INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

TREE AUTO-CONFIGURATION FOR THE RELIABLE MULTICAST TRANSPORT PROTOCOL

Wang Tianyu

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER

SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2002

ABSTRACT

TREE AUTO-CONFIGURATION FOR THE RELIABLE MULTICAST TRANSPORT PROTOCOL

Wang Tianyu

The Reliable Multicast Transport Protocol is a hierarchical transport level protocol for IP multicast that provides reliable data transmission. The objective of this thesis is to define and implement the Tree Auto-Configuration algorithm, which generates a tree topology used for tree based reliable multicast transport protocols. The tree topology is comprised of a pre-deployed mesh, multiple senders, local service nodes, and a large number of receivers. We implement this algorithm in C++ based on Meta-Transport Library, which is a set of C++ base classes designed to present an infrastructure for building transport protocols.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. J. W. Atwood, for introducing me to multicast protocols, the next generation Internet protocol. His insight and advice support me to complete this thesis, and lead me to pursue my Ph. D.

I dearly thank all my family and close friends. My wife, Wang Qian, always gives me her support and understanding whenever I need them. My mother, Huang Qi, supports me both in life and in career for many years.

Contents

List of Figures

List of Tables

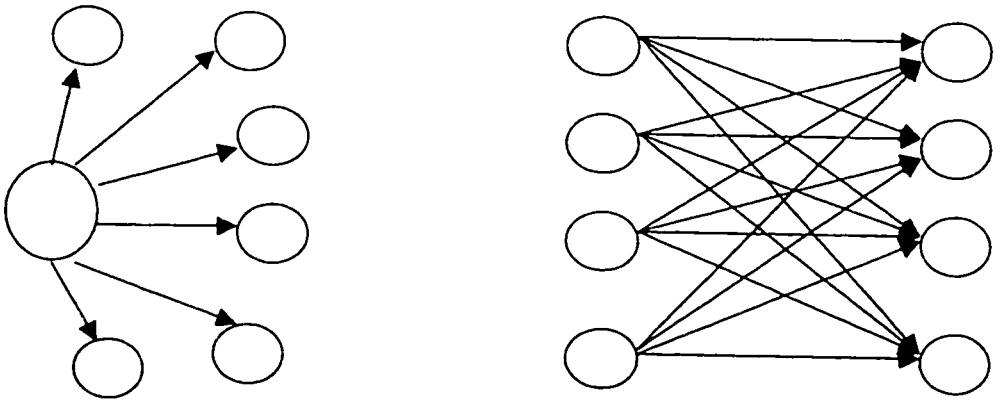
Chapter 1. Introduction.

1.1 Multicast.

There is a growing requirement for techniques that can exchange messages among a group on the Internet. This kind of group communication involves more than two application processes. Such group communication requirements exist in several applications, both in the local-area and wide-area environments, such as distributed-database applications, client and multi-server arrangements, multimedia applications, teleconferencing and video conferencing, industrial automation and process control.

In the current TCP/IP client-server model, this group communication requirement cannot be satisfied efficiently: the server has to serve every client and a separate copy of the message is sent from the source to every single destination. Because the server has just limited resource, it cannot efficiently serve all data requirements from an enormous number of clients. When such a situation occurs, the server's capacity will become the bottleneck of the group communication and delivering quality of service (QoS) will become impossible.

When data is to be sent from a single source to multiple destinations, unicast has been proven to be inefficient. A better and more reliable alternative solution is multicast. In multicast, the sender just sends a single copy to all receivers, and this communication group can be identified by a single group address. The multicast group can also encapsulate the internal states of this group and conceal the interactions of those group members. The multicast protocols can be fault-tolerant, despite random communication delays and communication or application process failure. The multicast protocols can enable the application processes to maintain consistent states of the communications and retransmit data packets if necessary.



Point-To-Multipoint (1to-N) Multipoint-to-Multipoint (N-to-M)

Figure 1: Topologies of Multicast

Based on user's requirement, the multicast protocols can be divided into two groups, Point-to-Multipoint (1-to-N) protocols, which send data from a single source to multiple destinations, and Multipoint-to-Multipoint

(N-to-M) protocols, which involve multiple senders and multiple receivers. The topologies of these multicast protocols are shown in Figure 1.

Multicast is an efficient way to distribute information from a single source to multiple destinations. Efficiency in multicast comes from two ways:

a. Number of transmissions from a source.
b. Number of packets generated within the network.

A source need only transmit once instead of n times for n destinations when multicast is used instead of multiple unicasts. Similarly, by virtue of using a source-based tree at the network level for distribution, multicast is able to reduce the number of packets within the network significantly compared to multiple unicasts [1].

Multicast communication is the focus of intense study in the research community. It is growing into a true deployment challenge for Internet engineers. It is now being offered by some Internet service providers (ISPs), and it is starting to be used by a number of companies offering large-scale Internet applications and services.

As for next-generation Internet, multicasting is one of a few techniques without which a certain class of application is absolutely infeasible. Although multicast has received lots of attention, it still has some issues to be resolved. As a result of this, the multicast protocol is still evolving, and some standards are still not finished yet.

There are several existing multicast protocols: DVMRP, MOSPF, PIM-DM/SM, CBT, MBGP/PIM-SM/MSDP, and so on. We will discuss these protocols later.

1.2 Motivation of Thesis.

The existing multicast protocols have some problems in scalability and manageability. These problems come from the topologies the protocols use and the growth of nodes involved in multicast communications. The existing protocols do not provide reliable communication capability as Reliable Multicast Transport Protocol (RMTP) does.

RMTP is a hierarchical protocol that provides reliable data transmission from a few senders to a large group of receivers. An RMTP tree consists of a single top node (TN), one or more sender nodes (SDs), many

receivers (LNs), and zero or more designated receivers (DRs). There is considerable difficulty in configuring the topology of the hierarchy in a way that is approximately congruent with the underlying physical network topology. So the designers of RMTP have left this issue out of their original design and have focused their work on the core features needed for reliable delivery.

In this thesis, we propose an algorithm that defines a standard process to generate a tree topology used for tree based reliable multicast protocol, such as RMTP. This algorithm is motivated by and concerned with the requirements of the tree-based acknowledgement (TRACK) multicast building block, and can be extended to provide other services. This tree topology can also provide scalability and manageability to multicast protocols.

1.3 Scope of the Thesis.

In Chapter 2, we will start our discussion with the overview of multicast protocols. In this chapter, we describe these protocols and discuss their advantages and disadvantages, especially RMTP.

In Chapter 3, we introduce the definition and specification of the Tree

Auto-configuration algorithm. The terminology is also provided in this chapter.

In Chapter 4, we give the details of the algorithm implementation. We also introduce the Meta-Transport Library (MTL) on which the implementation is based.

In Chapter 5, we offer our conclusion and discuss possible future work.

Chapter 2. Overview of Multicast Protocols

2.1 Multicast protocols in Network layer.

There are some multicast protocols designed in Network layer. These protocols are divided in two groups: the Intra-domain multicast protocols and the Inter-domain multicast protocols.

2.1.1 Intra-domain Multicast Protocols.

From the first Internet multicast experiment in 1992, the Internet multicast protocols development was focused on a single flat topology. There were several multicast routing protocols developed for this flat topology in Internet multicast standardization and deployment. From the middle of 1997, the research community realized the need for a hierarchical multicast infrastructure and inter-domain routing [2]. The existing multicast protocols before 1997 are now called intra-domain multicast protocols.

2.1.1.1 The Standard IP Multicast Model.

Stephen Deering is responsible for describing the standard multicast

model for IP networks [3]. This IP multicast model describes the explicit and implicit requirement for how end systems in an IP network send and receive multicast packets. The model has [4]:

a. IP-style semantics. A source can send multicast packets at any time, with no need to register or to schedule transmission. IP multicast is based on User Datagram Protocol (UDP), so packets are delivered using a best-effort policy.
b. Open groups. Sources only need to know a multicast address. They do not need to know group membership, and they do not need to be a member of the multicast group to which they are sending. A group can have any number of sources.
c. Dynamic groups. Multicast group members can join and leave a multicast group at will. There is no need to register, synchronize, or negotiate with a centralized group management entity.

The standard IP multicast model is an end-system specification and does not discuss requirements for how the network should perform routing. The model also does not specify any mechanisms for providing quality of service, security, or address allocation.

2.1.1.2 The early Multicast Backbone (MBone) and the

DVMRP protocol.

The early efforts of building multicast-capable Internet and creation of Multicast Backbone, MBone, is motivated by Stephen Deering's IP multicast model. In March 1992, the MBone carried its first worldwide

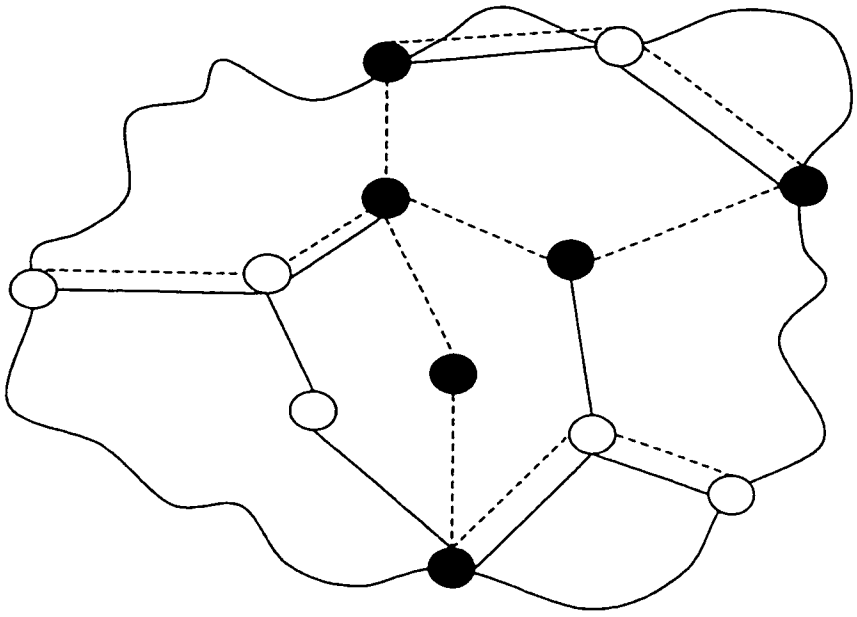


Figure 2. A tunnel-based topology of early MBone.

event when 20 sites received audio from the meeting of the IETF in San Diego. While the conferencing software itself represented a considerable accomplishment, the most significant achievement here was the deployment of a virtual multicast network [2].

A daemon process called *mrouted* was running on workstations, and this process provided the multicast routing function. While receiving unicast-encapsulated multicast packets from an incoming interface, the *mrouted* process will forward the packet through a proper set of outgoing interfaces. Connectivity among these machines is provided by point-to-point IP-encapsulated *tunnel* [2]. Each tunnel is a logical link between end-points, but it can cross several routers.

The original multicast routing protocol was the Distance Vector Multicast Routing Protocol (DVMRP). DVMRP constructs source-based multicast trees using Reverse-Path Multicast (RPM) protocol. So the multicast tree built by DVMRP is called a reverse shortest path tree. The basic RPM algorithm in this protocol is described as follow:

a. The source broadcasts each packet on its local network. An attached router receives the packet and sends it on all outgoing interfaces.
b. Each router that receives a packet performs a reverse path forwarding (RPF) check. That is, each router checks to see if the incoming interface on which a multicast is received is the interface the router should use as an outgoing interface to reach the source. In this way, a router will choose to only receive packets on the one interface that it believes is the most efficient path to the source. All packets received

on the proper interface are forwarded on all outgoing interfaces. All other packets will be discarded silently.

c. Eventually a packet will reach a router with some number of attached hosts. This *leaf router* will check to see if it knows of any group members on any of its attached subnets. A router discovers the existence of group members by periodically issuing Internet Group Management Protocol (IGMP) queries. If there are members, the leaf router forwards the multicast packets to the subnet. Otherwise, the leaf router will send a *prune message* toward the source on the RPF interface, that is, the interface the leaf router would use to forward packets to the source.

d. Prune packets are forwarded back toward the source, and routers along the way create prune state for the interface on which the prune message is received. If prune messages are received on all interfaces except the RPF interface, the router will send a prune message of its own toward the source [2].

