward towards the application (e.g., reconstructing a set of fragments). Shall only be called from RSM.

2.3.2.3 Protocol Memory

In order to complete formal analysis of a component it is critical to identify all memory that is accessed by the component. In addition, the scope of each memory accessed must be carefully defined. Memory can be classified into four categories, based on its accessibility and scope:

1. Component-Local Memory

2. Stack-Local Packet Memory
3. Global (external) Memory
4. Packet Memory

The diagram in Figure 35 provides a graphic display of the scope of these memory categories relative to a protocol stack and each protocol component within the stack. Each category is described in more detail in the following sections.

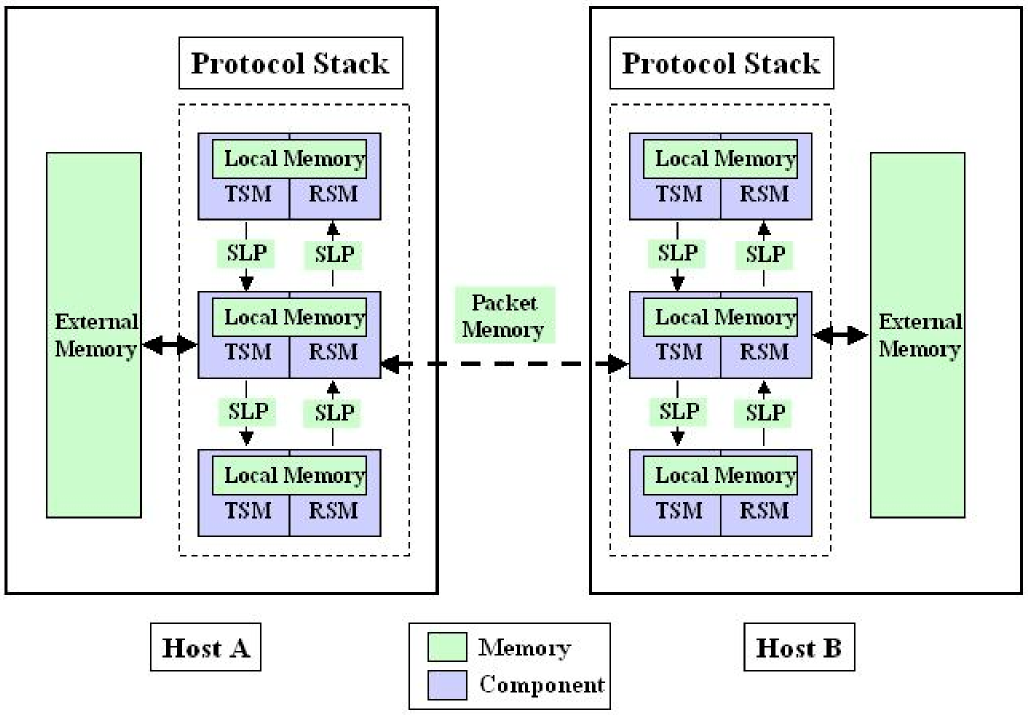


Figure 35 illustrates the different types of protocol memory and where they are located within the protocol stack.

2.3.2.3.1 Component-Local Memory

Component-local memory is internal to a protocol component instantiation. It is accessible only by action functions, the TSM, and RSM of the component. These memories are separately instantiated at the send and receive. If the protocol component is part of a duplex protocol (one which transmits information in both directions between sender and receiver) then the TSM and RSM at each endpoint share the same component-local memory.

The format and content of this memory is unconstrained for each protocol component specification. This memory is instantiated with the protocol component initialization and extends until the protocol component is closed. All access, read and write, is strictly limited to a specific component instantiation. Additional controls for simultaneous memory access are not required for component-local memory, since access to this memory is restricted to the single component for which it was instantiated.

An example of component-local memory is the sliding window buffer in the Reliable Delivery component at the sender to store unacknowledged packets.

2.3.2.3.2 Stack Local Packet Memory

'Stack Local Packet' (SLP) memory provides a mechanism for components within a protocol stack to share information. In our design, components invoke other components by generating events between them. If all protocol components were completely independent there would be no need for this form of communication. Unfortunately, for a wide variety of protocols, complete independence is not possible.

The abstract format of this memory is an association list, or a name-value pairs. The "name" of each pair identifies an element of the SLP memory, and the second element is the value of that element. The actual implementation may choose a more efficient representation. Read access is provided by the framework function,

```
Value getSLP(Event, name).
```

Write access is provided by the framework function,

```
putSLP(Event, name, value).
```

The extent of this memory is limited to the life of a PacketArrivalEvent within a protocol stack. These memories are accessible for read and write access throughout a protocol stack, the collection of protocol components that realize a specific network protocol.

This describes a memory unit of indefinite form with wide scope, which provides great flexibility for inter-component communication. The use of getSLP(event,name) expresses a dependence on another component to provide the named value; but it places no requirement on the presence of a specific component to provide the value, nor a specific call interface for the component with the requirement. These access functions precisely define all inter-component dependencies within a specific protocol stack. We propose static analysis of composed network protocols to ensure that for every entity accessing a value in the composed protocol there exists a value writer. In our initial work we also limit the number of writers to a single component in the protocol stack. This limitation further improves the results from formal analysis, but some protocol stacks may find this limitation too restrictive. Furthermore, if the order of execution of protocol components within a composed protocol is fixed, as in our framework, this static analysis can further verify that the write operation always occurs before any read access.

Additional controls for simultaneous memory access are not required for SLP memory, in spite of its relatively wide scope, since it is attached to a specific instance of a PacketArrivalEvent and at any instant of time is within the scope of a single component.

An example of stack-local packet memory is the source address of a packet which is set by the forwarding protocol component, perhaps mapping from source address to port identifier.

2.3.2.3.3 External (Global) Memory

External memory contains information that is shared among separate protocol stacks. For example, IP routing tables are accessed by routing protocols, such as RIP or OSPF, IP forwarding protocol components, and management and monitoring protocols.

This form of memory has arbitrarily wide scope and extent. This makes it extremely difficult to handle in formal analysis. The format of external memories, access rights, etc. must be agreed to by the designers of all protocols that share the information. Furthermore, any modification in this shared memory by one protocol design may require modifications in all other protocols that share the memory. Details of mechanisms to coordinate and control global memories are outside the scope of this paper.

In our design, global memory access is abstracted through a functional interface for both reading and writing values. The specification of a network component includes the list of external memory functions it utilizes. These functions make explicit the dependencies of a protocol component (and any protocol stack which contains it) on external memory. At the same time, they allow the definition, maintenance and control of the memory to be completely separated from the protocol stacks that utilize it. These functions are responsible for resolving simultaneous access, restricting access rights, etc.

Because the extent of an external memory does not necessarily match the extent of any specific protocol that accesses it, this memory must be maintained by the node environment on which the protocols that accesses it execute. External memory must be instantiated on each node on which one or more of the protocols execute.

An example of an external memory is the IP routing table. An example of an access function, is `nextHopLookup(destAddress).`

2.3.2.3.4 Packet Memory

Finally, a protocol component defines data that it attaches to a packet for transmission to its peer component at the next-hop or endpoint. This memory corresponds to the traditional “packet header.” We describe this as “memory” because it transfers information between non-local entities.

The format and content of this memory is unconstrained for each protocol component specification, much like the component-local memory. This memory is instantiated with the protocol component when it transmits a packet, and extends to the receiving peer protocol component. Access to this memory is strictly limited. It must be written by the component in the transmitting node and read only by the component at the receiving node. All intervening components have an opaque view of this memory as a read-only, linear sequence of bytes. This limited access is required for encapsulation so that security and error detection/correction components are able to perform their functions over the user data as well as the packet memory of other protocol components.

Additional controls for simultaneous memory access are not required for packet memory, since access to this memory is restricted to the single component for which it was instantiated.

An example of packet memory is the sequence number attached to a packet in the Reliable Delivery component.

2.3.2.4 Protocol Component Properties

Protocol components are included in a protocol stack to provide a desired capability, e.g. reliable communication, addressing, or in-order delivery. We use the term ‘Property’ to describe the capabilities a protocol component provides, requires, or maintains. Properties can be used to select components from a library, to verify that a protocol stack provides a needed capability, and/or to determine protocol component inter-dependencies. A component may provide more than one property, but it is usually desired to keep the number of provided properties low to provide finer grain components that allow for additional flexibility.

We use three types of properties. The first type states characteristics of the protocol component. For example, a component might provide an ‘In Order Delivery’ property, or a ‘Reliable Delivery’ property. The second type states characteristics required of other protocol components. For example, before routing a packet the forwarding protocol component may require that the destination address not contain an error. So, it might require a ‘Destination Address Correct’ property be provided by a lower protocol component. The third type of property states invariance of properties. For example, a protocol component that decrypts packets maintains the ‘In Order Delivery’ property.

2.3.2.5 Parameterization

Parameters of a component are the options that can be used to tune the operation of the component. For example, a reliable delivery component may take the maximum send buffer size as a parameter. An increase in the send buffer size allows this component to buffer more messages that have not been acknowledged yet. Parameter values are distributed to each component at initialization time.

2.3.2.6 Identification of Components

There are several possible methods of identifying useful protocol components, but one of the best and simplest may be the decomposition of current protocols. We spent a great deal of effort trying to extract components that made up current protocols in hopes that we would be able to eventually simulate the original. Some of the components we considered were checksum, reliable delivery, in-order delivery, windowing, and fragmentation.

2.3.3 Composition of Protocol Components

Protocol components communicate with peers across a communications channel. The manner in which the packet memory of one protocol component is collected with the packet memories of other components is called composition and defined by a composition operator. The most common composition operator in network protocol definitions is encapsulation. The encapsulation operator treats the content of upper protocol layers as a single, undefined entity and carries it as a payload. A typical composition sequence is user messages are encapsulated in TCP packets, which are encapsulated in IP packets, which are encapsulated in Ethernet packets. While encapsulation is a powerful abstraction and implementation technique, there are other composition operators and sometimes strict encapsulation is inconvenient or violated (e.g. TCP checksum assuming certain values for IP header fields).

Besides the encapsulation composition operator, there are at least two other possible operators. The first, and simplest, is simple linear accumulation of protocol component packet memory. That is, one defines the protocol stack as a sequence of protocol components. Each protocol component from top to bottom on the transmit side adds its packet memory (packet fields) to the beginning of the packet. On the receiver side packet memory is removed from the incoming packet in the reverse order and either used to implement the associated protocol component or recorded in the stack local memory for other components to reference. Thus packet memory acts as a “last added, first used” data structure.

