Building a Reliable Multicast Service Based on Composite Protocols for Active Networks

S. Subramaniam, E. Komp, M. Kannan, and G. Minden*

Information and Telecommunication Technology Center
Department of Electrical Engineering and Computer Science
University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045-7612 USA
gminden@ittc.ku.edu

Abstract. Active Networking embodies rapid development and deployment of new services. A network service typically consists of two or more cooperating protocols. In this paper, we describe a case study applying a composite protocol framework to developing a multicast network service. The composite protocol framework provides a rigorous mechanism to check protocol behavior before deployment. Our multicast serviced incorporates protocols for multicast routing, creation of spanning trees, reliable replication of multicast data and joining/leaving multicast groups. These protocols are built from re-usable components and communicate by means of global memory.

Keywords: Active networks, protocol components, composite protocols, composable services.

1 Introduction

In an active network [1], routers and switches in the network are programmable by the user and are capable of performing customized computations on packets passing through them. This allows easy injection of customized and innovative protocols and services into the network without the need for network-wide standardization. Several active networking architectures have been developed to deploy services need by an application on intermediate nodes of the network.

Active networking is built on the concept that many people will design, build, and deploy new protocols and services in the network. There is valid concern that network reliability is a risk if just anyone can place code in the network. One part of the effort to protect the network is insuring that new services are well thought-out, reasoned about, and tested before deployment. Composite Protocols [2] is a modular approach for specifying and implementing network protocols and services. In this paper, we present a case study of applying Composite Protocols to a multicast service.

* This research was supported by the Defense Advanced Research Projects Agency and the U.S. Air Force Research Laboratory under contract F30602-99-2-0516.

G.J. Minden et al. (Eds.): IWAN 2004, LNCS 3912, pp. 101–113, 2007.

Reliable-delivery, sequential delivery, error checking, some form of routing, authentication, and request/reply are some of the common functions used in protocols. Any new protocol may also use of some of these functions. We call such single-functional protocol modules, *protocol components*. A group of such protocol components collected and connected together by means of a composition operator constitutes a *protocol*. For example, Time-To-Live (TTL), Fragmentation, Header Checksum, Forwarding, and Addressing are protocol components, which contribute to the IP protocol. Though many forms of composition exist, the most common form of composition and the one used in our implementation is a linear composition.

A collection of two or more cooperating protocols is called a *service*. Multicast is an example of such a service. Multicast consists of protocols for group membership and management, multicast routing and spanning trees, tunneling and reliable replication of multicast data. Our composite protocol framework [2] describes how protocol components are specified and how these protocol components are composed to form a composite protocol.

1.1 Multicast Service

Traditional IP-based multicast network services typically consist of multicast routing protocols like DVMRP, MOSPF or PIM and group-management protocols like IGMP in operation. In this paper, we describe how a component based multicast service is built by stacking protocol components into three different protocol stacks: (1) a DVMRP like multicast routing stack for creating and managing multicast routing tables and spanning trees, (2) an IGMP like group-management stack for managing group-memberships and (3) a multicast-traffic delivery stack for reliable and secure transmission of application data. We then describe how these protocols communicate among themselves using a global memory object.

The rest of this paper is organized as follows. Section 2 describes the various steps involved in building a composable service using our framework with multicast service as a case study. Section 3 briefly discusses the functionality of all the protocol components that constitute the service and illustrates how the stacks cooperate together to render the multicast service. Section 4 focuses on Inter-stack communication, its need and forms of representation. Section 5 summarizes the results and presents the conclusion.

2 Building a Composite Multicast Service

Multicast is an excellent example of a network service which is made up of several cooperating protocols. IP Multicast is a collection of multicast routing protocols like DVMRP, MOSPF, PIM 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, OSPF,

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 complete implementations of Internet standards for DVRMP [3], IGMPv1 [4], and IGMPv2 [5]. We are interested is the basic functionality in these protocols and evaluation of the composite protocol framework. Only a sub-set of the standard functionality is specified and implemented. We assume that the reader has a basic understanding how IP multicast and other protocols like DVMRP and IGMP work in general.