The procedure of reverse shortest path tree construction is shown in Figure 3. In Figure 3, the arrow lines are the direction of datagram flows. The dashed lines are the direction of prune message flows.

In DVMRP, if a router is on the reverse shortest path tree and receives

datagrams from the source on the proper interface, it forwards received datagrams on the other interfaces only if the interface I on which the datagram arrives lies on the shortest path back to source. There are datagrams for a (source, group) pair periodically sending across the network, shown in Figure 3b. If a leaf router has no group members on its attached subnet, it periodically sends prune message toward the source, shown in Figure 3c. Periodically the prune state times out, and the process of forwarding datagrams across the entire network and trimming tree branches based on prune message received repeats [4]. The resulting shortest path tree is shown in Figure 3d.

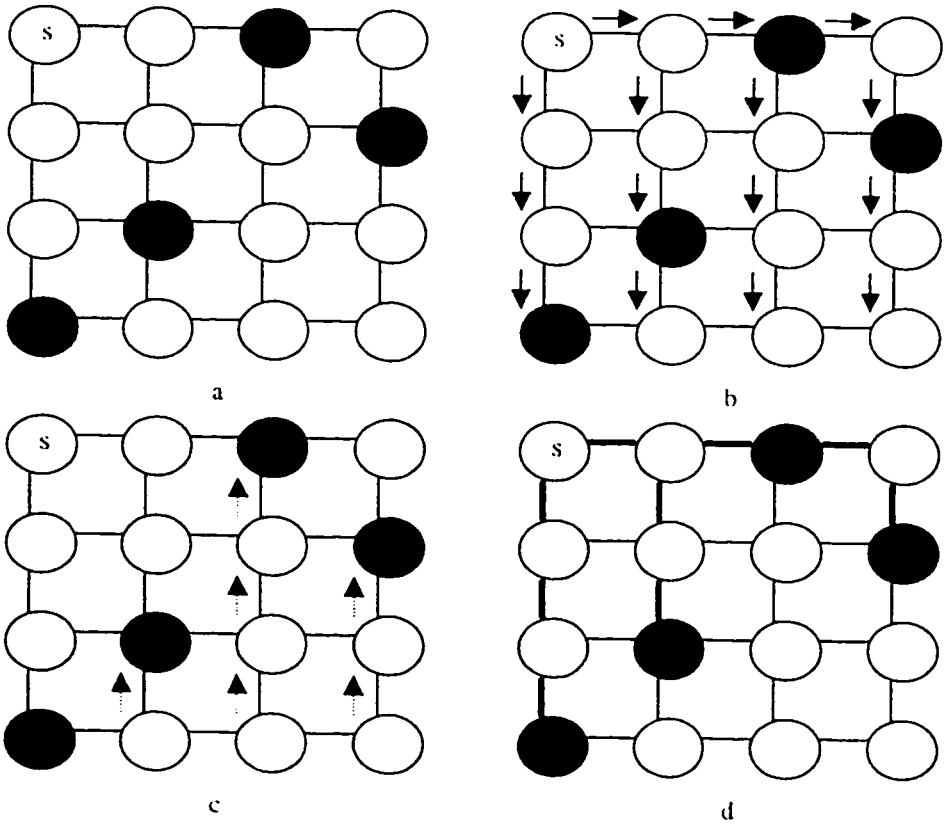


Figure 3. The tree construction process in DVMRP.

This method is a primitive multicast routing algorithm. It is actually a controlled form of flooding algorithm. It makes routers send a lot of prune message and forward lots of packets to achieve a better and dynamic routing solution. The large number of prunes and forwarded packets makes this algorithm inefficient and infeasible in a wide-area network, because there are thousands of routers that may be interested in this multicast session and there may also be thousands of multicast sessions on the Internet. A method to reduce the prune latency is to add a *Graft* message to the original protocol. If a router has sent a prune message previously and finds a new multicast group member on its attached subnet, it sends a *Graft* message to the upstream router. Then the upstream router that receives the *Graft* message updates its prune states accordingly.

2.1.1.3 Multicast Extensions to Open Shortest Path First (MOSPF)

This method is based on extending the Open Shortest Path First (OSPF) protocol to provide multicast routing capacity. In OSPF, each router keeps topological and state information of the routing domain, by link-state advertisement (LSA) flooding. Similarly, MOSPF routers use IGMP to

monitor multicast membership on directly attached subnets and flood an OSPF area with information about group receivers. This allows all MOSPF routers in this area to have the same view of group membership [2]. Each MOSPF router can independently construct the shortest-path tree for each source and group by Dijkstra's algorithm, in the same way as in OSPF. After the multicast tree is built, group membership is used to prune the branches that do not lead to subnets with group members. The result is a pruned shortest-path tree rooted at the source [4]. MOSPF is considered as a dense mode multicast protocol because the membership information is broadcast throughout the area and to all the MOSPF routers.

2.1.1.4 Protocol Independent Multicast – Dense Mode (PIM-DM)

Dense mode refers to an environment where the multicast members are relatively densely packed and bandwidth is plentiful [4]. PIM-DM is similar to DVMRP. The PIM-DM uses the RPM algorithm and uses *Graft* messages to add branches that have been previously pruned. There are only two differences between these two kinds of multicast protocol. The first one is that PIM takes advantage of the IP routing information to perform the RPF checks, while DVMRP maintains its own routing table.

The second is that DVMRP tries to avoid sending unnecessary packets to its neighbors who will generate prune message based on a failed RPF check. So the DVMPF router builds its routing table in a way that the routing table only includes the downstream routers that use the given router to reach the source. PIM-DM simply floods packets on all outgoing interfaces.

The multicast protocols described above are all dense mode multicast protocols, which broadcast membership information throughout the network. Now let us discuss another class of multicast protocol, sparse mode multicast protocols, which explicitly send join requests to the router, which acts as a core, without widely broadcasting traffic and triggering the prune message. The dense mode multicast protocols are usually used for an area where a lot of members are located in the area and interested in the multicast session. The sparse mode multicast protocols are used when there are only a few widely distributed group members. We will discuss two kinds of sparse mode multicast protocols: CBT and PIM-SM.

2.1.1.5 The Core Based Tree (CBT)

CBT uses the basic sparse mode paradigm to create a single *shared tree* used by all sources [2]. The root of this shared tree is called a core. All

senders send their data to the core, and the core forwards these data packet to all receivers. Receivers send explicit join messages to the core. The shared tree is a bi-directional tree, which is more complicated but more efficient when a packet traveling from a source to the core comes across branches of the multicast tree.

A host first sends a *join-request* message to the local router. This step is to explicitly express its interest in the multicast session. Then the local router will contact the next-hop router on the shortest path toward the core router. The *join-request* message sets up transient join states on the router on the path it traverses. The *join-request* travels hop by hop toward the core, until a core or an on-tree router receives this message and accepts this join request. Then the router that accepts this new child sends a *join-acknowledgement* back along the reverse path to the router, which initiates the join request. When a router on the path, which received the *join-request* previously and is in join state, receives this join-acknowledgement, it updates its forwarding table, becomes an on-tree router, and forwards the *join-acknowledgement* toward the requesting router.

Figure 4 shows the member join procedure in CBT. The R1 and R2 represent the direction of *join-request* messages, and A1 and A2 represent

the direction of *join-acknowledgement* messages.

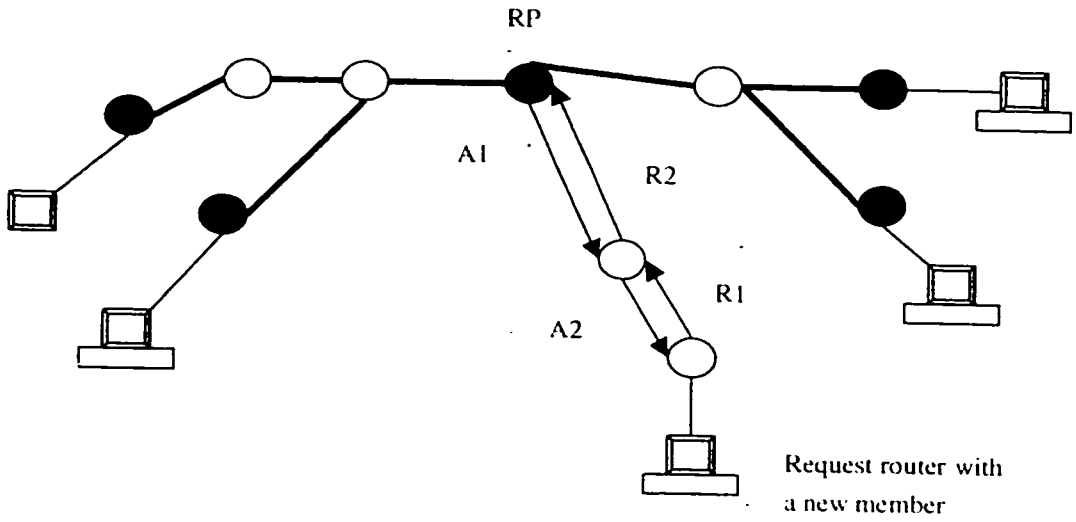


Figure 4. The member join process in CBT.

There is a dynamic and automatic tree maintenance mechanism in CBT. The routers can periodically send a CBT "keep-alive" (i.e., *echo-request*) to its parent router on the tree. The parent router sends a response (i.e., *echo-reply*) back to its child, if the parent receives a "keep-alive" message from a valid child. If there is no response in a predefined time threshold, the child should send a "*quit-notification*" message toward the core and send "*flush-tree*" message to all downstream branches. In this way, all its child routers can know the changes of the multicast tree, leave the tree, and re-join individually, if it is necessary. If a member host wants to leave the multicast tree and its directly attached router does not have any other member on its subnet, the local router sends a

"*quit-notification*" message to its parent and deletes the corresponding forwarding entry.

During data transmission, the data packets flow from the source to its parents and its children. Then the parent of the source sends the datagrams to its children other than the source, and forwards toward the core. This procedure continues until the datagrams reach the core. Then the core sends datagrams to other downstream branches.

2.1.1.6 Protocol Independent Multicast – Sparse Mode (PIM-SM)

Sparse mode refers to an environment where group members are distributed across many regions of the network, and bandwidth is not necessarily widely available. PIM-SM is similar to the PIM-DM in routing decision. It uses the underlying unicast routing table. However, it uses the similar tree construction algorithm to that used by CBT. PIM-SM uses a unidirectional-shared tree:

a. A core called a rendezvous point (RP) in PIM terminology must be configured. Different groups may use different routers for RP's, but a group can only have a single RP. All routers must discover information about which routers in the network are RP's and the

mapping of multicast groups to RP's. RP discovery is done using a bootstrap protocol. However, because the RP discovery mechanism is not included in the PIM-SMv1 specification, each vendor implementation of PIM-SMv1 has its own RP discovery mechanism. For PIM-SMv2, the bootstrap protocol is included in the protocol specification. The basic function of bootstrap protocol, in additional to RP discovery, is to provide robustness on case of RP failure. The bootstrap protocol includes mechanisms to select an alternate RP if the primary RP goes down.

b. A receiver that wishes to join the multicast tree contacts its directly attached router via IGMP query/report messages. The local router then creates a forwarding cache for (*, group) pair, which means that it joins all sources to that group, and explicitly joins the distribution tree by sending a unicast PIM-join message to the group RP. An intermediate router forwards the PIM-join message and creates the (*, group) pair. As with other multicast protocols, the tree is a reverse shortest path tree – the join message follows a reverse path from receivers to the RP.

c. When a source host first transmits a multicast packet to a group, the local router encapsulates the packet in a PIM-register packet and

unicasts it to the RP. When an RP receives one of these PIM-register packets, a number of actions are possible. First, if the RP has forwarding state of the group (i.e., there are receivers who have joined the group), the encapsulation is stripped off the packet, and it is sent on the shared tree. However, if the RP does not have forwarding state