A third composition operator we call Type-Length-Value. Using this operator, each element of the packet memory is tagged with its type or attribute. For example, one might tag the destination address with ‘DestinationAddr’ or the time-to-live value with ‘TTL’. While adding the attribute field expands the packet size, this approach can facilitate some arbitrary ordering of packet processing and even parallel processing (execution) of protocol components.

For our initial implementation of composite protocols we use the simple, linear composition of packet memory elements and process packets in the “last added, first used” manner.

2.3.4 Related Work

The idea of composing protocols from modular components has a long history. Indeed, basic courses in networking introduce network concepts based on the OSI seven layer model and network protocols are designed to work with different lower and upper layers.

The X-Kernel [Hutchinson1991] is an operating system kernel that provides an architecture for constructing, implementing and composing network protocols. The key idea behind X-Kernel architecture is to split the traditional protocol stack, which has a simple linear topology and complex per-node functionality, into a complex protocol graph consisting of individual protocols called micro-protocols and virtual protocols.

The Cactus system [Wong2001] has a two-level model. Protocol components, termed micro-protocols, are combined together with a runtime system to form a composite protocol. A composite protocol is composed with other protocols in a normal hierarchical manner (using X-Kernel) to form a network subsystem.

Ensemble [Hayden1998] is a group communications system designed for constructing a variety of distributed applications from a set of reusable components. It builds upon the Horus and Isis systems [vanRenesse1995] and is written in Objective Caml (Ocaml) [Leroy2002], a dialect of the functional language ML. In Ensemble, a protocol component is called a “layer”; and a set of layers is combined into a “protocol stack.” The composition methodology used in Ensemble is simple and regular. Each layer is stacked on the single layer immediately below it; and communication between these two layers is handled by two FIFO queues, one for information passing from the lower layer to the one above it; and the other for information passing from the upper layer to the one below it. These queues are implemented as unbounded in length.

CHANNELS [Boecking1995] is a run-time system for network protocol processing. It enables dynamic configuration of protocols from a set of protocol components. It provides facilities like buffers, timers and mapping tables that simplify common protocol operations. It also provides a standard framework for inter-component communication. CHANNELS is implemented in the C++ programming language and is used in TIP (Transport and Internetworking Package). CogPiT (Configuration of Protocols in TIP) defines a dynamic configuration methodology for protocol components (modules) [Stiller1995]. The configuration algorithm (COFAL) selects appropriate protocol components out of a pool of predefined components and generates a 'channel template’. The channel template describes the list of selected components and the interconnection among components. The CHANNELS runtime system uses the channel template to create instances of selected protocol components and setup an application-tailored protocol.

Netscript [daSilva1998A, daSilva1998B] is a coordination language and environment for developing composite protocols for active networks. It uses the dataflow-programming model to structure the software on an active node. An active node is visualized as a processor of packet streams. It consists of one or more Netscript engines. A Netscript engine is a set of computational elements called boxes, which are interconnected through their input and output ports. A box consists of sequential event-handling code. When a message arrives at an input port of a box, the event handler associated with the input port is executed. Typically, this processing leads to the message being sent out of one (or more) of the output ports of the box. The computation in a box is reactive to the arrival of messages. There is no global state or state shared between boxes, so different boxes may execute concurrently. Messages flow from box to box as a Netscript engine processes them. New boxes can be added or removed dynamically from a Netscript engine.

2.3.5 Conclusion

We have defined a formal approach to defining protocols in modular units called protocol components. Each component is simple, well defined, and suitable for automatic (machine) analysis. We have defined a simple composition operator, which is also suitable for automatic analysis. By formalizing the definition of protocol components and their interaction, we anticipate that more robust, reliable, and secure protocols will be defined and deployed.

2.4 Design of Composite a Multicast Service

This section describes the various steps involved in building a composite 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[Lin1996] 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[Hedrick1988], OSPF[Moy1994], 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 sub-set 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 composite service using our framework.

2.4.1 Steps in building a composite service

2.4.1.1 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.

2.4.1.2 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 [Minden2002]. 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 in Section 2.4.6 and following. 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.

2.4.1.3 Building the stacks

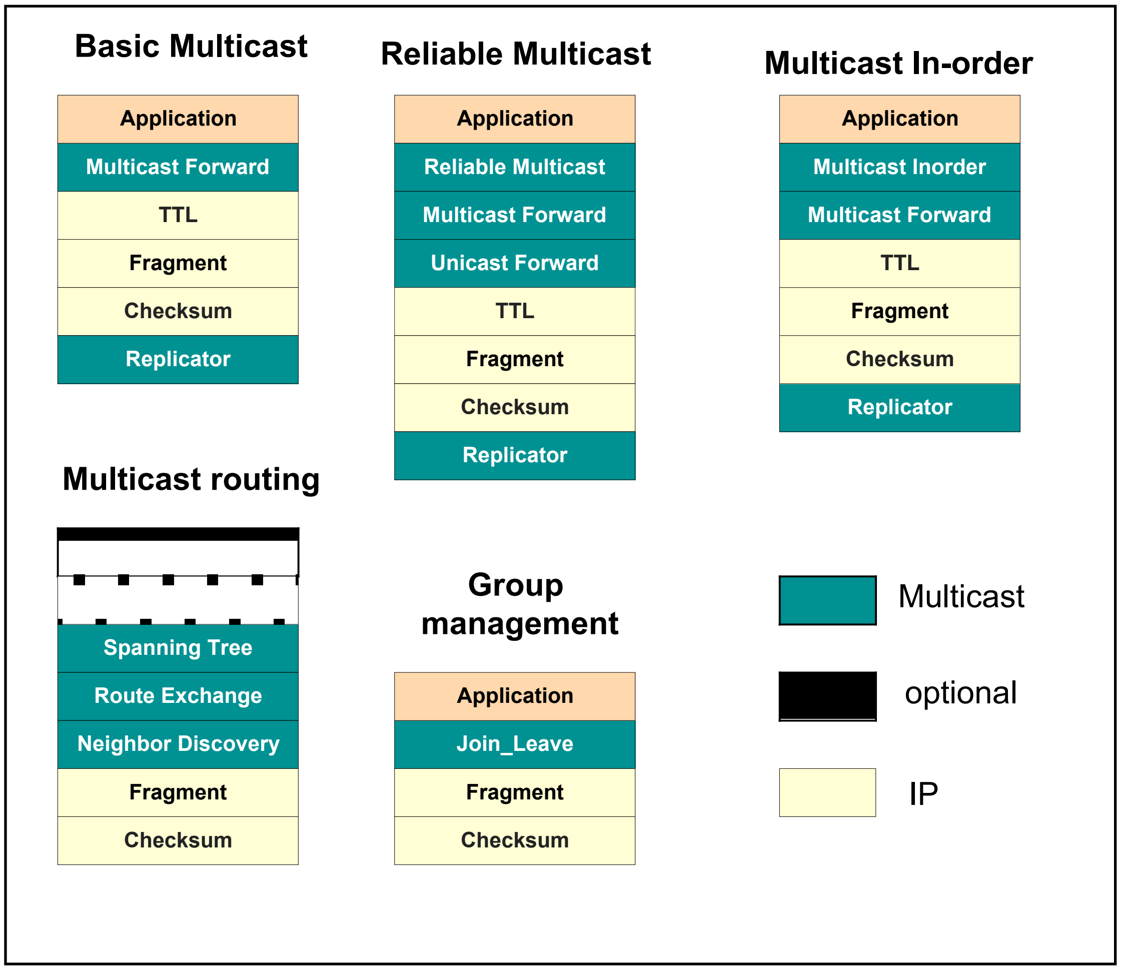


Figure 36 Multicast Service Stacks

Once all the individual protocol components are specified, related components are grouped in protocol stacks called composite protocol stacks. The composite service, Figure 36, 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 may 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.

2.4.2 Deployment - Placing the stacks in the network

We focus 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 report, the composite protocol stacks are manually deployed on normal nodes (non-Active nodes).

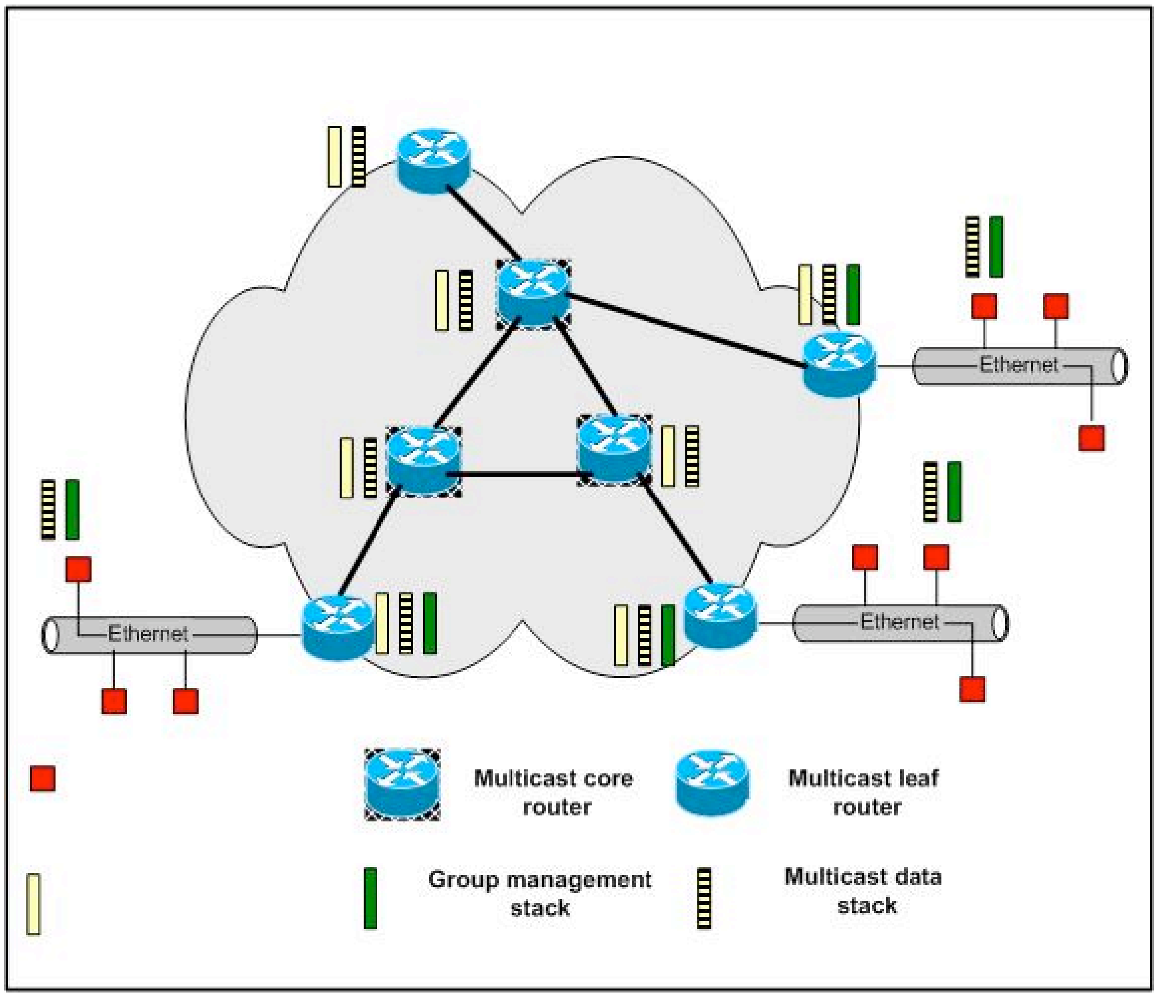


Figure 37 illustrates a multicast network and the deployment of services and protocol stacks.