2.1 Building a Composite Service

Step 1: Decomposition - Identify components from the monolithic protocols in the service.
For multicast service, we decomposed the monolithic DVMRP [4] protocol into the following protocol components: *Neighbor Discovery, Route Exchange, Spanning Tree, Pruning* and *Grafting*. The IGMP [5] protocol is decomposed into the following components: *Join/Leave* and *Query/Report*. Other components include *Multicast Forwarding, Reliable Multicast (ACK/NACK based), Security (Authentication / Encryption)*. Figure 1 illustrates these stacks.

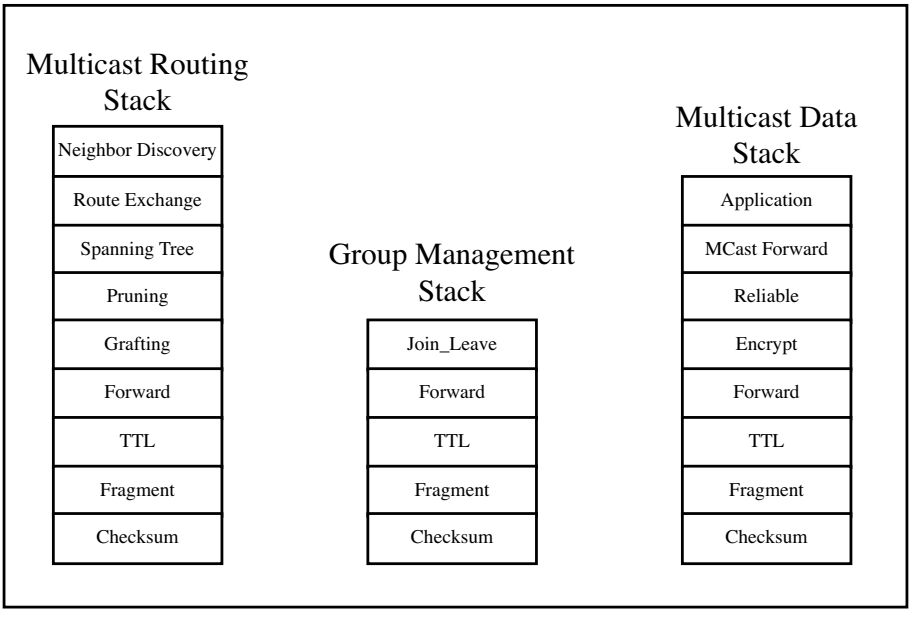

Fig. 1. Multicast service is a collection of three stacks viz. Multicast routing stack, Group Management stack and Multicast data/traffic stack. Multicast service is a collection of three stacks viz. Multicast routing stack, Group Management stack and Multicast data/traffic stack.

Step 2: Specification of protocol components.
Once the individual components are identified, the next step is to specify each of these components using Asynchronous Finite State Machines (AFSM) [6] as described in [2]. Each component is represented by a Transmit State Machine (TSM) and a Receiver State Machine (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 assumptions. The individual functionality of each protocol component is described later. 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. In practice, some minor amount of dependence on other protocol components may be required. We shall describe on the individual functionality of each protocol component in section 3.

Step 3: Building Protocol Stacks.
Once all the individual protocol components are specified, these are grouped into protocol stacks. The multicast service is the collection of these stacks.

Step 4: Deployment - Placing the stacks in the network.
Composite-protocol stacks are deployed in an Active Network.