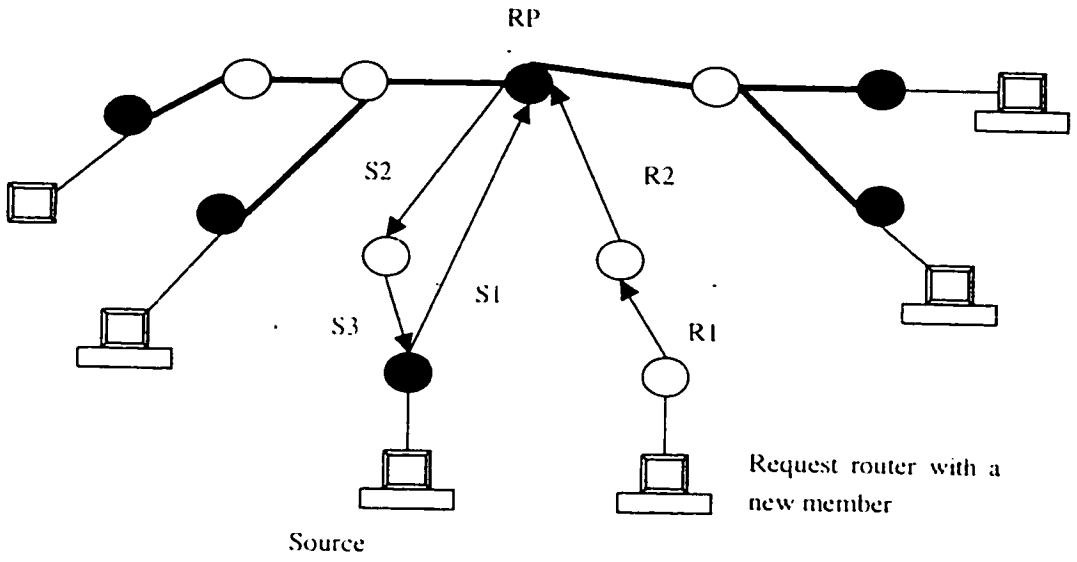


Figure 5. The member join process in PIM-SM.

for the group, it sends a register-stop message to the source. This avoids wasting bandwidth between the source and the RP. Second, the RP may wish to send a PIM-join message toward the source. By establishing multicast forwarding state between the source and the RP, the RP can receive the source's traffic as multicast and avoid the overhead of encapsulation [2].

The basic goal is to use RP as a "meeting place" for source and receivers.

A PIM-SM tree is rooted at the RP and constructed to span all the group members. Receivers send an explicit join message to the RP. Forwarding state is created in each router along the path from the receiver to the RP. A single shared tree, rooted at the RP, is formed for each group. The source first sends PIM-register message to RP via its local router. The primary RP then transmits a PIM-join message back to source router. Upon receipt of the PIM-join message from the RP, the source router then ceases to encapsulate data packets in RIM-registers but forwards them in the native multicast format to the RP.

Figure 5 show the PIM-SM tree built by the procedure mentioned above. In Figure 5, S1 is the PIM-register message sent from the source. S2 and S3 are PIM-join messages sent from RP to the source. R1 and R2 are PIM-join messages sent from a new group member to the RP.

PIM-SM also has a dynamic refresh mechanism. In the steady state, each router periodically sends PIM-join/prune messages, for each router active PIM route entry, to capture state, topology, and membership changes [4]. A PIM-join/prune message may also be sent based on some events that new route entries are added for some new sources. There is no form of explicit acknowledgement for PIM-join/prune messages. The periodic refresh mechanism can help routers recover from lost packets.

The sparse mode multicast protocols have some advantage over the dense mode multicast protocols. First, the sparse mode protocols have better scalability in terms of routing states. The routing states are only kept in the routers on the path between the source and receivers in sparse mode protocol, while the dense mode protocols require all routers to keep routing states. Second, the sparse mode protocols require receivers to join the shared tree explicitly, so the data flows across only the links that have been explicitly added to the tree.

The sparse mode multicast protocol also have some disadvantages. First, the RP can a single point of failure. Using bootstrap protocols, which can select an alternate RP, can solve this problem. Second, the RP can be the bottleneck of multicast traffic. The third problem is that having data forwarding from a source to RP and then to each receiver means that a non-optimal path may exist from source to receivers. The CBT protocol solves the second and the third problem by using a bi-directional shared tree. The PIM-SM provides a mechanism to switch the shared tree to the shortest path tree to solve these problems. When some traffic rate threshold is exceeded, this mechanism is triggered. Forwarding state is changed, so that the data flow is sent from the source to the receivers directly across the shortest path, instead of through the RP.

2.1.1.7 Problems with intra-domain multicast

As the MBone has grown, it has suffered from an increasing number of problems, and these problems have been occurring with increasing frequency [2]. The problems of scalability and manageability are the two most important problems.

Scalability problem comes from inherent instability of a large, flat network and the organizational mechanisms that do not provide significant routing aggregation. At its peak, the MBone had almost 10,000 routes. Unfortunately, most of these routes had long prefixes (between /28 and /32), which meant that very few hosts could be represented in each routing table entry [2]. The best solution for this problem is route aggregation and hierarchical routing. The problem now is how to apply the solutions to multicast.

The manageability problem comes from the fact that there are no protocol mechanisms to build a hierarchical multicast routing topology. The current model of Internet is to establish autonomous system (AS) boundaries between Internet domains. Autonomous systems are owned and managed by different organizations. Entities in one AS are typically

not trusted by entities in another AS. As a result, routing information is very carefully exchanged across AS boundaries. Peering relationships among ASes are provisioned using the Border Gateway Protocol (BGP), which provides routing abstraction and policy control. Because the current intra-domain multicast routing protocols do not provide such an inter-domain protocol, it does not offer the protection across domain boundaries. When there is a single flat topology connected using tunnels, routing problems can easily spread throughout the topology.

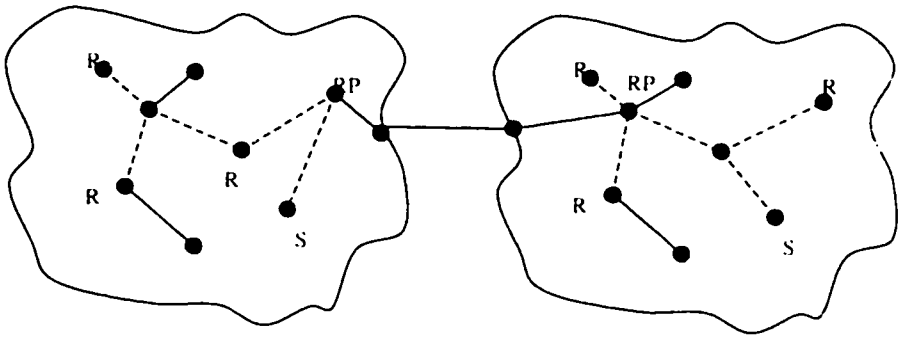


Figure 6. The problem of connecting sources and receivers across two sparse mode domains.

As an example, let us discuss a problem with PIM-SM. It is difficult to inform an RP in one domain that there is a source in another domain. The underlying assumption is that a multicast group, which spans two or more domains, can have multiple RPs, each domain has only one RP. There is no mechanism to connect the various intra-domain multicast trees together. When sources are located in different domains, as shown in Figure 6, receivers cannot discover the existence of sources in another

domain using different RPs. There is no mechanism for RPs to communicate with each other when one receives a source register message.

2.1.2 Inter-domain Multicast Protocols.

Since 1997, multicast research community realized the need for a hierarchical multicast infrastructure and inter-domain routing. Inter-domain multicast has evolved out of the need to provide scalable, hierarchical, Internet-wide multicast [2]. However, the inter-domain technology is relatively immature. Protocols that provide the necessary functionality are being considered by the IETF and are being evaluated through extensive deployment. Because the protocols lack elegance and long-term scalability, they are considered as a near-term solution and possibly only an interim solution.

2.1.2.1 Carrying Multicast Routes in BGP.

It is a straightforward extension of the inter-domain unicast routing and follows the need of making multicast routing hierarchical in the same way as unicast routing. BGP provides route aggregation and abstraction as well as hop-by-hop routing policy. ISPs or administrators can choose

any routing protocol within their own domains.

BGP can be used to reliably exchange network reachability information among ASes. This information is used to compute an end-to-end distance-vector-style path of AS numbers. Each AS can advertise the routes it can reach and the associated costs with the routes. Then the border routers can find the proper route to reach any network by using a distance vector algorithm together with full path information, which is an improvement over the traditional distance vector algorithm.

The functionality provided by BGP, and well-understood paradigm for connecting ASes, are important catalysts for supporting interdomain multicast. A version of BGP, which can carry multicast routes, is called Multi-protocol Extensions to BGP4 (MBGP). The MBGP is able to carry multi-protocol routes by adding the Subsequent Address Family Identifier (SAFI) to two BGP4 messages: MP_REACH_NLRI and MP_UNREACH_NLRI. The SAFI field can specify unicast, multicast, or unicast / multicast.

The MBGP can not only provide the hierarchical routing and policy decision for inter-domain multicast, but also use different topologies for unicast and multicast traffic. With MBGP, each router only needs to know

its own domain topology and routes to reach other domains, instead of the entire flat multicast topology. The domains can be connected together by different connections for unicast and multicast, as one case in Figure 7.

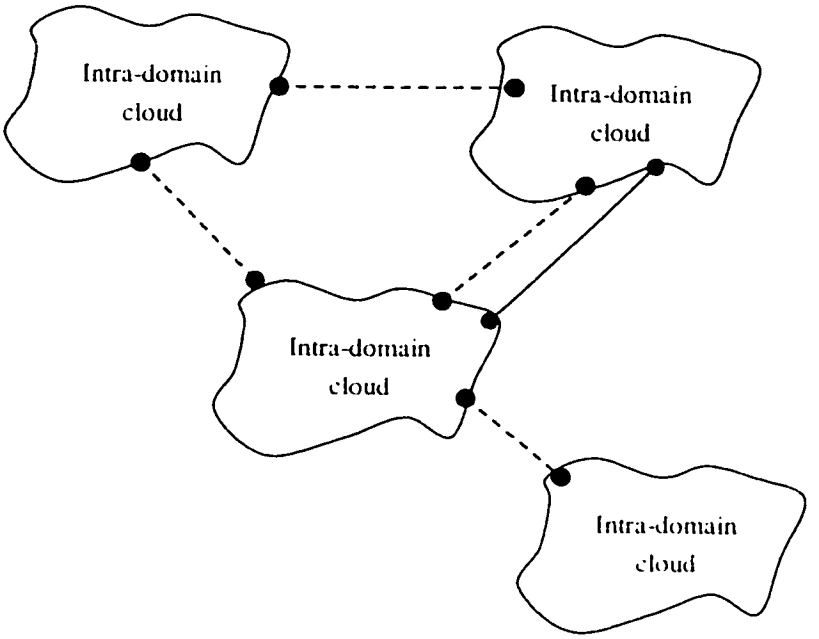


Figure 7. Inter-domain multicast topology running BGP and/or MBGP

In Figure 7, dotted lines are Multicast-capable connections between intra-domain multicast clouds (ASes), and solid line is a unicast connection. MBGP messages do not carry information about multicast groups (i.e., class D addresses). MBGP information is used when a join message is sent from a RP or receiver toward the source [2]. An MBGP message can carry hop-by-hop routing information between domains. This kind of message can help the join message to find out the best reverse path to the RP or receiver. If the topology for multicast between domains is the same as the unicast topology, the reverse path is the same

next hop that unicast traffic would follow. MBGP allows network administrators to set up a different reverse path for a join message to follow, and a different forward path for data packets.

MBGP is not a complete solution for inter-domain multicast routing. It just provides next-hop information between domains. There is still a need for a tree construction solution.

2.1.2.2 The Multicast Source Discovery Protocol.

PIM-SM was chosen to build the multicast tree in the current short-term inter-domain multicast solution, because it can avoid the broadcast and prune method to build the tree. MBGP is chosen to provide the boundary routing information. To solve the problem shown in Figure 6, the multicast source discovery protocol (MSDP) was created. This protocol works by having representatives in each domain announce to other domains the existence of an active source. This protocol is run in the RP of a PIM-SM tree in a domain. The basic operation step is described as follows:

1) When a new source is active in one domain, it registers itself with the domain's RP.
2) The MSDP peer in this domain will detect the existence of the active source, and send a Source Active (SA) message to all directly

connected MSDP peers.