Figure 37 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

2.4.3 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.

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

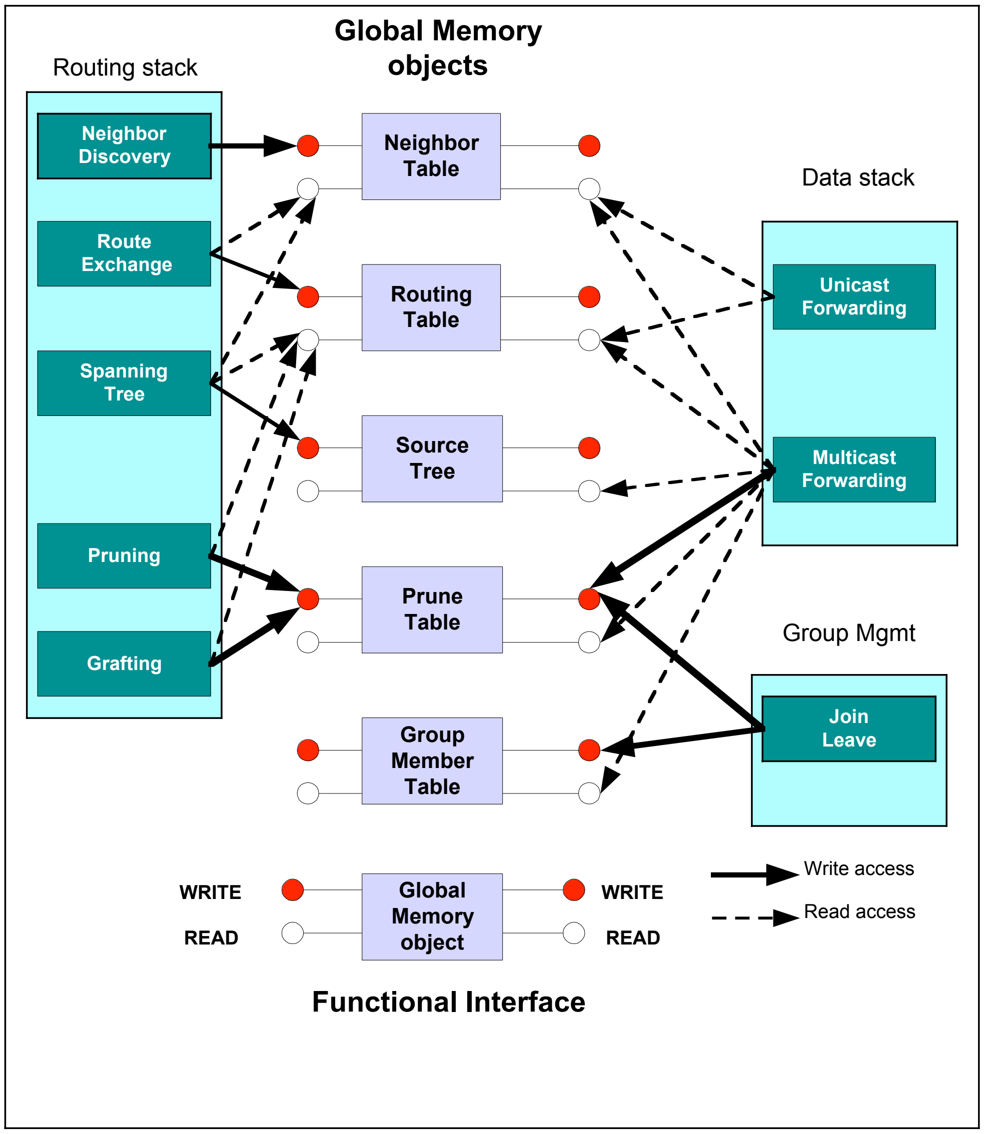


Figure 38 Multicast service global memory objects.

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.

2.4.4 Global Memory features:

2.4.4.1 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.

2.4.4.2 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.

2.4.4.3 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.

2.4.4.4 Implementing global memory:

We now discuss a few approaches to implement global memory.

2.4.4.4.1 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.

2.4.4.4.2 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.

2.4.4.4.3 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.

2.4.5 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 38 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.

2.4.5.1 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.

2.4.6 IMPLEMENTATION

The multicast system described in this section is built on the IANS framework described in Section 0. Section 0 provides a top level description of protocol components. Details of our implementation are below.

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.

Ensemble was selected for the following reasons:

- Ensemble is written in Ocaml[Leroy2002], a functional programming language, and dialect of ML[Appel1991]. 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.

2.4.6.1.1 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 39. 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.

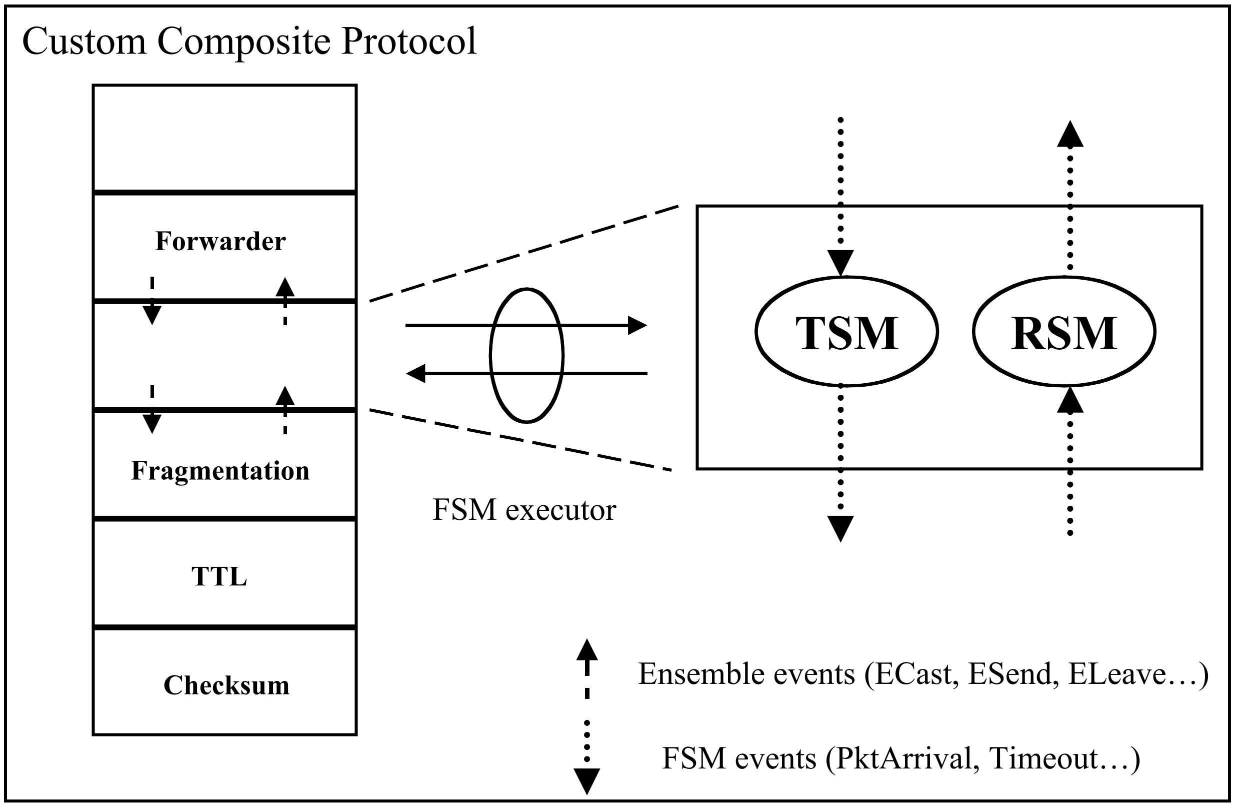

Custom Composite Protocol
Forwarder
Fragmentation
TTL
Checksum
FSM executor
TSM
RSM
Ensemble events (ECast, ESend, ELeave…)
FSM events (PktArrival, Timeout…)

Figure 39 illustrates the 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 e.g. 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.

2.4.6.2 Mapping of framework functions

The Table 7 shows the mapping between few of our framework functions and Ensemble UP and DN events

Framework Functions	***Ensemble Event***
Packet Transfer	
pkt_send(pktmem)	*DN (EV, ABV, hdr)*
new_pkt_send(pktmem)	*DNLM (ev, hdr)*
pkt_deliver()	*UP (EV, ABV)*

new_pkt_deliver(pktpayld)	*UP (ev, pktpayld)*
Buffer Management	
send_kept_packet(pktpayld)	*DN (ev, hdr, pktpayld)*
deliver_kept_packet(pktpayld)	*UP (ev, pktpayld)*

Table 7 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.

2.4.6.3 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 up to 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.

2.4.6.4 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.

2.4.7 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.

Shared memory is another method of inter-process communication (IPC) whereby two 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.

Details of setting up and using shared memory in the Linux operating system are described in [ITTC-F2004-TR-19740-11].