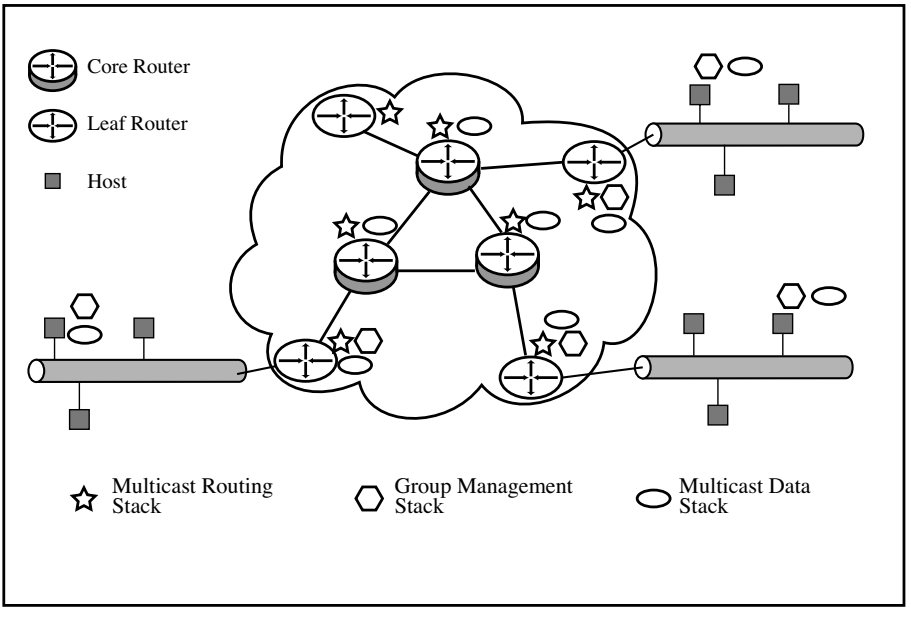

Fig. 2. An example multicast network showing core routers, leaf routers, and hosts. The shapes indicate where each protocol stack is deployed in the network.

Figure 2 shows an example multicast network with the following types of nodes:

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

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

3 Component Description

This section contains a brief description of each component in the multicast service. Detailed state machine specifications for each component are beyond the scope of this paper [8]. For each component, its sender (TSM) and receiver (RSM) functionality, access to global memory data structures and dependencies on other components are briefly discussed. We start with components from the multicast routing stack first.

3.1 Neighbor Discovery

This component forms a part of the multicast routing stack deployed at multicast core and leaf routers. The main functionality of this component is to dynamically discover neighbors (multicast routers) on all its interfaces. The sender side of this component 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. The receiver side of this component 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 a global data structure called *Neighbor Table*, which is part of Global Memory. It also provides a keep-alive function in order to quickly detect neighbor loss. Timers are used for sending probe packets and also for detecting dead neighbors. This component can be used in other protocols where there is a need for neighbor discovery e.g. in unicast routing protocols like RIP and OSPF.

3.2 Route Exchange

This component forms a part of the multicast routing stack deployed at all multicast core and leaf routers. 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 sender side of this component 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. For each route exchange packet received, the receiver 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*. Timers are used for the periodic transmission of *route exchange packets*. This component can be re-used in other distance vector-based unicast routing protocols like RIP.

3.3 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 sender side of this component needs access to the global *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 to their corresponding next-hops (which are actually upstream neighbors for the source networks in the list). The receiver side (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 as global data structure *Spanning Tree*.

3.4 Group Membership/Join Leave

This component forms a part of the *group management stack* deployed at multicast leaf routers and end-hosts. Initially, the IGMP protocol was decomposed into two separate components: *Join Leave* and *Query Report*. 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, these were merged into a single component called *Group Membership*. 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. At the end-host: The TSM responds to *Control events EJoin* and *ELeave* (These events are generated by the application when the host wants to join or leave a particular multicast group). It also updates the local *group cache* when these events occur. The RSM responds to the Query packets from the leaf-router by sending back a *Report packet* containing the list of group addresses it belongs to. At the multicast-leaf router: The TSM periodically multicasts *Query packets* on the local network to the "all-hosts-group" and the RSM processes the Report packets received from its attached hosts and updates the local *group cache* and the global memory structure *Group Members Table*. 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. This component thus creates and maintains the Group Members Table structure in global memory. Each multicast router contains in its Group Members Table the list of group addresses to which its attached hosts have joined.

3.5 Pruning

This component forms a part of the multicast routing stack deployed at multicast leaf and core routers. The primary purpose of this component is to create and maintain the global data structure *Prune Table* that stores the list of pruned downstream interfaces for each source/group pair. This along with the *Spanning Tree* component constructs per source-group multicast trees at each node. (Note: the *Spanning Tree* component by itself constructs a per-source broadcast tree at each node). The sender side of this component is responsible for sending prune packets for a particular source-group pair addressed to the corresponding upstream neighbor under the following conditions:

1. If all its downstream dependent neighbors have sent prunes and all its IGMP interfaces are also pruned.
2. If all its downstream dependent neighbors have sent prunes and there are no IGMP interfaces (at multicast core routers).
3. If there are no downstream dependent neighbors and all IGMP interfaces are pruned (at multicast leaf routers).