3) MSDP message flooding:

   -MSDP peers that receive an SA message will perform a peer-RPF check. The MSDP peer that received the SA message will check to see if the MSDP peer that sent the message is along the correct MSDP-peer path. These peer-RPF checks are necessary to prevent SA message looping.

   -If an MSDP peer receives an SA message on the correct interface, the message is forwarded to all MSDP peers except the one from which the message was received. This is called peer-RPF flooding.

4) Within a domain, an MSDP peer (also the RP) will check to see if it has state for any group members in that domain. If state does exist, the RP will send a PIM join message to the source address advertised in the SA message.

5) If data is contained in the message, the RP then forwards it on the multicast tree. Once group members receive data, they may choose to switch to a shortest path tree using PIM-SM conventions.

6) Steps 3-5 are repeated until all MSDP peers have received the SA message and all group members are receiving data from the source [2].

The solution is referred to with the abbreviation for the three relevant protocols: MBGP/PIM-SM/MSDP. MBGP/PIM-SM/MSDP is a

functional solution largely built on existing protocols. It is already deployed on the Internet with a fair amount of success.

However, MBGP/PIM-SM/MSDP still has some disadvantages. The first problem is the scalability problem. Because of the way MSDP operates, if multicast becomes tremendously successful, the overhead of MSDP may become too large [2]. If there are thousands of multicast sources located in thousands of domains, the number of SA messages being flooded could become very large.

Other problems occur in dynamic groups. In MSDP, information about the existence of sources must first be transmitted before routing state can be created. When groups are dynamic, due to either bursty sources and frequent group member join/leave events, the overhead of managing the group can be significant [2]. First, because SA messages are only sent periodically, there may be a significant delay between when new receivers join and when they hear the next SA message. To solve this problem, MSDP peers may be configured to cache SA messages. The trade-off is the extra state and complexity of maintaining the cache. Another problem is due to bursty source, which sends a short packet bursts separated by silent periods on the order of several minutes. It is difficult to establish a multicast tree for such a kind of source. When one

or a few packets are sent to the RP, the RP will hear the packet and flood an SA message, and RPs in other domain will send join message back to the source. However, because no multicast forwarding state existed when the original packet was sent and it takes time to flood SA message and to establish forwarding state in RPs in other domain, the original burst will not reach new receivers. Once forwarding states are created in all RPs, all subsequent packets will reach these receivers. However if the period of silence between packet-bursts exceeds the forwarding state timeout value, the forwarding states will be discarded. When another burst is sent, the procedure described above will be repeated. The initial packets are lost and no packets from bursty source reach group members.

2.2. Multicast Protocols in the Transport layer

There are some multicast protocols designed in Transport layer. We introduce some Transport Multicast protocols in this section: Local Group Multicast Protocol and Reliable Multicast Transport protocol.

2.2.1. Local Group based Multicast Protocol.

Local Group based Multicast Protocol is a multicast protocol designed in the Transport layer by Institute of Telematics, University of Karlsruhe.

The local Group based Multicast Protocol (LGMP) defines a hybrid approach. It supports reliable and semi-reliable transfer of both continuous media and data files [15].

LGMP is based on the principle of sub-grouping for local error recovery and local acknowledgement processing. Receivers dynamically organize themselves into subgroups, which are called Local Groups. They dynamically select a Group Controller to coordinate local transmissions and to handle status reposts. The selection of appropriate receivers as Group Controller is based on the current state of the network and of the receivers themselves. However, the selection of Group Controller is not a task of a data transfer protocol like LGMP. The author of LGMP has defined and implemented a separated configuration protocol, which is called Dynamic Configuration Protocol (DCP). Packet errors are firstly recovered inside Local Groups using a receiver-initiated approach. Missing data units are requested from the sender or a higher level Group Controller only if not even a single number of the Local Group holds a copy of the missing data unit. Otherwise, errors will be recovered by local retransmissions. Full reliability and efficient buffer utilization are ensured by a novel, three-state acknowledgement scheme [15].

DCP provides mechanisms for an automated establishment of virtual

group structures and for dynamic reconfiguration in accordance with the current network load and group membership. No manual administration is necessary. The definition of subgroups is based on a combination of multiple metrics depending on the QoS requirements of the user. DCP is self-organizing and tolerant with respect to failing controllers. Each Group Controller periodically sends packets of type LG_ADVERTISE to announce its existence. The DCP uses an Expanded Ring Advertisement scheme to reduce network load while allowing short reaction times upon changes within the local scope of a receiver. Group Controllers send their advertise message with dynamically changing TTL values, such as 15, 31, 15, 63, 15, 31, 15, 127 and so on. Receivers within a scope of 15 will get each of the advertise messages. If the distance of a host is between 16 and 31 hops, it will receive every second advertisement. This scheme continues in a way that every 16^{th} advertise message will be distributed worldwide. The Expanded Ring Advertisement ensures that the frequency of advertise message decreases exponentially with increasing scope. The Expanded Ring Advertisement could also be used to estimate the number of hops between a receiver and a Group Controller [15].

Initially, it is the founder of a Local Group that will become the Group Controller. Due to the joining and dropping out of receivers, the group structure has to be reconfigured dynamically during the lifetime of an

association. It might be beneficial to split a growing Local Group or to merge several waning subgroups. In addition, a joining receiver might be a better Group Controller than the current one. Therefore, a dynamic reconfiguration mechanism is used to choose a new Group Controller and rebuild the local group. All groups are bound to a tree that is the backbone of the multicast transmission. This multicast tree traverses most of the distance in the network.

2.2.2. Reliable Multicast Transport Protocol.

IP multicast is an Internet Engineering Task Force (IETF) standard that allows a sender to unreliably send a single packet to thousands of receivers. However, most applications require some form of reliability to recover from link-level losses and from the packet losses that are an inherent by-product of Internet congestion control [6]. Reliable Multicast Transport Protocol II is a transport protocol for IP multicast, which draws heavily on the original RMTP protocol [7]. The original RMTP protocol provided groundbreaking work in the use of a tree topology for acknowledgement (ACK) aggregation and local recovery, but it does not support negative ACKs (NACKs) or forward error control (FEC). Also, it only provides rudimentary control on the amount of ACK traffic it generates, and its automatic tree configuration algorithms are not suited

to all environments. Finally, it does not include explicit network management, counted membership, or time bounded reliability. So the IETF extended the core idea of RMTP to RMTP-II.

RMTP-II is a hierarchical protocol that provides reliable data transmission from a few senders to a large group of receivers. An RMTP-II tree consists of a single top node (TN), one or more sender nodes (SDs), many receivers (LNs), and zero or more designated receivers (DRs). There may be a backup top node. The topology of an RMTP-II tree is shown in Figure 8.

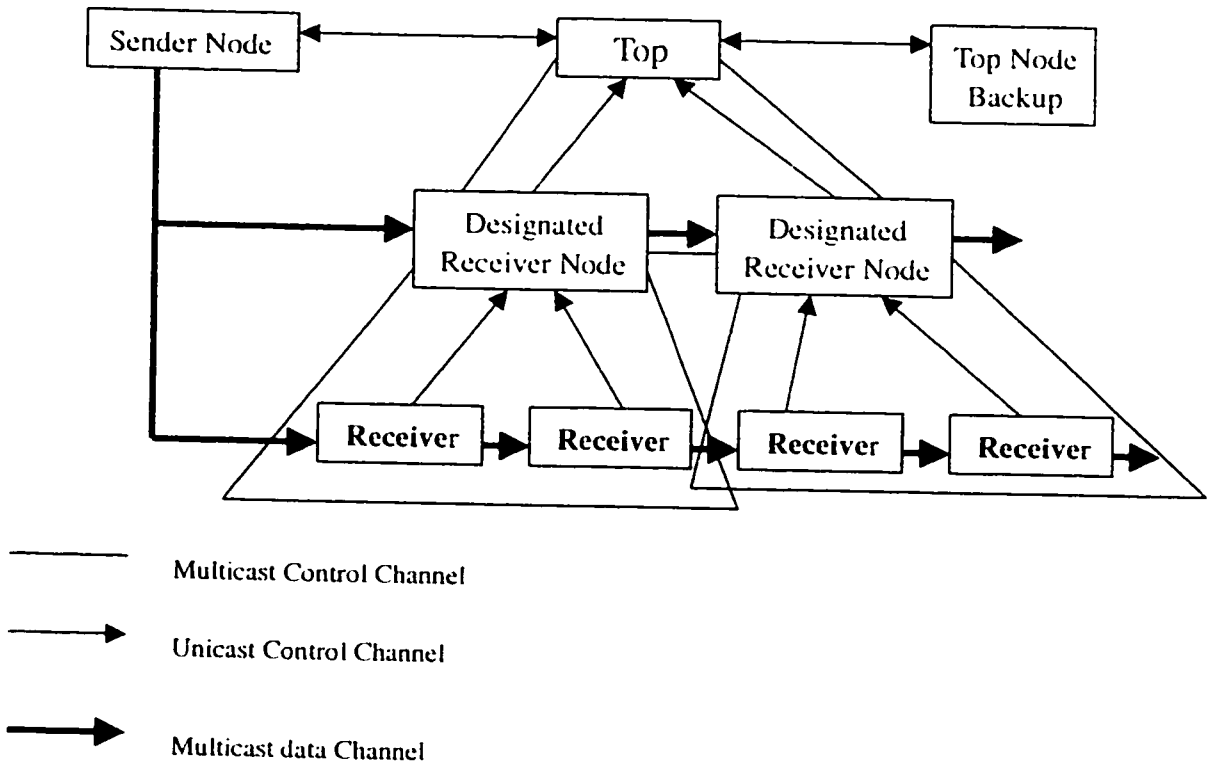


Figure 8. An RMTP-II tree.

The top node is assigned administratively and is the core of the tree [8]. It can control and dynamically change the transmission parameters and congestion parameters for each sender. One of its most important tasks is to aggregate the Tree-Based ACKs (TRACKs) received from its children and then send aggregated TRACKs to senders. The TRACK aggregation is essential in guaranteeing data reliability. The top node can also accept unicast data from the sender and multicast these data to the group, when the sender is not multicast capable. It is also the core of membership management and data stream ID allocation.

The sender node maintains the values of variables relating to the data stream: the size of the data queue, the packet admission rate, the sequence number of the lowest numbered packet that needs to be retransmitted, and the sequence number of the highest numbered packet in the data queue. It maintains data queue according to the TRACK received from the top node. It can retransmit a data packet if necessary.

The designated receiver can aggregate ACKs received from its children, send an aggregated ACK to its parent, and forward received NACK to its parent. The designated receiver accumulates the membership count of its receivers, and passes this count to its parent [8]. It can forward data retransmission received from its parent, and also it can buffer the data for

potential local data recovery.

A receiver joins the data stream by sending an explicit JoinStream message to its parent. It can send ACK for stable data and send NACK to expedite the recovery for missing data packets.

The top node (TN) and designated receivers (DRs) have local control channels. The control channel of a TN or DR is a multicast group that connects to all its children and is used for local multicast retransmission of lost packets. This multicast control channel is also used to transfer periodic heartbeat messages that inform children of the existence and functionality of itself. An RMTP-II tree can support multiple multicast data channels. A sender can use one or more data channel, and receivers must join all the data channels that they are interested in. A receiver uses a unicast control channel to periodically send TRACK to its parent. The TRACKs inform the parents about the packets that a receiver has or has not received. Each parent receives all the TRACKs from its children and sends a single aggregated TRACk to its parent via the unicast control channel. The TN can also send an aggregated TRACK to the sender via the unicast control channel.

The core of RMTP-II is a set of algorithms that provide and manage

Tree-Based ACKs (TRACKs), which is a key requirement of many applications that need group management and positive confirmation of data delivery to receivers. FEC and NACK algorithms are optional in RMTP-II. FEC is required in order to scale in the face of independent loss, and NACKs are essential for providing low-latency delivery in the face of packet loss.