2.4.7.1 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.

2.4.8 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.

2.4.8.1 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	*void write_ntable([in] struct ntable_entry ntable[], [in] int num);*

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

Table 8 Neighbor Table- Functional Interface.

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

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

Table 9 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 }
```

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

Table 10 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;
};
```

2.4.8.4 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	*void pruneIGMPIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);*
	Add/update prune table entry for *(src_addr,grp_addr)* pruning igmp interface *intf_ipaddr*
	void pruneIGMPIntfforGrp([in,int64] int grp_addr, [in] int intf_ipaddr);
	Add/update prune table entry for (all src_addr's,grp_addr) pruning igmp interface intf_ipaddr
	void pruneCoreIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);
	Add/update prune table entry for *(src_addr,grp_addr)* pruning core interface *intf_ipaddr*
	void graftCoreIntfforSrcGrp([in,int32] int src_addr, [in,int32] int grp_addr, [in,int32] int intf_ipaddr);
	Add/update prune table entry for *(src_addr,grp_addr)* grafting core interface *intf_ipaddr*
	void graftIGMPIntfforGrp([in,int32] int grp_addr, [in,int32] int intf_ipaddr);
	Add/update prune table entry for *(src_addr,grp_addr)* grafting the igmp interface i*ntf_ipaddr*
READ	*int get_no_of_entries();*
	returns the number of entries present in the Prune Table.
	struct prunetable_entry getentry([in] int n);
	returns the nth entry from the Prune Table
	struct prunetable_entry getentryForSrcGrp([in,int32] int src, [in,int32] int grp_addr);
	returns the Prune Table entry corresponding to the (source,group) pair *(src,grp_addr)*

Table 11 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
};
```

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

Table 12 Group Table – Functional Interface

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

2.4.9 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 [Pusateri2000] 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.

2.4.10 Component Implementation

In this section, we describe the three protocol stacks needed to implement the multicast service; the multicast data stack, the multicast routing stack, and the group join/leave stack; and list each protocol component used in the multicast service. For complete technical details see [ITTC-F2004-TR-19740-11].

2.4.10.1 Multicast Data Stack

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.

2.4.10.1.1 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.

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.

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.

2.4.10.1.2 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).

2.4.10.1.3 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 (Tdally) , retrans_timer(Tretrans)* and nack*_timer(Tnack)*.

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. *Tdally* 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 *Tdally*. 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.

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

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

2.4.10.1.4 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 it 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.*

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

2.4.10.2 Multicast Routing Stack

2.4.10.2.1 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.

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*.

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.

2.4.10.2.2 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.

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*.

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

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.

2.4.10.2.3 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.

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*.

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.

2.4.10.2.4 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 S*panning 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*.

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.

2.4.10.2.5 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.

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.

2.4.10.3 Group Membership Stack Components

2.4.10.3.1 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 q*uery 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 r*outer_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.

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. g*roup_address in Report packets* refers to the group address being reported.

2.4.10.4 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 throughout, 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 2.4.10.4.1 describes the functionality test and Section 0 describes the various performance measurement tests that were performed using composite protocol stacks.

2.4.10.4.1 Functionality Testing

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

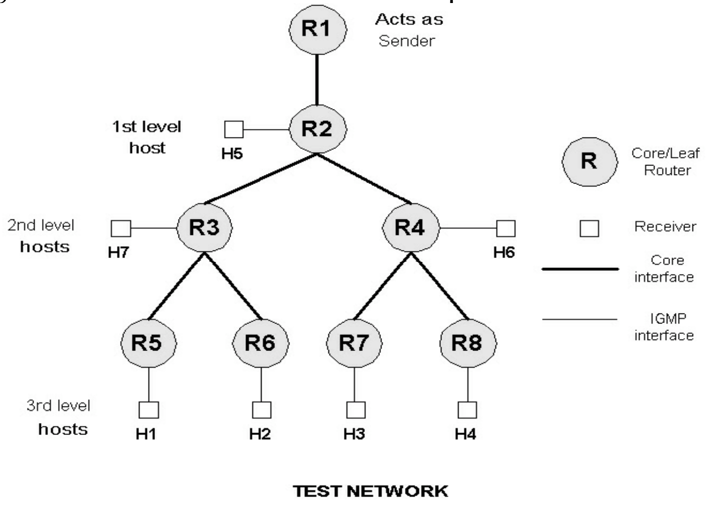


Figure 40 Multicast Test Network

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.

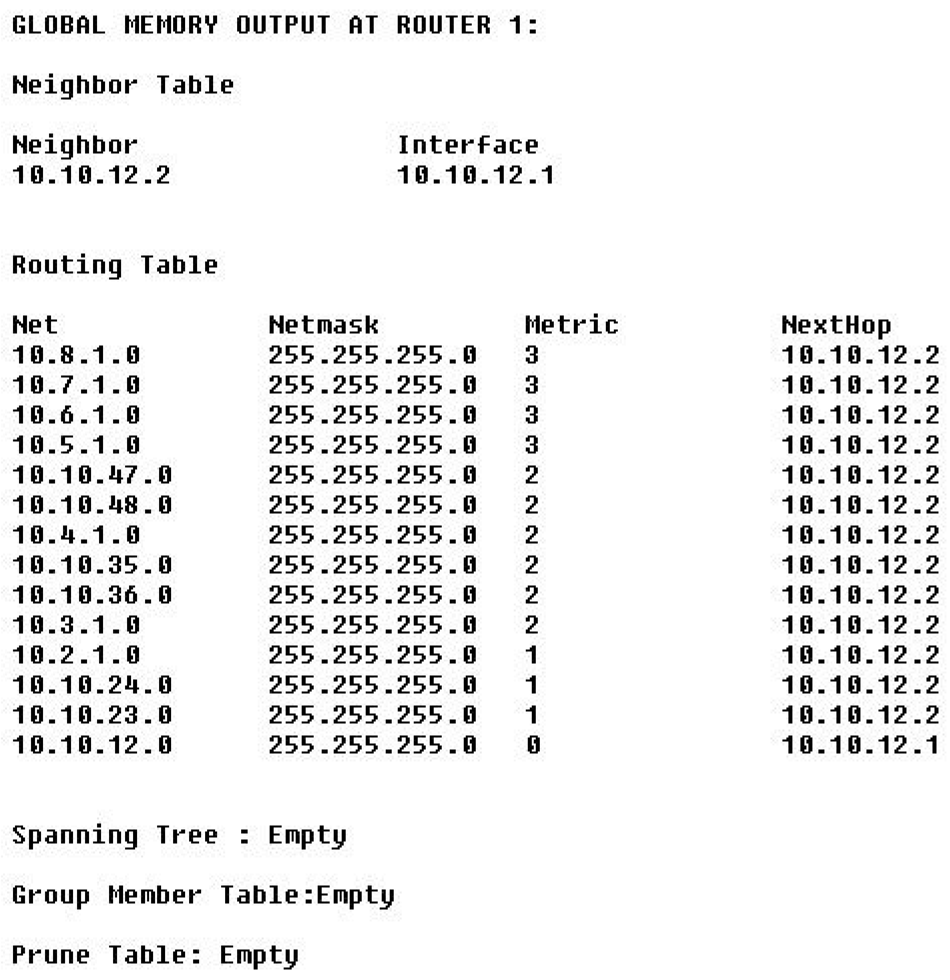
Running the Multicast Routing stack:

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

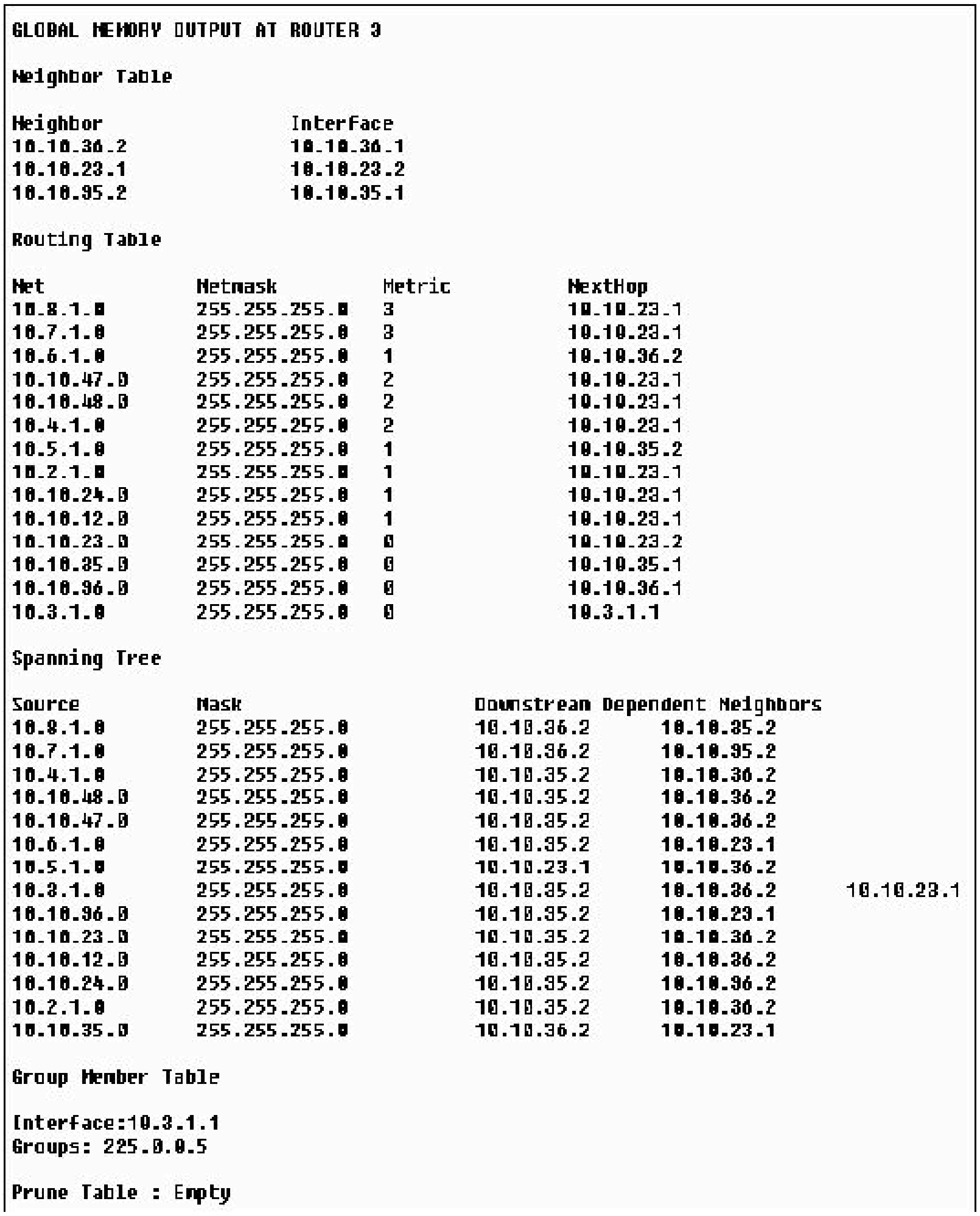
The stack ordering from top to bottom is *pruning, grafting, poison_reverse, route exchange, neighbor_discovery* over the default *checksum* component. The component that generates packets most frequently is kept at the bottom-most. So *neighbor_discovery* was placed at the bottom and *grafting* was 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.*

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 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 has pruning has not started.