The receiver side of this component is responsible for updating the global memory *Prune Table* with entries containing source, group and incoming interface (interface to be pruned). Note that the TSM reads from the *Prune Table* and the RSM writes to the *Prune Table*. Components from other stacks also write to the Prune Table. The Multicast Forwarding component writes to this structure when there are no members for the source-group present in all attached host interfaces. The Join Leave component (router side) of the group membership

stack also writes into this structure when a last member of a particular group leaves a multicast group. Thus this design of this component addresses some intra-stack communication issues. The global memory Prune Table is used here to communicate between the two stacks. These issues are discussed further in section 4.

3.6 Grafting

This component also forms a part of the multicast routing stack deployed at multicast core and leaf routers. 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 sender side of this component reads from the global Prune Table, and sends a separate graft packet to appropriate upstream routers for each source network under the following conditions:

1. On leaf-routers if the interface attached to all hosts is un-pruned.
2. On core routers if a graft packet is received on all previously pruned downstream interfaces.

The receiver side of this component on receiving a graft packet writes to the global Prune Table to update the list of grafted 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. 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.

3.7 Multicast Forwarding Component

This is a part of the multicast data stack deployed at all nodes. This component is responsible for multicast of traffic on all the branches of the source-group 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 multicast the traffic (core and leaf routers) and end-hosts (multicast receivers) the TSM remains inactive and only the RSM is operational. The receiver first performs the RPF (Reverse Path Forwarding) check on the packet. This checks if the packet is received on the correct upstream interface, which is the one that is used to reach the source of the multicast packet. If the RPF check is successful, the RSM forwards the application data on (a) Each attached IGMP enabled interface if there are group members on that interface. If there are no group members then it writes to the

global memory *Prune Table* to prune the interface and drops the packet. (b) On all un-pruned branches of the tree to its downstream dependent neighbors. On multicast receivers it delivers the data to the application.

3.8 Multicast Reliability and Security Components

These components can be optionally inserted to the multicast data stack if needed by the application. The multicast reliable component if inserted below the Multicast Forwarding component provides hop-to-hop reliability. Several protocols have been developed to address reliable multicast. This component can be either designed as a sender-initiated component based on ACKs or as receiver-initiated component based on NACKs. The flexibility of the composite protocol framework supports the easy addition and removal of different versions of these reliable multicast protocol components. The security components consist of the Authentication or the Encryption components, which provide hop-to-hop authentication and privacy of application data. Different versions of these security components like Encryption based on DES or IDEA and Authentication based on MD5 or SHA can be used.

4 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.

Our services use a active node based global memory object (GMO) shared between the protocols. This GMO is independent of any protocol that uses it. The scope and extent of the GMOmust 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. For example, the RouteExchange component uses a function to write new routes into the Routing Table. It would use *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 use an external function *ipaddr getNextHopForDest (dest-addr)* to get the nexthop address. These functions *addNewRouteEntry()* and *getNextHopForDest()* are provided through the functional interface of the global Routing Table object.

Generally, the GMOcan be regarded as a server, providing access to shared information to protocols reading or writing shared information. For example, in the TCP/IP world the IP Routing Table is created and maintained by protocols like RIP or OSPF and is accessed by IP while forwarding data packets. In our framework, the routing table is maintained as a GMO that is external to both protocols IP and RIP.

4.1 Global Memory Attributes

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

Synchronization: Protocols access the GMO 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 which shares the information to incorporate these controls in its implementation. And the solution is much more robust, since the integrity of the shared data cannot be compromised by a single protocol, which does not correctly implement synchronization algorithm.

Extensibility: The GMO 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.

4.2 Implementing Global Memory

We consider three approaches to implementing a global memory object: a process model, a shared memory model, and a kernel based (NodeOS) based model.

In the process model, each GMO is implemented as a separate process running as a server on each node. Typically, each global memory server is started 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 local 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 limit performance of network protocol implementations.

In the shared memory model, the GMO is stored in shared memory. The functional interface containing the set of all functions provided by the GMO is packaged into a dynamic link library (DLL). The protocol stacks, which run as individual processes on a node, link to this library at run-time. Thus, each protocol stack imports a copy of the DLL code space.

The function implementation is visible only internally and is opaque to the protocols that use it. Each function internally invokes the shared memory functions for reading/writing into shared memory. The shared memory library routines handle synchronization.