The following is a list of the top RMTP-II design goals [7]:

1. Few-to-many delivery: The protocol supports up to a few senders per transport session, but without any ordering guarantees across these senders. Its primary focus is on scaling the number of receivers.
2. Receiver scalability: This is a primary design goal. Scalability is achieved through the combination of the following:
   a. Hierarchical positive ACKs, with strict management of the amount of return traffic.
   b. Local retransmission from Designated receivers (DRs).
   c. Use of optional FEC to reduce the effects of independently distributed loss.
   d. Use of optional NACKs to allow TRACKs to be sent very infrequently; NACKs are scoped in a hierarchical fashion, as opposed to being sent to the entire group.
3. Strong reliability guarantees: The protocol provides a fully distributed

group membership and acknowledgement algorithm, which allows TRACKs to give positive confirmation when all receivers have delivered a packet.

4. Explicit network management: All senders must connect to the TN in order to send to the group. The TN provides a centralized point of management and control for network managers.
5. Asymmetrical network support: The protocol differentiates between the control channel and the data channel, to allow the protocol to work in fundamentally asymmetrical environments, such as with a satellite downlink and a terrestrial return path.
6. Optional router assist: It has been well demonstrated that special software in the routers can increase the scalability of a protocol. However, if IP multicast is an example, this software tends to take an extremely long time to deploy widely. Hence, the protocol is designed to take advantage of generic router assist when it becomes available, but without requiring it for scalability.
7. Support of congestion control: RMTP-II is designed to support TCP-friendly congestion control. Its hierarchical positive ACKs allow it to solve two of the most difficult problems: slow start and scalable RTT measurement. Its use of hierarchy also allows it to take advantage of restricted worse edge calculations.
8. No requirement of many-to-many multicast: Some multicast

deployments do not support receivers being able to source multicast packets to the same group a sender is using. RMTP-II is designed not to require this feature.

9. Simplicity: While more complex than some protocols, RMTP-II strives to be as simple as possible. One way it does this is by differentiating the roles of the group members.

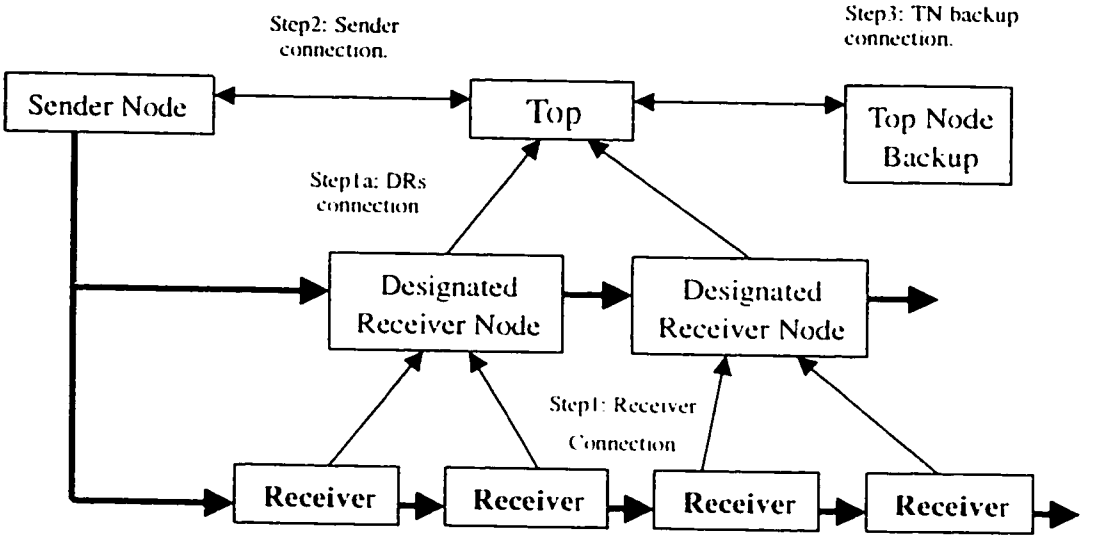


Figure 9. The normal operation of nodes joining the RMTP tree.

Figure 9 shows the normal operation process of nodes joining the RMTP tree. At the beginning, time 0, there are only a top node and two DRs running, but they are not connected to each other. At time 1, when four receivers want to join the tree, they contact the two DRs and join the tree. Then the DRs join the TN according to the join request of their children. At time 2, a sender joins the tree by connecting to the TN, and advertises

a new multicast data channel. The receivers will join the new data stream. The TN backup can join the tree at time 3.

In order to join a tree, each node, except the top node, sends a JoinStream packet to its parent node and sets a Join timer for the confirmation [8]. If the parent can accept the join request and confirm immediately, it responds with a BindConfirm message, or it responds with a BindACK, if it needs some time to process the join request. It sends BindConfirm, after it fully processes the request. Each node can leave the data stream by sending LeaveStream to its parent, and its parent responds to the leave request by sending LeaveConfirm. A designated receiver can only leave the tree when it does not have any child.

The hierarchical structure of RMTP-II has some disadvantages. First, the TN could be a potential bottleneck in the multicast transmission because of the risk of generating more control traffic than a NACK-only protocol; and it would seriously damage the multicast group if the TN were to fail. RMTP-II provides a set of smoothing and control algorithms to manage and limit the TRACK control traffic. These algorithms do not eliminate the control traffic trade-off, but allow it to be explicitly monitored and controlled [7]. RMTP-II also minimizes the risk that TN becomes the bottleneck of the system by minimizing the amount of work done by the TN, including restricting TN from transmission of the data packet.

RMTP-II also provides an optional hot backup of the top node to eliminate the potential single point failure of top node. However, the risk is still very high. Even more important, there is considerable difficulty in configuring the topology of the hierarchy in a way that is approximately congruent with the underlying physical network topology [7]. RMTP-II provides an algorithm for automatically configuring the tree if there is only a single level hierarchy, which can be sufficient for real-time applications of up to 100 or more receivers and non-real-time applications of up to 1000 or more receivers. For large deployment, RMTP-II assumes the existence of manual configuration files or a separate session manager component, to handle the configuration of interior tree nodes (DRs) [7]. So the designers of RMTP-II have left this issue out of their original design and focus their work on the core features needed for reliable delivery.

2.2.3. TRACK algorithm

The Internet Engineering Task Force (IETF) defines a building block, Tree Based Acknowledge (TRACK) in Reliable Multicast Protocols. This building block contains functions relating to positive acknowledgements and hierarchical tree construction and maintenance. It is designed to be useful as part of overlay multicast system that wishes to offer efficient confirmed delivery of multicast messages. The TRACK relies on a repair

tree to provide good-put as well as confirmed delivery. It provides some functionality: Hierarchical Session Creation and Maintenance, TRACK generation, Local Recovery, TRACK aggregation, and Distributed RTT Calculations [14].

The Hierarchical Session Creation and Maintenance functions are a set of functionality that is responsible for creating and maintaining the hierarchical tree of Repair Heads and Receivers. The TRACK generation is a set of functionality that is responsible for periodically generating TRACK messages from all receivers to acknowledge receipt of data, report missing messages, advance flow control windows, provide roundtrip time measurements and provide other group management information. The functionality of Local Recovery describes how repair heads maintain state on their children and provide repairs in response to requests for retransmission contained in TRACK messages. In order to provide the highest levels of scalability and reliability, interior tree nodes provide aggregation of control traffic flowing up the tree. The aggregated feedback information includes that used for end-to-end confirmed delivery, flow control, congestion control, and group membership monitoring and management, as described in section 2.2.2 [14].

The TRACK algorithm is proposed for one layer topology and expected

to extend to multiple layers. The repair tree is the infrastructure of the TRACK algorithm. However, the TRACK building block for Reliable Multicast Transport protocol does not provide an auto-configuration mechanism to build such a tree.

2.3 Summary and motivation.

As we can see in this chapter, there are some intra-domain multicast protocols designed in Network layer. These multicast protocols are designed to distribute data on the network. These network protocols can be divided into two groups: dense mode and sparse mode. The dense mode multicast protocols construct source-based trees and are suitable for multicast within a LAN. The sparse mode multicast protocols construct shared trees and are suitable for autonomous systems (ASes). As we discussed in section 2.1.1.7, these intra-domain protocols have some problems in manageability and scalability. The flat topology of these protocols cannot support inter-domain multicast communications. An example is the problem of RP advertisement in PIM-SM. Therefore the Internet research community introduced hierarchical inter-domain multicast protocols to solve these problems. The hierarchical topology can provide inter-domain capacity to multicast protocols. However, these inter-domain protocols are only short-term solutions and have some

problems as we discussed in section 2.1.2.2. The requirement for long-term solutions is rising.

Some multicast protocols are developed in transport layer. These protocols can be divided into two groups: multiple-to-multiple and one-to-multiple. The one-to-multiple transport protocols are used to distribute data packets from a single sender to a large number of receivers. These protocols need a tree topology. The multiple-to-multiple transport multicast protocols are based on collaborative groups and comprised of a small number of senders and a large number of receivers. These protocols need a multiple header tree topology to support multiple senders distributing data packets across the tree.

We introduced Local Group based Multicast Protocol (LGMP) and Reliable Multicast Transport Protocol (RMTP). The RMTP-II is an Internet Engineering Task Force (IETF) standard of reliable multicast transport protocol. This protocol has some problems. To support hierarchy, it needs interior nodes to bind to a tree. However, the tree topology is difficult to construct. Moreover, it needs multiple trees to support multiple multicast instances. These multiple trees may share a lot of the edges. This feature makes RMTP protocol inefficient and difficult to manage.

A tree topology is essential in all multicast protocols, both network layer and transport layer. It is the infrastructure of multicast data distribution and membership management. We develop our Tree Auto-Configuration algorithm as a solution to these problems we mentioned above. There are two important features in the topology generated by this algorithm: a mesh and the local groups.

We divided receivers into multiple local groups. Each group is comprised of a local service node and multiple receivers within a LAN. The local service node is responsible of managing local groups, contacting to upper level nodes, and distributing data packets among the local group. The group division can support multiple source-based trees. The local service node can join multiple multicast sessions and distribute received data packets among the local group. The receivers on different multicast trees can share the local service node as a common data source. The feature can improve efficiency of multicast and solve the problem for multiple instances in RMTP.

To bind all local groups together, this algorithm builds a mesh that is a group of pre-deployed service nodes. All local groups bind to the mesh directly or indirectly. Each sender can have its source-based tree. It

chooses a node on the mesh as the root of the multicast tree and sends data to the root. The root distributes data within the mesh. Each service node on the mesh sends data to its children, local service nodes. The local service nodes forward data to other local service nodes or leaf nodes, receivers. The mesh method can provide inter-domain capacity by configuring routers on the AS boundaries as a node on mesh. This feature can solve the problem for inter-domain session announcement in intra-domain protocols.

This algorithm is motivated by and concerned with the requirements of the tree-based acknowledgement (TRACK) algorithm. TRACK is designed to provide confirmed delivery, receiver-based flow control, distributed management of group membership, and provide aggregation of information up the tree. It also provides requests for retransmissions as part of TRACK messages, and local recovery of lost packets [14].

The algorithm can be extended to multiple-layered tree topology. The algorithm combines the Local Group Concept (LGC) from LGMP, Designated Receiver node and Aggregated Node from RMTP-II, and Mesh proposed in IETF internet draft "Reliable Multicast Transport Building Block: Tree Auto-Configuration" [6]. This algorithm happens to solve the problems of network layer multicast protocols as well.