```
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
```

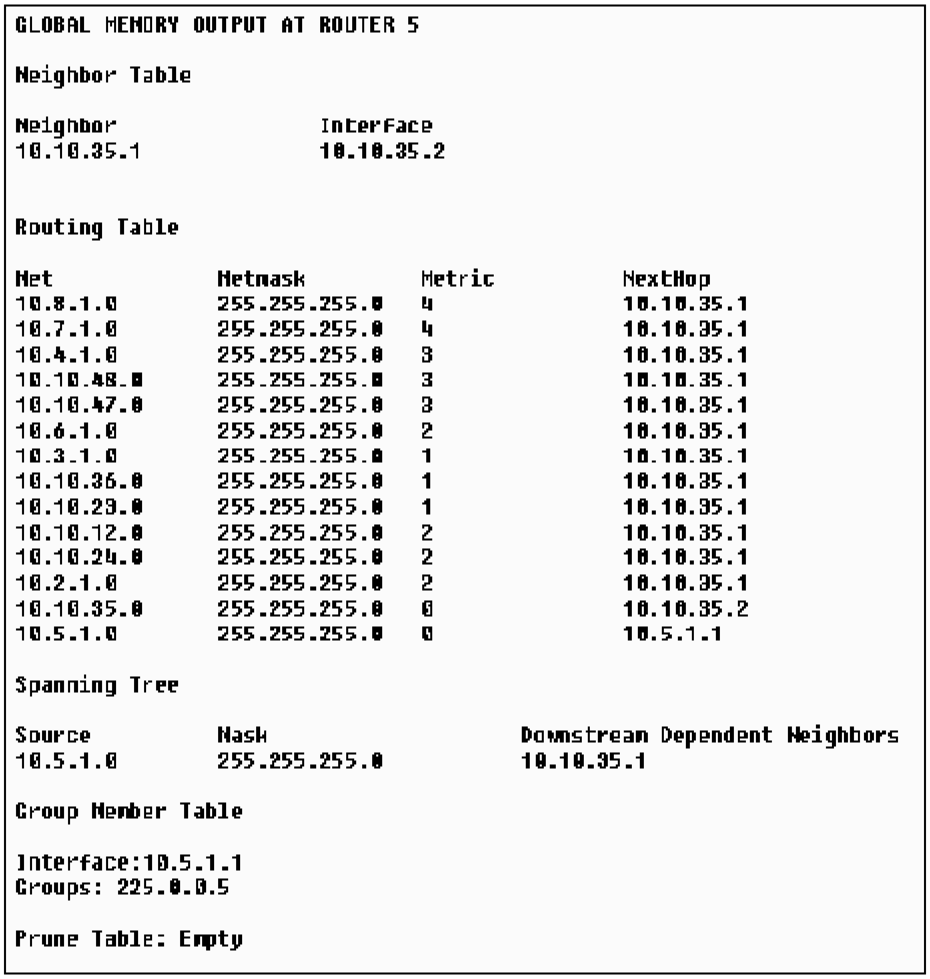
Figure 41 lists the routing table at Router 1



```
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
```

Figure 42 lists the routing table at Router 3

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
```

Figure 43 lists the routing table at Router 5

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.

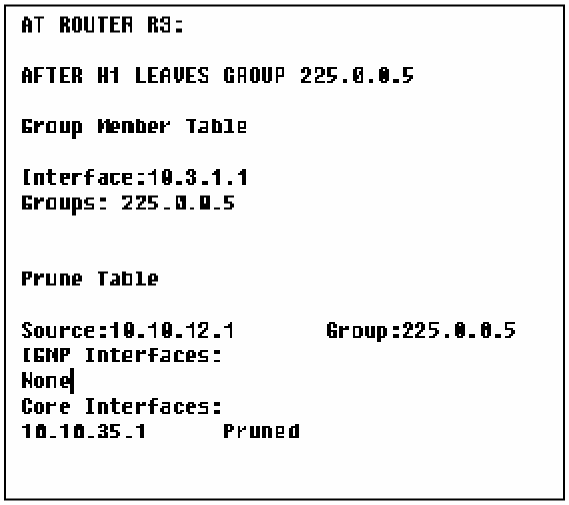
Initial State: We have an un-pruned tree rooted at the source R1 as shown in the figure 6. All the hosts have joined the group 225.0.0.5 and have started receiving data from the source.

Event A: H1 leaves group 225.0.0.5.

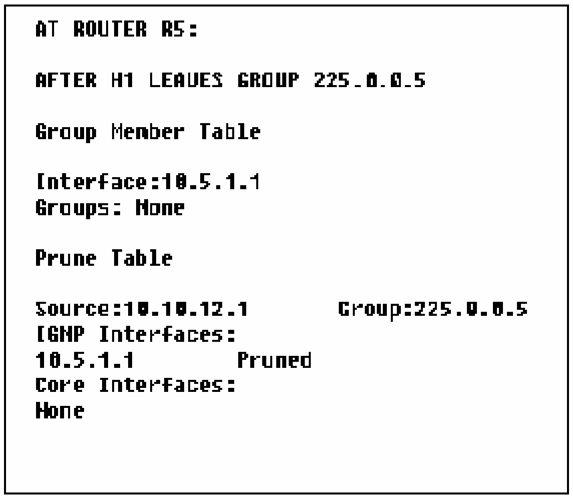
Observation: We observe changes in global memory at routers R5 and R3. We show group table and prune table contents only, as contents of other tables are not expected to change due to group joins and leaves. At router R5, the leaf interface 10.5.1.1 connecting

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. 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.



```
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
```



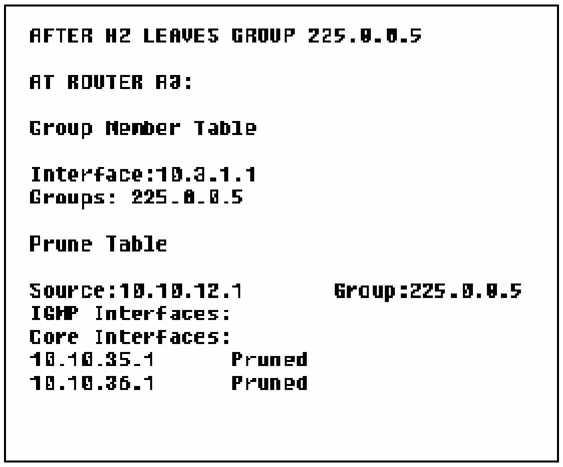
```
AT ROUTER R5:

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
```

Figure 44 shows the pruning action after H1 leaves the group

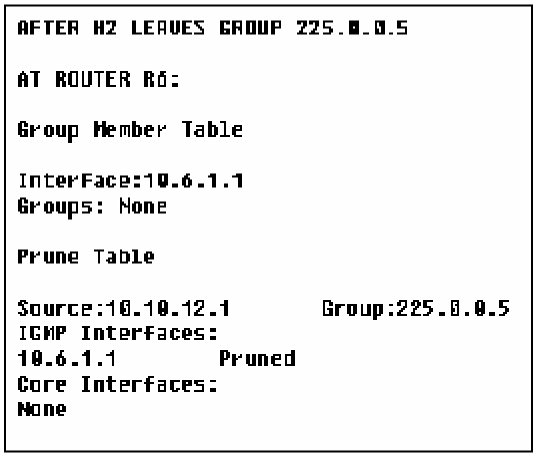
Event B: H2 leaves group 225.0.0.5

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

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.



```
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
```



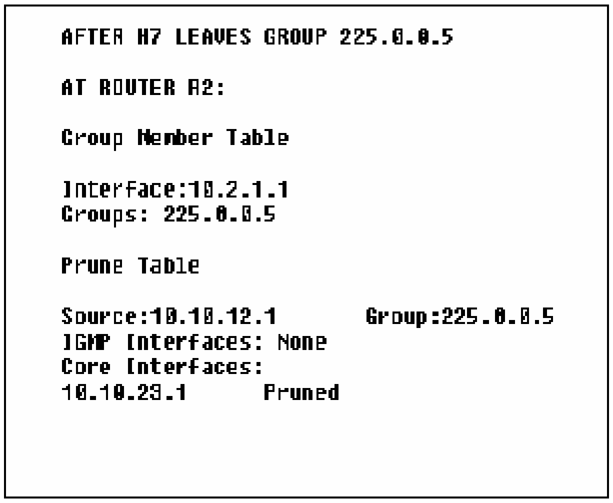
```
AFTER H2 LEAVES GROUP 225.0.0.5

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
```

Figure 45 shows the pruning action after H2 leaves the group

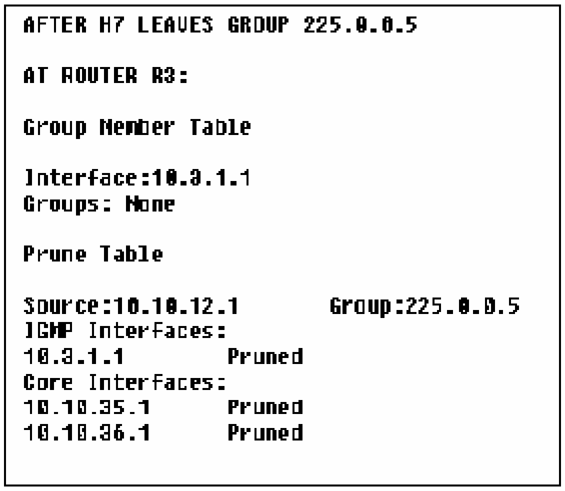
Event C: H7 also leaves the group 225.0.0.5.