The shared memory approach strongly preserves the abstract functional interface we want for global memory. Users of global memory have only an abstract view of it through the functional interface provided by the DLL. Thus, protocol components are not concerned with the details of the how the shared memory is accessed. Also, the semantics and syntax of shared memory access functions may differ depending on the operating system, but is has no effect on the protocol component. Shared memory function calls are generally faster than IPC function calls, thus providing faster global memory access.

A third alternative is to embed global memory objects directly in the operating system on which the protocols run. With this alternative, the operating system (kernel) interface must be expanded to incorporate the GMO functional interface. The operating system implicitly operates as the GMO server. 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.

4.3 Initialization

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

The above figure illustrates different protocols of the multicast service cooperating by means of global memory objects. NeighborTable, RoutingTable, SpanningTree, PruneTable and GroupMemberTable are all global memory objects that provide a set of read/write functions through their respective functional interfaces. For example, 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

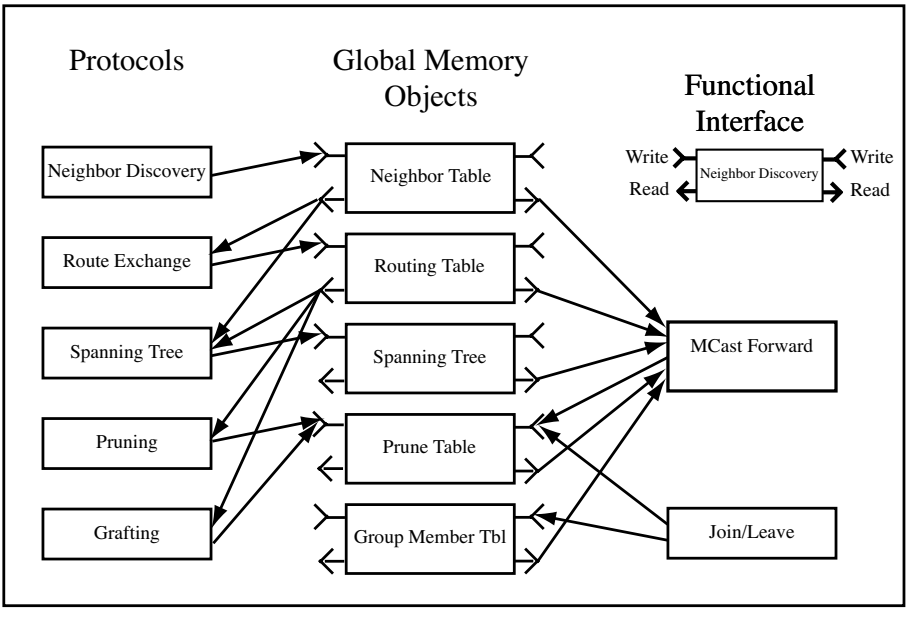

Fig. 3. The global memory of the multicast service and the protocols that access each memory section

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.

4.4 Independence

The global memory objects are designed to be mutually independent of each other. A multicast service may need both the global memory objects Routing Table and Spanning Tree, but another network service might require only the Routing Table. Dependency of the Routing Table on the Spanning Tree is undesirable.

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

5 Conclusion

A multicast network service has served as a case study in understanding a composite protocol design framework. The basic functionality of traditional IP multicast protocols DVMRP and IGMP have been successfully expressed in the form of several protocol components and composite protocol stacks. Global memory

has been proposed as a solution for inter-stack communication in our framework. Global memory design and features have been presented.

References

[1] D. Tennenhouse, J. Smith, W. Sincoskie, D. Weatherall, G. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, Vol. 35, 1997.
[2] 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.
[3] Pusateri, T. "Distance-Vector Multicast Routing Protocol Version 3."
[4] S. Deering "Host Extensions for IP Multicasting", RFC 1112, August 1989.
[5] W. Fenner, "Internet Group Management Protocol, Version 2", RFC 2236, Xerox PARC, November 1997.
[6] Yuri Gurevich, Sequential Abstract State Machines Capture Sequential Algorithms, ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.
[7] M. Hayden, "The Ensemble system", Ph.D. dissertation, Cornell University Computer Science Department, January 1998.
[8] S. Subramaniam, E. Komp, G. J. Minden and J Evans, Building a Reliable Multicast Service Based on Composite Protocols, The University of Kansas, Information and Telecommunications Center, ITTC-F2004-TR-19740-11, Lawrence, Kansas, July 2003.