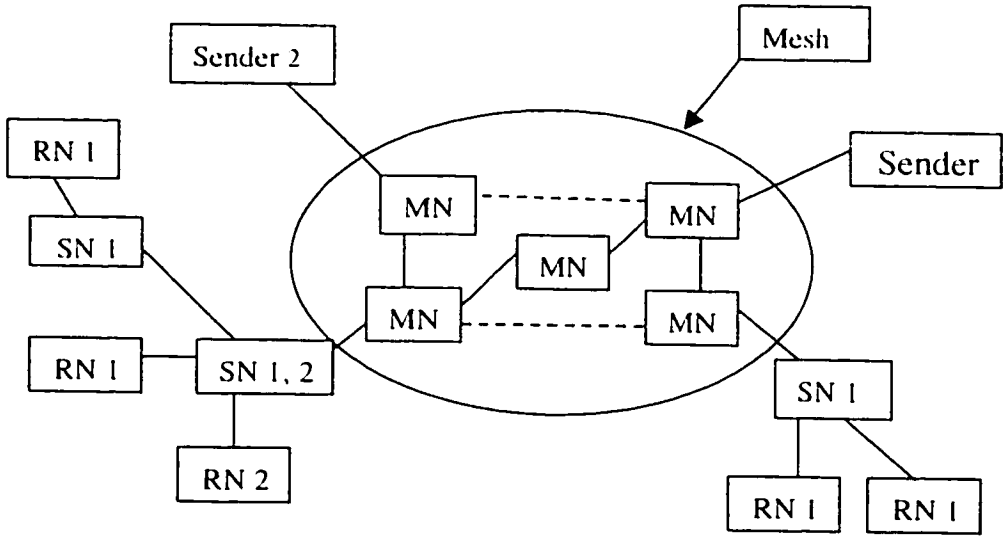
Chapter 3. Tree Auto-Configuration Algorithm.

3.1 Description.

As we have already seen in Chapter 2, both intra-domain and inter-domain protocols have some problems in scalability and manageability. The intra-domain protocols lack a hierarchical multicast routing topology. The existing inter-domain protocols still need a mechanism to build multicast trees in dynamic groups. Both of them lack reliable multicast capacity. The Reliable Multicast Transport Protocol II provides reliable multicast transmission, but still needs a tree construction mechanism.

The objective of this thesis is to define and implement a Tree Auto-Configuration algorithm that generates a hierarchical tree topology used for tree based multicast protocols. The algorithm can be used in sparse mode multicast protocols in which members join and leave the multicast group dynamically and explicitly. This algorithm is motivated by and concerned with the requirements of the tree-based acknowledgement (TRACK) multicast algorithms, and can be extended to provide reliable multicast transport and management.

The IETF draft, Reliable Multicast Transport Building Block: Tree Auto-Configuration [6], defines a process for auto-configuration of a tree comprised of a single Sender, Service Node, and Receivers into a tree, which is called a *session tree* in Reliable Multicast Protocol. A *session* is used to distribute data over a multicast address. A Session Tree is used to provide reliability and feedback services for a session [5]. This session tree is to satisfy the requirement of not only ACK-based protocol but also other services.



MN = Service Node in the mesh.
SN = Service Node
RN = Receiver Node
Dashed Line = Connection that is not used in the session trees.

Figure 10. Topology generated by Tree Auto-Configuration Algorithm.

Here we extend the basic topology proposed in the IETF's draft to a more complicated topology that is comprised of multiple Senders, multiple Service nodes, a pre-deployed mesh of service nodes, and receivers. This

topology can provide a session tree for each multicast group. Moreover, it is an infrastructure with better scalability for multiple multicast sessions. A sample topology is shown in Figure 10. The topology in Figure 10 is used by two multicast sessions. The numbers following the node types are session numbers of the multicast tree that the nodes joined.

The session tree auto-configuration algorithm proceeds generically as follow:

1. Mesh construction.
2. Sender locates a neighbor MN, sends a Session Announcement, and binds to the closest MN in the mesh. MN 'broadcasts' this session announcement on the mesh.
3. Receivers locate a local SN and send a BindRequest message to this local SN.
4. Local SNs join the session tree and accept receivers' bind request.
5. Session tree in the mesh is built.

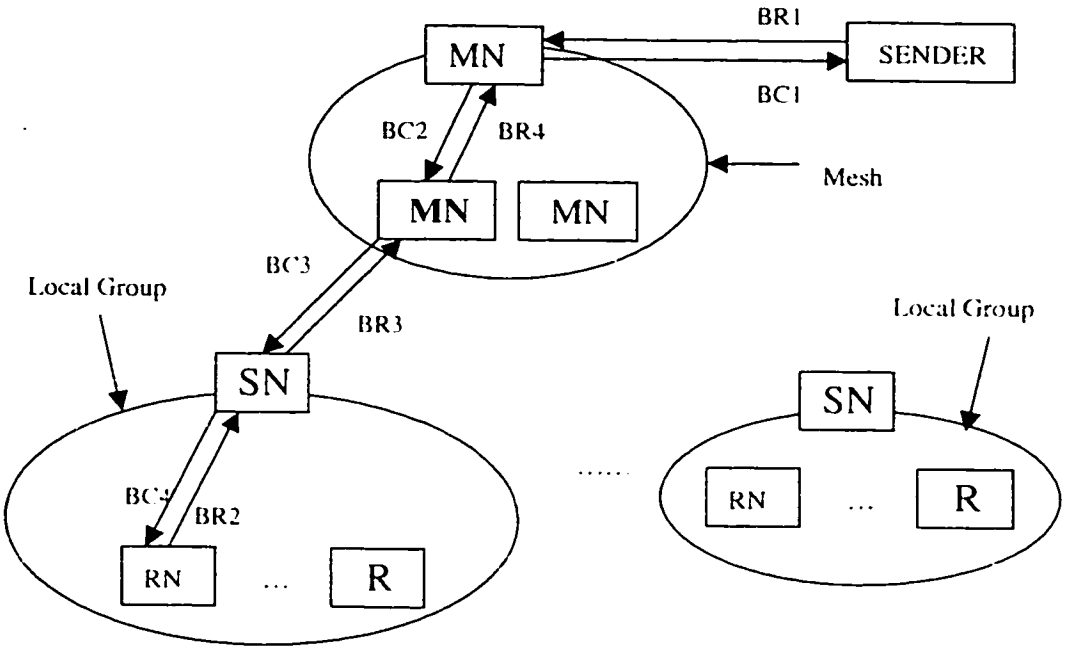
The Mesh shown in Figure 10 is a set of pre-deployed service nodes as infrastructure servers. These service nodes are online, but are not necessarily aware of any particular session unless informed by a Sender. We can call these service nodes on the mesh as *MN*. Each MN in the

mesh knows who its neighbor MN's in the mesh are and has a forwarding table. The forwarding table gives a "Next-hop" MN that can be used to reach the destination MN. A given MN can "broadcast" information to all other MN's on the mesh (in the sense of having a means of sending the same information to all other MNs, but not necessarily simultaneously). An MN can also cache the data packets for necessary retransmission. If each MN has direct connections to all other MNs, the Mesh is a fully connected mesh. The Mesh construction process and algorithm will be discussed in section 3.2.1.

The Sender should bind itself to the Mesh before it can start a multicast session. The Sender finds a neighboring MN, when it wants to start a multicast session. It sends a BindRequest message to the selected MN and waits for the BindConfirm message, as shown in Figure 10. After sending the BindConfirm message, the MN selected by the Sender becomes the root of the session tree. The Sender sends a session announcement, which includes a multicast session ID, multicast address and port number. The root MN is responsible for managing the whole session tree.

As shown in Figure 10, different senders can choose different MN as the root of its multicast session tree. This feature can avoid a single MN becoming the bottleneck of the multicast transport. Sender can also

choose another MN as the root of its session tree, if the previous root crashed. This feature can avoid potential single point failure of the root node. The detailed process of sender binding to the mesh will be discussed in section 3.2.2.



BR = BindRequest Message
BC = BindConfirm Message

Figure 11. Normal process of member join in Tree Auto-Configuration Algorithm

After the Mesh is constructed and sender binds to the Mesh, the tree construction starts with join request from a leaf node, a receiver. Each SN or MN will join a specific session tree before it can accept a session join request from a child.

A receiver that wants to join a multicast session broadcasts a BindRequest

message (message BR2 in Figure 11) on its LAN. There are three scenarios. The first scenario is: if there is a local service node (SN) of the desired multicast session on the LAN, it accepts this bind request and makes the receiver bind to itself.

Second: if the there is a SN on the LAN, but it is not on the session tree, it tries to find its closest SN or MN as its parent, sends a BindRequest message (as message BR3 in Figure 11) to its parent, and joins the session first. The SN uses a controlled expanding ring search algorithm (Controlled ERS) to find and bind to a parent. This algorithm will be discussed in section 3.2.4. After binding to the session tree, the local SN can accept the receiver as its child.

The third scenario is the most complicated one: if there is not a SN on the LAN, all receivers of all multicast sessions in this LAN should elect one of themselves as a local SN, and this SN joins every session tree first by using Controlled ERS. The SN election process is similar to the group controller selection and replacement process in Local Group based Multicast Protocol (LGMP), which is a group-based multicast protocol. The detailed process of a receiver joining the session tree will be discussed in section 3.2.3.

Any SN that is not on a specific session tree will not accept any other SN as its child, although it may be on the shortest path and be on another session tree. It can avoid unnecessary nodes joining the tree and reduce the delay between nodes on the session tree. When a SN node receives BindRequest, it holds the child's request and tries to bind to the tree first, if it is interested in this session but is not on the tree yet.

If an MN accepts BindRequest message from a child (a SN or an MN) and it is not on the session tree yet, it tries to bind itself to the "next-hop" MN that is in the forwarding table entry for the session tree root. It sends a BindRequest message (as message BR4 in Figure 10). The "next-hop" also tries to bind to its "next-hop" and so on. This process continues until this binding request reaches the root or an MN that is already on the tree. The root of MN on the session tree sends a BindConfirm message (as Message BC2 in figure 10) back to the requesting MN. This process can build a session tree on the mesh, which is rooted at the MN that binds to the Sender. After binding itself to the tree, MN sends RequestConfirm message to all its children and starts to receive multicast data.

The service node that is in a lower tree level joins the tree after receiving BindConfirm message (as message BC3 in Figure 10) from its parent. Then it accepts children, receivers or other SNs, by sending BindConfirm

message (as message BC4 in Figure 10) to all its children. When the process ends, the session tree is built.

The built session tree can be maintained and reconfigured automatically. The session tree can be dynamically and automatically rebuilt by following the process above, if some nodes join or leave the tree. The SNs and MNs periodically send a heartbeat message to inform their children that they are still functional. Children will respond to this Heartbeat if necessary. If the Heartbeat messages are not received in a specific interval, the children will restart the tree auto-configuration process to find another parent node. If the responses to the Heartbeat message are not received in a specific interval, the service node deletes the entry for the children in its child list. In multicast session, the children can send ACK or NAK for the received data packets to their parent, and parent aggregates these messages and sends aggregated ACK to upper tree level. If the quality of service is worse than a pre-assigned threshold, a SN can start the process above to find a better parent node. If the root is shut down, the sender can try to choose another MN as the root.

3.2 The process of the Tree Auto-Configuration algorithm

The process of tree Auto-Configuration algorithm in reliable multicast is

complicated and cannot be solved by only one method optimally. This section is to combine some effective algorithm and try to make all nodes in the session build a tree based on available information. The built session tree may not be the most optimal solution but a useful and manageable one. The dynamic rebuild mechanism can find a better solution according to the real network conditions.

3.2.1. Mesh construction.

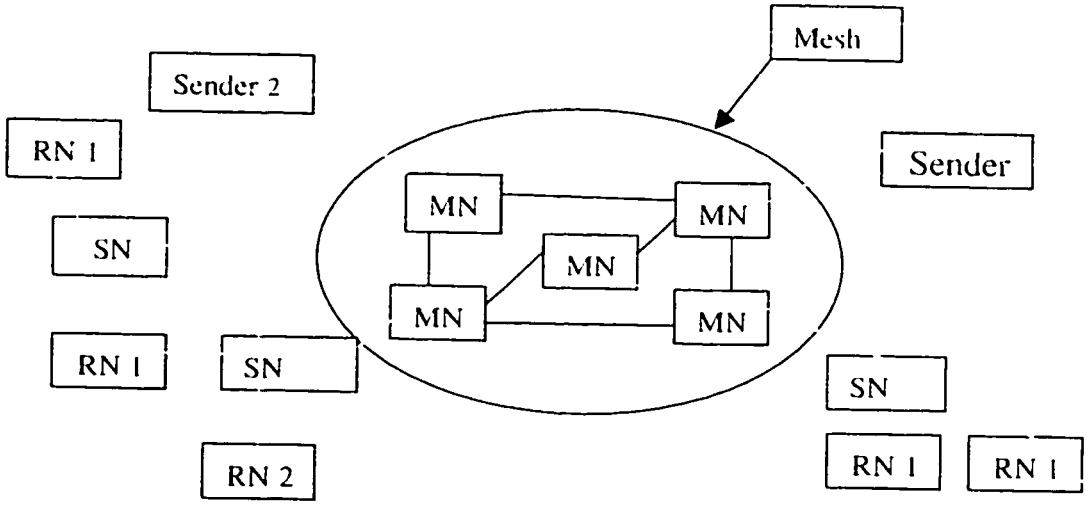


Figure 12. Mesh Construction.