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

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.



```
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
```



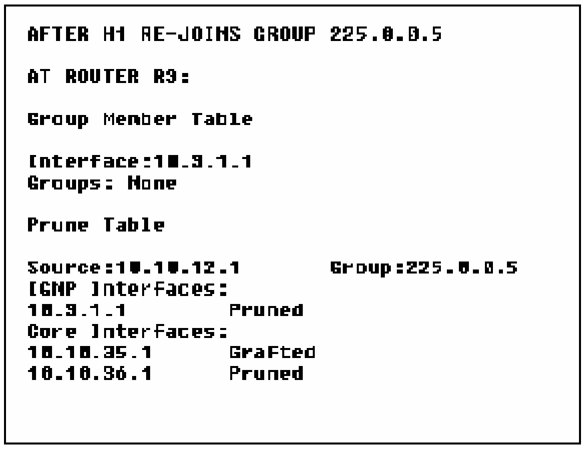
```
AFTER H7 LEAVES GROUP 225.0.0.5

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
```

Figure 46 shows the pruning action after H7 leaves the group

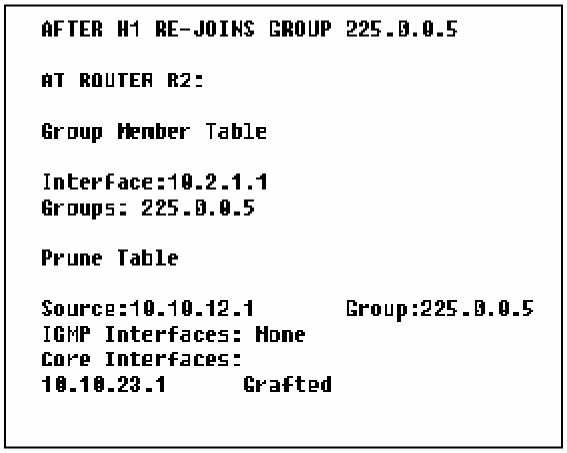
Event D: H1 re-joins the group 225.0.0.5

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

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



```
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
```

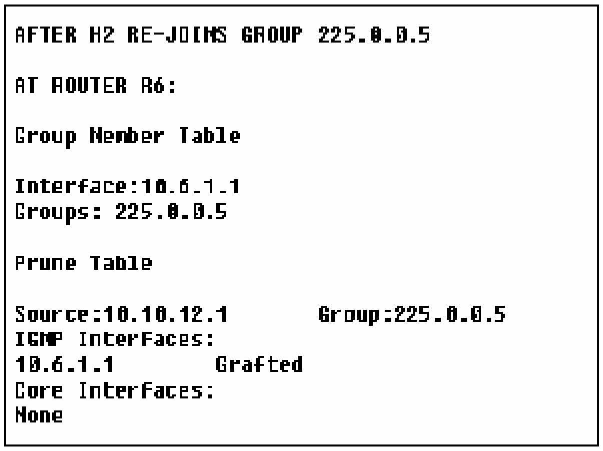
Figure 47 shows the joining action after H1 re-joins the group

Event E: H2 re-joins the group 225.0.0.5

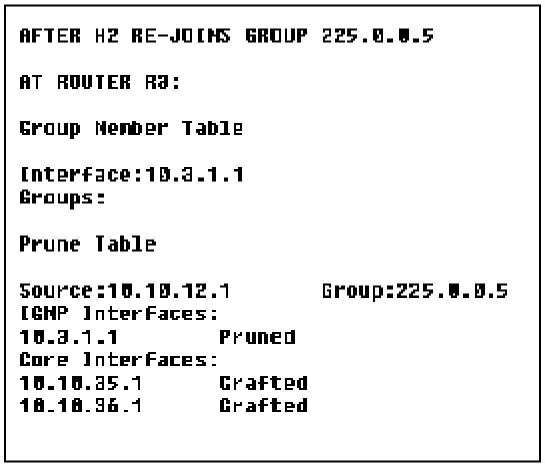
Observation: We observe the effect of join on routers R6 and R3.

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.



```
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
```

Figure 48 shows the joining action after H2 re-joins the group

2.4.10.4.2 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□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.

2.4.10.4.2.1 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.

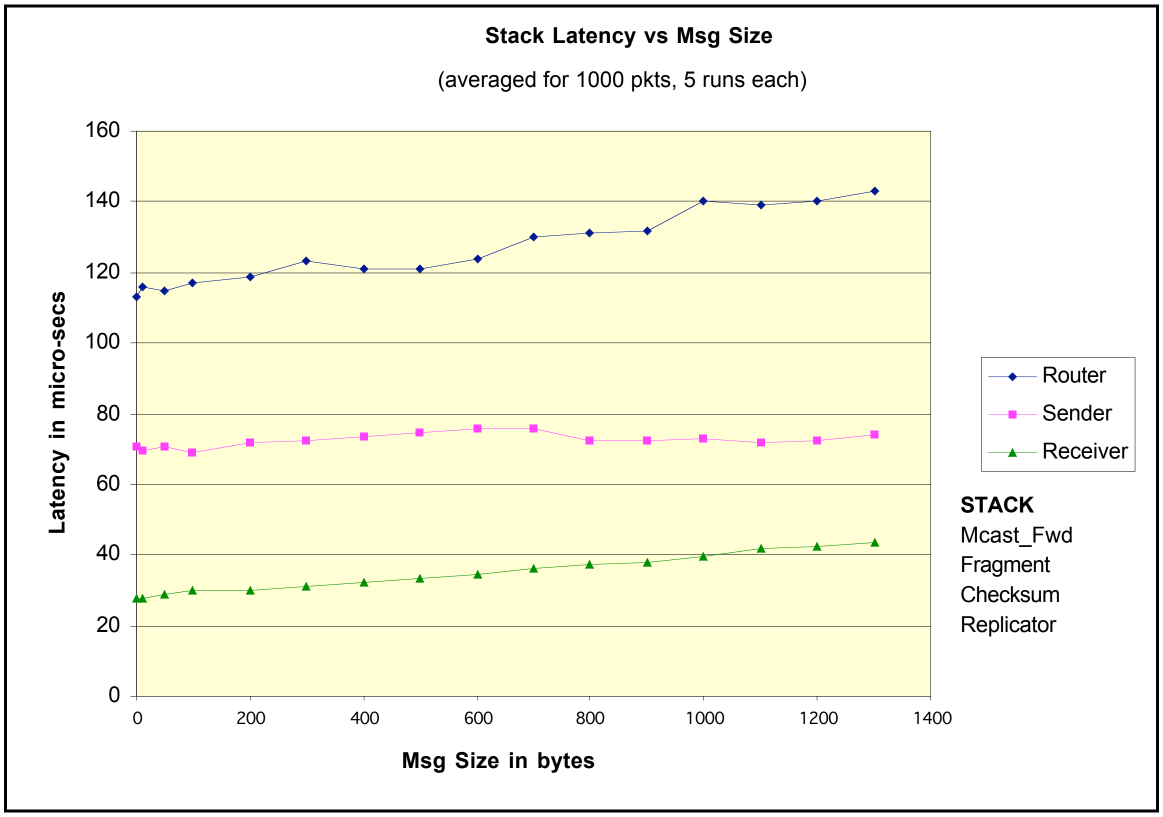


Figure 49 Variation of Stack Latency with Message Size

The Figure 49 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 113s for 1 byte message to 143s for a 1300-byte message. At the receiver it increases from 27.8s to 43.5s. 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 13 Variation of Stack Latency with Message Size

2.4.10.4.2.2 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 14 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 15 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

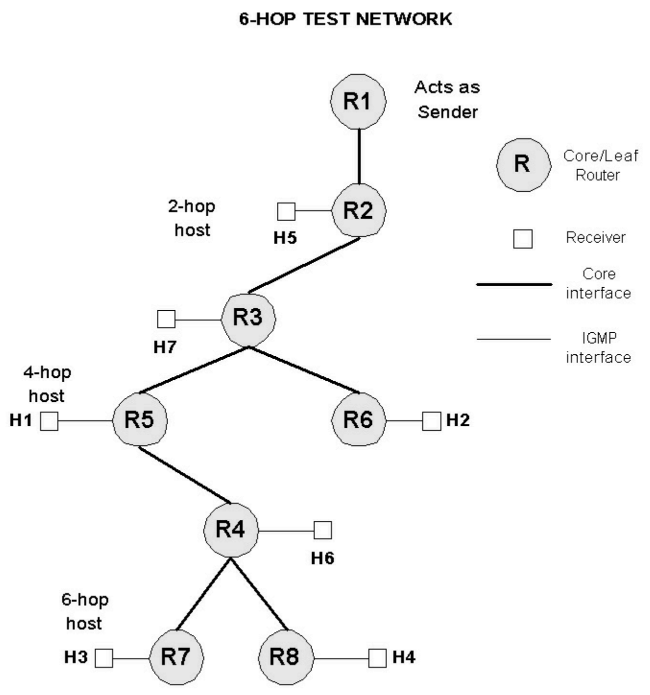


Figure 50 Multicast Hop Test Network

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[Mills1988] 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 up to 6 network hops.

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

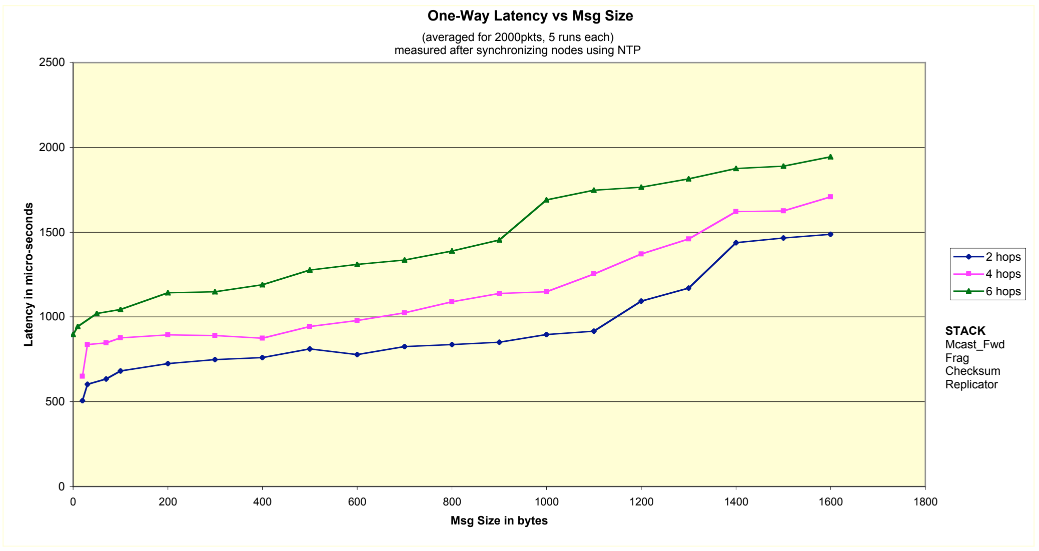


Figure 51 Multicast Hop Test Network

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 16 Variation of end-to-end latency with number of 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.