The first step in the Tree Auto-Configuration is to build the mesh that is comprised of some pre-deployed service nodes. These nodes, MNs, can exchange forwarding messages with each other and build their forwarding tables when they are started up. Each MN maintains a forwarding table in which the information of "Next-hop" MNs to reach the destination MNs is stored. A given MN in the mesh can "broadcast"

information to all other MN on the mesh. The built Mesh is shown in Figure 12. Each MN also maintains a child list for a session. This list stores the children's IP address, binding status, child count, and other information. A MN periodically sends Heartbeat to its neighbors and its children and waits for responses from children.

There are two ways to build the mesh. The first one is a static way. This method assumes that all MNs are reliable, robust, and available forever. The topology of the Mesh never changes, and the connections have fixed distance values. If some SN shuts down, maintenance team can repair it as soon as possible. The Mesh will always provide reliable connections. The SNs know the existence of others and have pre-calculated forwarding tables. The ISP configures the forwarding table entries carefully and administratively. The forwarding tables of all MNs must not form a loop in the Mesh.

The static method is easily to implement: ISP can put forwarding table in a specific file. MN can access this file when it starts up. It does not need complicated routing-like algorithm to set up forwarding table at the beginning or to refresh forwarding table later. However, it requires that information about the network connections must be known before the mesh is built, and the information always exactly reflects the condition of

underlying network. It has no automatic fault tolerant and dynamic re-configuration capabilities. If some MNs must connect to root via a specific MN, the single point failure of the intermediate MN will prevent those MNs from continuing their multicast communication. It has no adaptability for growth. If another router wants to join the Mesh, the ISP must build a forwarding entry for the new MN in all other MN's forwarding tables. This feature will hinder the scalability of the Mesh approach.

Another way is a dynamic way. In this method, MN needs a dynamic method to set up and refresh the forwarding tables according to the real condition of the underlying network.

In the dynamic way, a new MN does not know the existence of others at the beginning. It may contact with a known MN or a designated node to get information of other nodes on the mesh. The MN or the designated node, called *Point of Contact* (POC), sends information of all available MNs to the new MN. POC method will be discussed later. After knowing all the MNs, the new MN tries to find its neighbors, build a forwarding table, and join the mesh by binding to its neighbors.

It is the first thing for a new MN to do that it should learn who its

neighbors are. Because this algorithm is built in transport layer and the routing function locates in network layer, the algorithm should use some underlying network layer protocol services to find neighboring MNs. The network layer service is out of the scope of this thesis, so we just make some proposals below.

We can use unicast routing information to build forwarding tables, the same way as in PIM-DM/SM. The MN may check the path to all other MNs according to the unicast routing information. It regards the closest MNs on each path it finds as its neighbor and the next-hops to the destinations. Then the MN tries to bind to its neighbors by sending BindRequest message and waiting for BindConfirm message like other kinds of nodes do. After binding to its neighboring MN, it measures the distance to its neighbors, exchanges reachability information with neighbors, and finds the shortest path to all other MNs.

Some routing-like algorithm can also be used by MNs to exchange reachability information and to find shortest paths. Link State Routing, and hierarchical routing algorithms can be used here. The Link State Routing algorithm has five steps to build the routing table: discover its neighbors and learn their network addresses, measure the delay or cost to each of its neighbors, construct a packet telling all it has just learned,

sending this packet to all other routers, and compute the shortest path to every other router [10]. Because the MNs may not be directly connected to each other, the new MN has to use the method mentioned above to find its neighbors. MN measures the distance to its neighbors by a metric. The metric that is used to measure distance can be hop-count, delay, or static parameters. Then it sends the link state packet to all other MNs. When all MNs get the link state packet, they can compute the shortest path by using Dijkstra's algorithm and refresh their forwarding tables. At a certain point the network may grow to the point where it is no longer feasible for every MN to have an entry for every other MN, and MN joining or leaving events may occur frequently. So the routing will have to be done hierarchically, as it is in the telephone network [9]. Each MN knows all detailed information about how to route packets within its own domain, but it knows nothing about the internal structure with other domains. When different domain are connected together, it is natural to regard each one as a separate domain in order to free the MNs from having to know the topological structures of other domains.

The dynamic way has some advantages and disadvantages. It can allow new MN to join or leave the mesh dynamically after the mesh is built, and allow the mesh to rebuild itself according to the membership changes. It has fault tolerant ability. Each MN may periodically send a heartbeat

message to its neighbors. If an MN did not receive heartbeat from one of its neighbors within a specific interval, it will know its neighbor is unavailable. Then it will initiate a process described above to let itself and others MNs refresh the forwarding tables. The Mesh is rebuilt when all MNs complete the refreshing routing computation. This method is complex and difficult to implement. When the mesh grows to a very large scale and node join/leave events occur too frequently, the time taken to compute and refresh the forwarding table can be very long, and forwarding information stored in each MN can be very large. The information of all MNs may not always be available and correct. Any invalid forwarding entry will prevent the Mesh from working properly. The difficulty and complexity of the dynamic way will damage its scalability.

As the infrastructure of the multicast transport, the mesh can provide intra-domain and inter-domain capability. In a specific domain, a set of MNs can construct the backbone of the sub-tree within the domain. All other service nodes and receivers only need to know the session information and send explicit join requests to the session tree. The MN can accept the session tree join requests and forward data packets downstream to all its children. The MN can cache the received data packets for potential retransmission. The MN can aggregate TRACKs

from its children. These TRACKs can be processed on MN or be forwarded to the nodes on upper levels. The multicast membership management and sub-tree configuration can also be done within the domain to reduce control information generation. The basic intra-domain topology is shown in Figure 13.

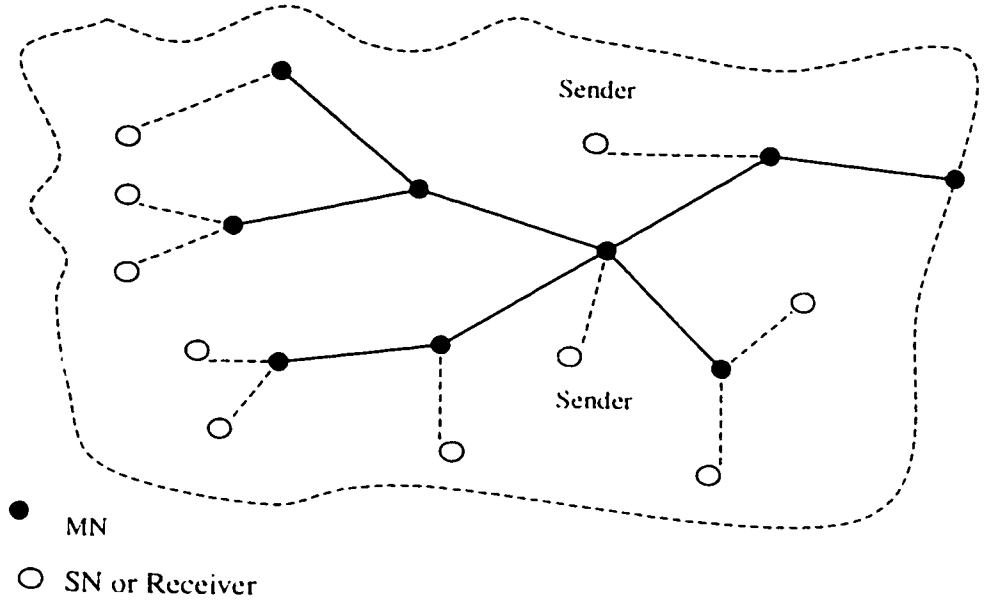


Figure 13. Intra-domain topology of Mesh

In Figure 13, the Mesh provides a backbone for the multicast transport. In the mesh, unicast control information path and multicast path can use the same topology as the underlying physical network, if we only choose multicast routers and native multicast links to build the mesh. This feature can reduce the complexity of using different multicast data channel and unicast control channel as in RMTP-II. All MNs can be chosen as a root node. So if the root node of a session tree fails or the quality of service is below a minimum level, the sender can easily choose another MN within

the same domain. There is no need to configure a backup root node. If there is more than one sender, each router can choose its own root node in the mesh, as shown in Figure 13. A MN can be root for more than one multicast session. The feature of supporting multiple sessions can avoid the problem in some core-based multicast protocols that the core can be overburdened if there is more than one session tree rooted at the core. The topology is also a good infrastructure for future many-to-many multicast communications.

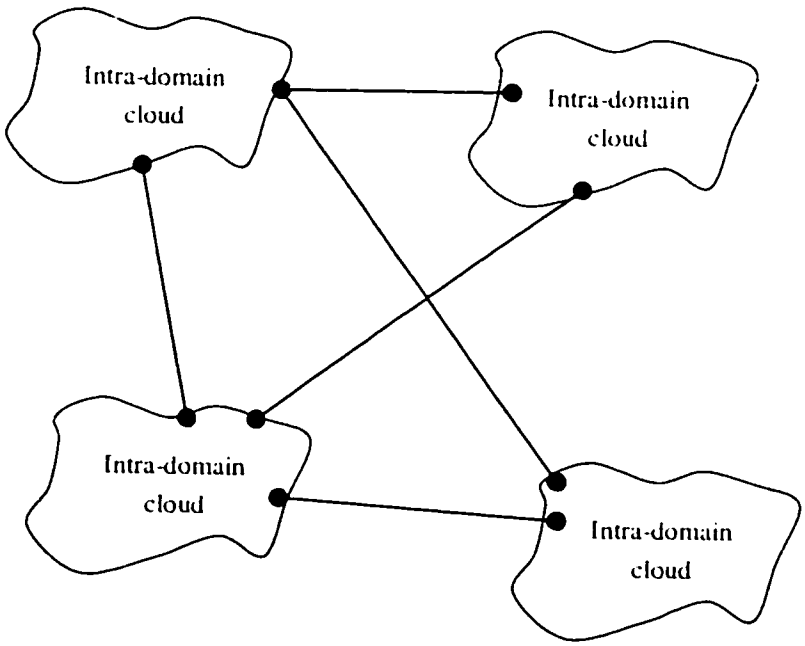


Figure 14. Inter-domain topology of Mesh

The Mesh approach can also provide inter-domain capability. In Figure 14, the Mesh has inter-domain connections. If we configure a border router in a domain as an MN, this MN can easily use the connection

between autonomous systems (ASes) and use BGP to compute inter-domain forwarding route. The forwarding route computation is simplified in the session tree construction process when a sub-tree is built on the Mesh. This feature can solve the problem of connecting sources and receivers across two sparse mode domains in Intra-domain multicast protocols. The Mesh is constructed before the session tree, and the inter-domain routing information is stored in forwarding tables in each MN. Because the MNs can "broadcast" session announcements to each other, a receiver does not need to know the location of sender. The receiver just gets the multicast session ID, sender's multicast address, and port number from the sender's multicast session announcement and sends a join request to local service node. The service node initiates the join request and binds to the mesh directly or indirectly, if one of its children is interested in a session. The Mesh can find out the sources in other domains according to pre-computed forwarding information. The mesh can also solve the join latency and bursty source problems described in section 2.2.2, because the MN has the pre-computed routing information and can cache data packets for necessary retransmission.