2.4.10.4.2.3 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}) seconds, 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.

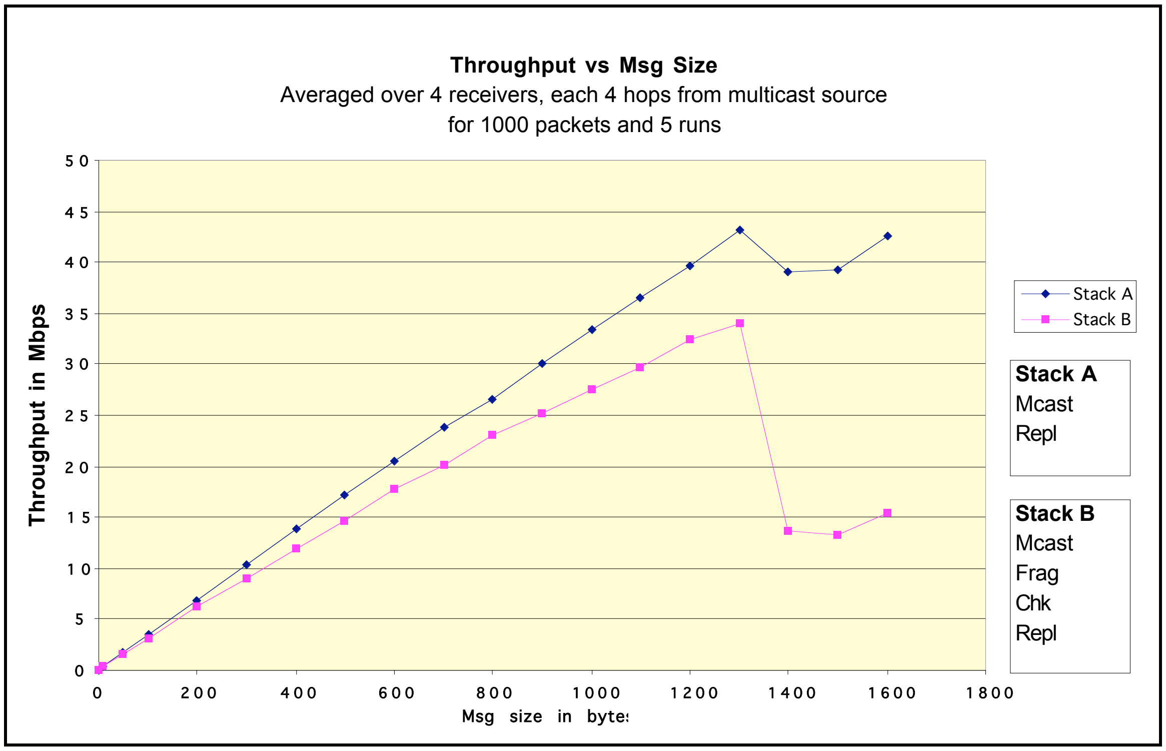


Figure 52 Variation of throughput with message size

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 17 Lists variation of throughput with message size.

2.4.10.4.2.4 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 dally 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.

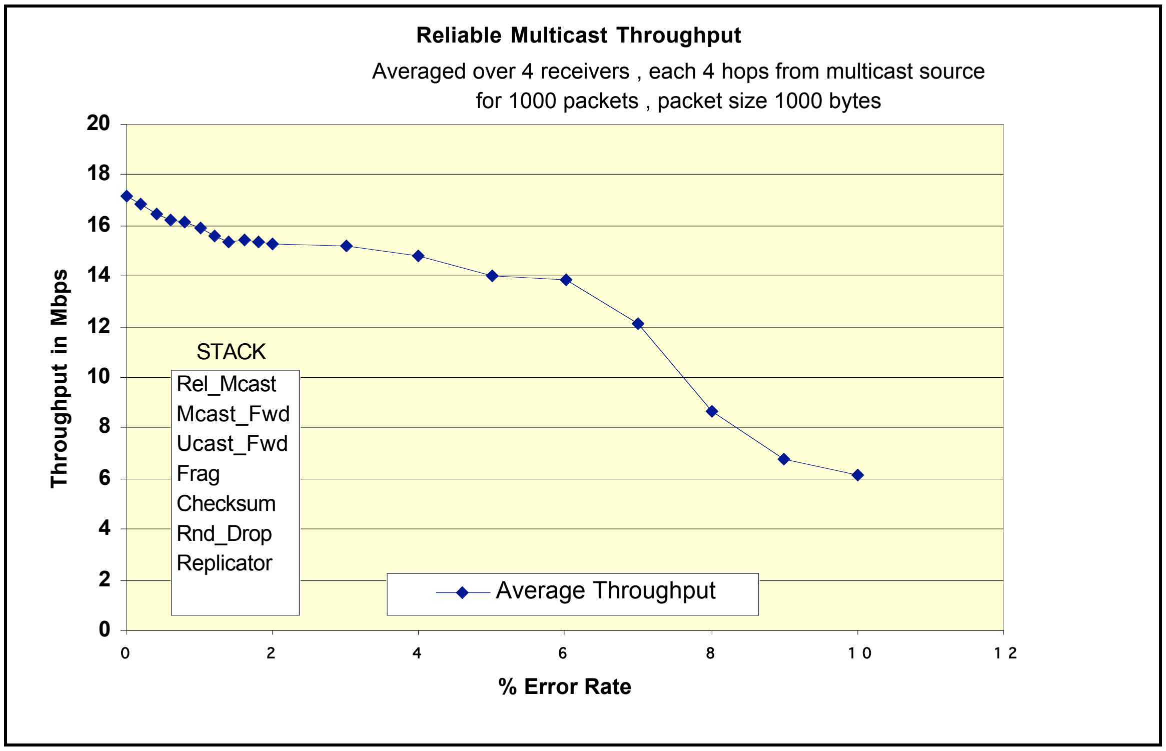


Figure 53 Variation of reliable multicast throughput with error rate.

For a 0% link-error probability (e), 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 probablity 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 18 Variation of reliable multicast throughput with error rate

2.4.10.4.2.5 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.

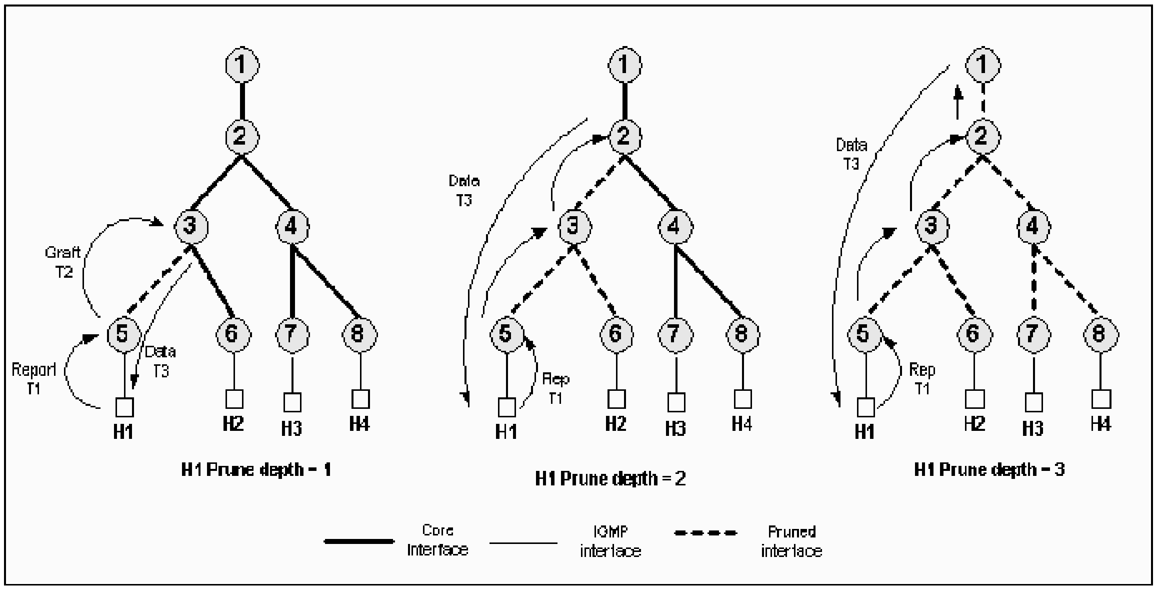


Figure 54 Prune depth of a multicast tree.

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

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

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

Table 19 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.

2.4.10.4.3 Comparison with Linux IP Multicast

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

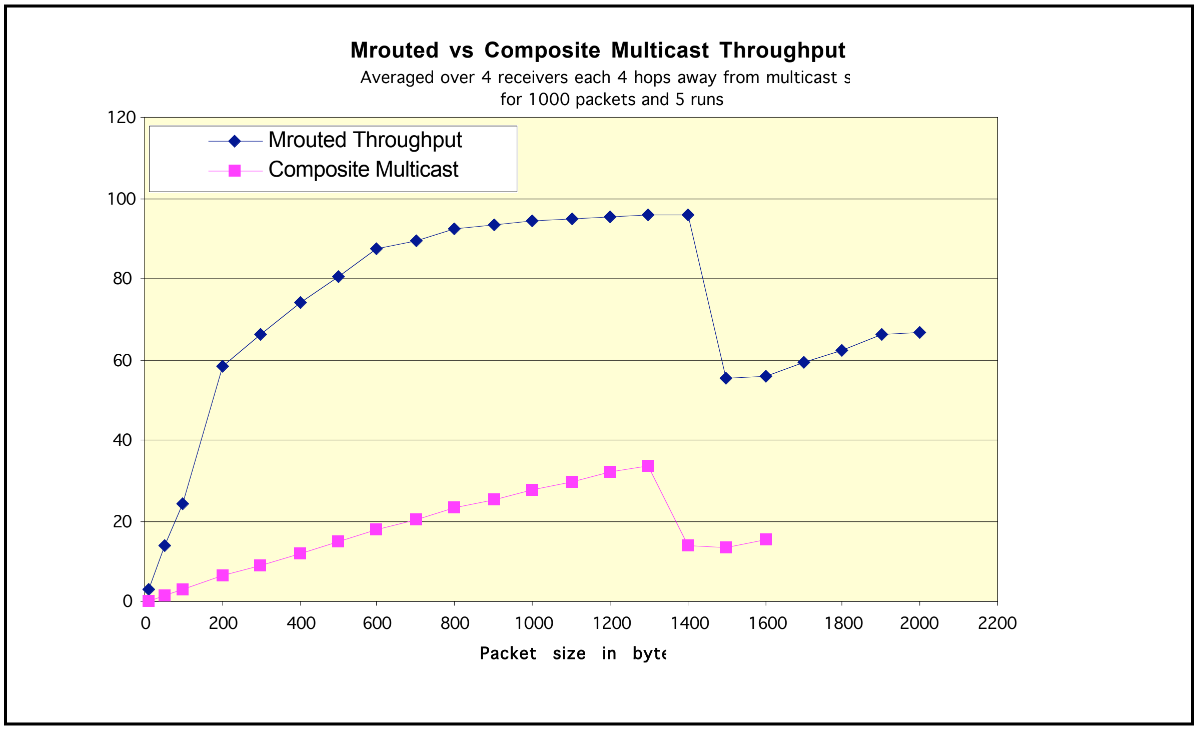


Figure 55 Comparison with Linux IP multicast throughput

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	Throughput (in Mbps)	
(in bytes)	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 20 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.

2.4.11 Conclusions and Future work of Multi-cast

This work 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-mode test network. The functionality tests confirm the expected behavior 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.

This section suggest possible improvements and enhancements to this work 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 composite 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 composite 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.

3 Project Information

3.1 Budget Summary

The IANS project ran from June 4, 1999 through May 31, 2003. The budget was $1,463,940 total, $1,382,750 Federal and $81,190 The University of Kansas. Table 2 shows the actual and cumulative federal expenses on a month-by-month basis. Contributions of The University of Kansas consisted of faculty salary match during the academic year.

Month	Actual Costs	Actual Cum.		Month	Actual Costs	Actual Cum.
Jun-99	0.00	2,307.08		Jun-01	29,402.17	688,696.35
Jul-99	1,086.37	3,393.45		Jul-01	89,544.34	778,240.69
Aug-99	3,259.11	6,652.56		Aug-01	33,752.13	811,992.82
Sep-99	12,282.56	18,935.12		Sep-01	15,227.06	827,219.88
Oct-99	20,970.05	39,905.17		Oct-01	18,713.42	845,933.30
Nov-99	15,932.05	55,837.22		Nov-01	25,129.29	871,062.59
Dec-99	32,768.53	88,605.75		Dec-01	12,436.62	883,499.21
Jan-00	14,470.40	103,076.15		Jan-02	15,977.86	899,477.07
Feb-00	11,563.93	114,640.08		Feb-02	16,707.29	916,184.36
Mar-00	16,370.53	131,010.61		Mar-02	15,754.48	931,938.84
Apr-00	32,144.30	163,154.91		Apr-02	17,125.60	949,064.44
May-00	17,171.17	180,326.08		May-02	28,894.11	977,958.55
Jun-00	43,897.64	224,223.72		Jun-02	20,144.73	998,103.28
Jul-00	72,604.21	296,827.93		Jul-02	60,990.71	1,059,093.99
Aug-00	58,594.50	355,422.43		Aug-02	66,826.82	1,125,920.81
Sep-00	21,334.42	376,756.85		Sep-02	31,183.05	1,157,103.86
Oct-00	30,587.35	407,344.20		Oct-02	50,792.93	1,207,896.79
Nov-00	50,085.10	457,429.30		Nov-02	34,033.45	1,241,930.24
Dec-00	30,351.50	487,780.80		Dec-02	38,695.65	1,280,625.89
Jan-01	28,692.61	516,473.41		Jan-03	30,292.98	1,310,918.87
Feb-01	32,172.85	548,646.26		Feb-03	25,344.16	1,336,263.03
Mar-01	27,630.68	576,276.94		Mar-03	11,395.78	1,347,658.81
Apr-01	27,205.73	603,482.67		Apr-03	156.22	1,347,815.03
May-01	55,811.51	659,294.18		May-03	9,442.58	1,357,257.61
				Jun-03	19,428.79	1,376,686.40

Table 21 lists the actual monthly and cumulative federal project costs.

3.2 Project Personnel

Professors Gary J. Minden and Joseph B. Evans directed the project. Research Engineer Ed Komp helped organize the project and direct the graduate students. Four graduate students and four undergraduate students worked on this project:

Ravi Chamarty, Vishal Zinjuvadia, Suresh Krishnaswamy, Yoganandhini Janarthanan, Steve Gange, Disha Chopra, Magesh Kannan, Sandeep Subramaniam, Srujana Vallabhaneni, and Shyang Tan.

3.3 Project Equipment

The IANS project obtained general purpose computers to develop software and an array of 12 computers to test and evaluate composite protocols.

4 Conclusion

The IANS project successfully implemented an active networking system. Our initial ideas for a simple NodeOS and execution environment did not prove successful when implemented. However, our work on composite protocols showed a mechanism to implement standard protocols by composing functional components and the ability to combine protocol components and protocol stacks into active services. We intend to continue this work in the future.

References

[Alexander1998] D. S. Alexander, W. A. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. T. Moore, C. A. Gunter, and J. M. Smith, "The Switchware Active Network Architecture," IEEE Network Magazine, 1998.

[Appel1991] 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.

[Boecking1995] Stefan Boecking, Vera Seidel, Per Vindeby, "CHANNELS: A Run-Time System for Multimedia Protocols," ICCCN 1995, Las Vegas, NV, September 1995.

[Burke1996] Garrett Burke, "KANGA: A Framework for Building Application Specific Communication Protocols", Master of Science Thesis, University of Dublin, September 1996.

[Burke1996] Garrett Burke. KANGA: A framework for building application specific communication protocols. M. S Thesis, Department of Computer Science, University of Dublin, September 1996.

[daSilva1998A] Sushil da Silva, Danilo Florissi, Yechiam Yemini, "Composing Active Services in Netscript," Position Paper, DARPA Active Networks Workshop, Tucson, AZ, March 1998.

[daSilva1998B] Sushil da Silva, "Programming in the NetScript Toolkit," http://www.cs.columbia.edu/~dasilva/pubs/netscript-0.10/doc/tutorial.html, September 1998,

[Deering] S. Deering, D. Estrin, V. Jacobson et al, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Motivation and Architecture" draft-ietf-idmr-pim-arch-01.ps , Internet Draft.

[Fenner] B. Fenner. "The multicast router daemon - mrouted," ftp://ftp.parc.xerox.com/pub/net-research/ipmulti.

[Fenner1997] W. Fenner, "Internet Group Management Protocol, Version 2", RFC 2236, Xerox PARC, November 1997.

[Ford1997] B. Ford, G. Back, G. Benson, J. Lepereau, A. Lin, O. Shivers, "The Flux OSKit: A Substrate for kernel and Language Research," Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[Freeswan] http://www.freeswan.org, Version 1.8

[Gurevich2000] Yuri Gurevich, “Sequential Abstract State Machines Capture Sequential Algorithms,”ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.

[Hayden1998] Mark Hayden, The Ensemble System, Ph.D. Dissertation, Cornell Computer Science Department, January 1998.

[Hedrick1988] C.Hedrick. Routing Information Protocol. RFC 1058, June 1988.

[Hicks1998] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, “PLAN: A packet language for Active Networks,” Proceeding of the Third ACM SIGPLAN International Conference on Functional Programming Languages, 1998, pp. 86-93, ACM.

[Hutchinson1991] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols, IEEE Transactions on Software Engineering, 17(1), pp. 64-76, Jan. 1991.

[ISO1994] ISO, "Information Processing Systems - OSI Reference Model - The Basic Model", ISO/IEC 7498-1, 1994.

[Kulkarni1998] Amit 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.

[Kulkarni1999] Amit Kulkarni and Gary Minden. Active Networking Services for Wired/Wireless Networks, INFOCOM, New York, 1999.

[Leroy2002] X. Leroy, "The Objective Caml system, release 3.04", Documentation and user's manual,INRIA, France, December 2001.

[Lin1996] J. C. Lin and S. Paul, "RMTP: A reliable multicast transport protocol," in Proc. IEEE Infocom, pp. 1414--1425, March 1996.

[lperf] Distributed Application Support Team, "Iperf", http://dast.nlanr.net/Projects/Iperf

[Mayden1998] M. Hayden, "The Ensemble system", Ph.D. dissertation, Cornell University Computer Science Department, January 1998.

[Mills1988] D. L. Mills, Network Time Protocol (version1) specification and implementation. DARPA-Internet ReportRFC-1059, DARPA, 1988.

[Minden2002] 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.

[Moy1994] J. Moy. Multicast Extensions to OSPF. Internet Requests For Comments (RFC) 1075, Mar. 1994.

[Moy1997] J. Moy, OSPF Version 2, Internet Request for Comments, RFC 2178, July 1997.

[Pusateri2000] T. Pusateri, "DVMRP version 3," draft-ietf-idmr-dvmrp-v3-10, August 2000.

[Remy1999] Didier Remy, Xavier Leroy, Pierre Weis, "Objective Caml – A general purpose highlevel programming language," INRIA Rocquencout, France, ERCIM News, (36), January 1999.

[Stiller1995] Burkhard Stiller, "CogPiT – Configuration of Protocols in TIP," Computer Laboratory Technical Report TR368, University of Cambridge, Cambridge, England, June 1995.

[vanRenesse1995] Robbert van Renesse, Kenneth Birman, Roy Friedman, Mark Hayden and David Karr. A Framework for Protocol Composition, Proceedings of the Principles of Distributed Computing, August 1995.

[Weatherall1998] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In IEEE OPENARCH, April 1998.

[Wong2001] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting, "A Configurable and Extensible Transport Protocol, Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001), Anchorage, Alaska, April 2001, pg. 319-328.