3.2.2. Sender joins the session tree.

After the mesh is constructed, the actual first step in session tree

auto-configuration begins at the sender node. Let us continue our discussion based on the basic scenario in Figure 10.

When a Sender wants to start a new multicast session, it tries to find the closest MN by asking the POC or just contacting an MN assigned by the ISP. Then Sender sends a BindRequest message to the selected MN. This message indicates that the requesting node is a Sender. The MN sends back a BindConfirm message telling the sender about the acceptance. The MN that accepts the Sender becomes the root of the session tree. The root is the only node in the tree that can communicate with the sender. It receives the data packets and forwards to all other MNs and receivers. It can aggregate TRACKs from the tree and send an aggregated TRACK to the Sender, if necessary. This feature can allow a source that lacks multicast capacity to start a multicast session. The root is responsible for managing the tree construction and membership. Different senders can use the Mesh to build their own session trees. A MN can be the root for multiple session trees.

After being accepted by an MN, the Sender sends a Session Announcement to the root. This Session Announcement should contain session ID, multicast address, port number, and other information about the multicast session. There is a standard Session Announcement protocol

[12]. In this way, a Receiver discovers the multicast group address, the Sender's address, and other information necessary for logical tree construction. Sessions may be announced in two parts, the first part containing generic information about the session, such as the multicast address, and the second part, announced on the multicast address, containing additional information [12].

The root will send this session announcement to all MNs. All MNs will maintain a session table to store information about each session on the tree. The MNs use the session ID to identify a specific session and the request for joining or leaving a specific multicast session.

3.2.3. Receivers join the session tree.

Receivers of a session use a standard out-of-band mechanism for discovery of a session's existence (e.g., Session Advertisement [12], URL, etc). This out-of-band session announcement is out of the scope of this thesis.

Before any MN, SN, or receiver joins the session tree, the tree consists of only one Sender and a root MN. The original join request is initiated at a leaf of the session tree, a receiver. The SN may join the session tree on

behalf of a child.

A receiver that is interested in the specific session should join the local group first, before it can receive any multicast data packet. Because there may be thousands of receivers on the Internet, they should be divided into local groups and choose a local SN. Division of local multicast group can avoid a very large number of control messages and data packets being transferred on the Internet. The local group controller, a SN, can do a lot of tree maintenance and data retransmission within the LAN. In this way, it can enhance the efficiency of multicast significantly.

A receiver broadcasts a BindRequest message on its local network. If there is a SN on this LAN, it responds to the BindRequest message. If the SN is already on the session tree and can accept one more child, it sends back a BindConfirm message, makes the receiver bind to itself, and builds an entry in its child list for this new receiver. If it is not on the session tree yet, the SN sends a BindACK to the receiver. This message causes the receiver to wait for SN to process its request. The SN sends a BindRequest to its parent and waits for the response.

If there is no service node on the LAN, all the receivers in this LAN should elect one of themselves as a local SN. The SN joins the session

tree first. After sending BindRequest on LAN, if receiver does not receive BindConfirm or BindACK message after a specific interval, the receiver knows there is no SN on the LAN and initiates a SN election process.

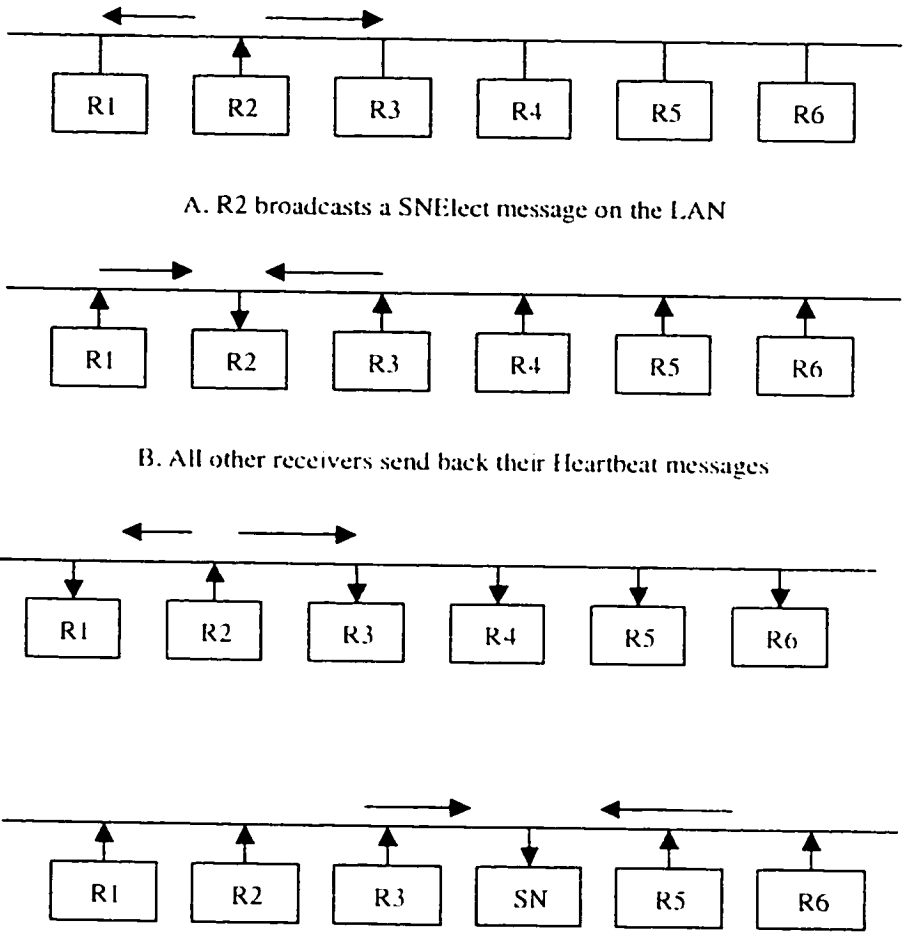


Figure 15. Local SN election.

The receiver broadcasts a SNElect message on the LAN. Any receiver on the LAN should respond to this message by sending back its IP address and other information in a Heartbeat message. The receiver who initiates

the SN election will choose a node as the new group controller, SN, according to the node capacity, CPU speed, or IP address. After choosing the new SN, the receiver broadcasts the information about the new SN in an SNConfirm message. The new SN will modify its own property from a receiver to a SN. Other receivers will send join requests to this new SN. The SN election process is shown in Figure 15.

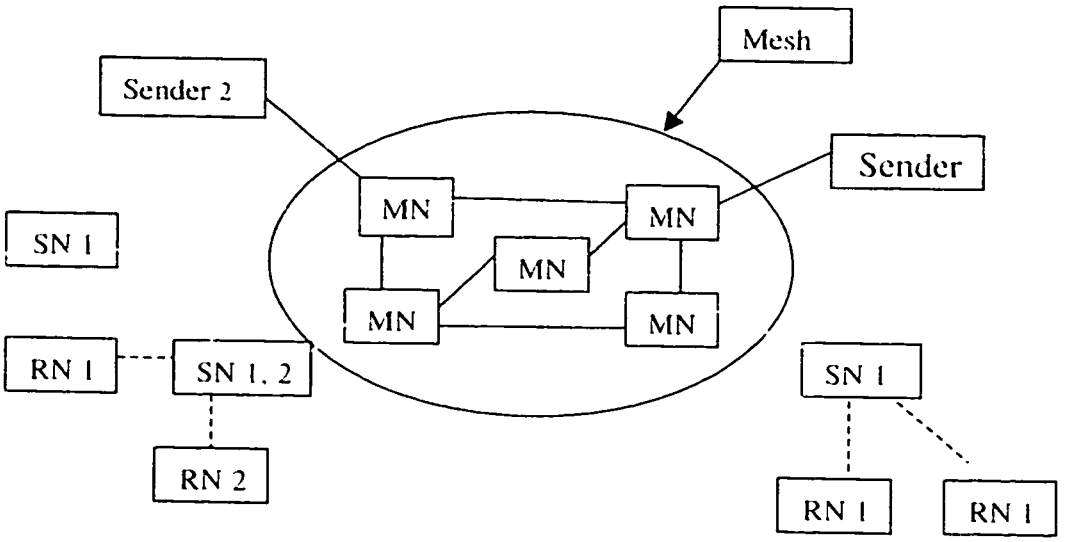


Figure 16. Receivers join the session tree.

The SN election process is also useful when a local SN crashes. It can be essential to local group management. To obtain the optimal multicast capability, we can configure the local router with an interface to the Internet as the local SN. However, it is not necessary. In the best condition, any node on the LAN can be the local group SN. Figure 16 shows the result of this step. The dashed line is the path of the

BindRequest.

3.2.4. Local SNs join the session tree.

The local SN should join the session tree before accepting any receiver's bind request. The SN can bind itself to an MN or an intermediate SN that is on the session tree and can accept another child. To find the shortest path to the root, the SN has to measure the distance of each possible path and choose the best one. Figure 17 shows the result of this step. The dashed lines show the paths of BindRequests.

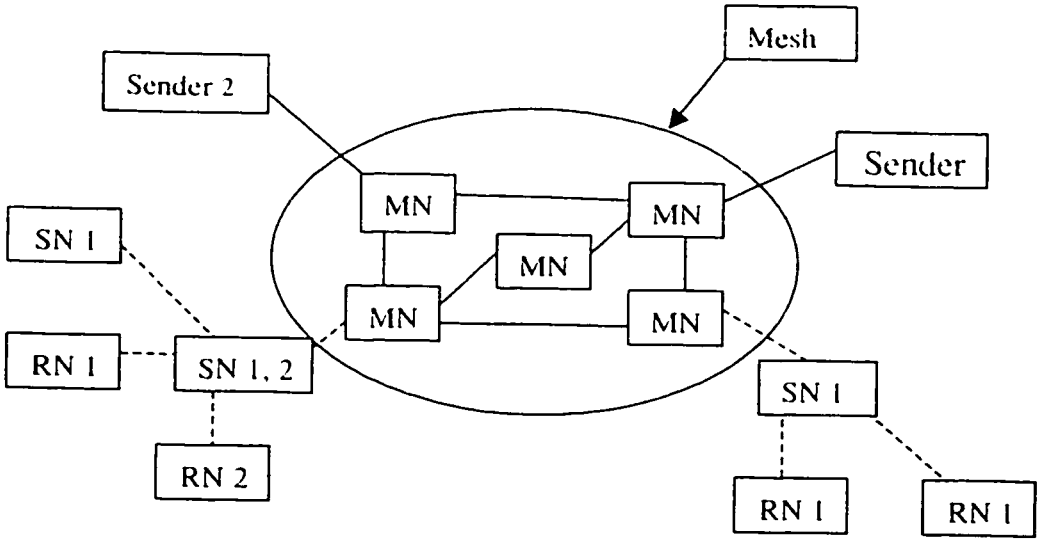


Figure 17. Local SNs join the session tree.

The most important and difficult function of a local SN is neighboring node discovery and selection. The local SN finds out some MNs, tries to measure the distance of each path to these MNs, and binds to the chosen

parent node.

There are two ways to find the closest neighboring SN. The POC method can be used here. The local SN queries POC for possible closest MN. The POC recommends some MNs to the local SN. The local SN sends a Query message to all recommended MNs to find out the closest neighbor. All SNs or MN that receive this Query message and have capability for another binding should reply to it. The local SN that sends Query message chooses the closest neighbor according to all reply messages it received. If there is no response from other nodes, the local SN should ask the POC for information about more SNs in the mesh.

Another way is to use a controlled expanding ring search algorithm. The local SN tries to trace the IP route to root by a function like the *traceroute* program. Such a function can find all the intermediate routers on the path. Then the SN sends the Query messages to all the routers, with a specific TTL (time to live) value, and wait for the reply in a specific interval, SolicitPeriod. If there is one or more reply from those routers within a SolicitPeriod, the SN figures out the round trip time of the message between the routers and itself and chooses the closest node as its parent. If there is no reply at all, the SN will increase the TTL and query for parent again. This process ends when at least one reply has been received

or the TTL becomes greater than a maximum TTL, TTLMax. If TTL is greater than the TTLMax, the binding has failed and the local SN will inform all receivers about the result.

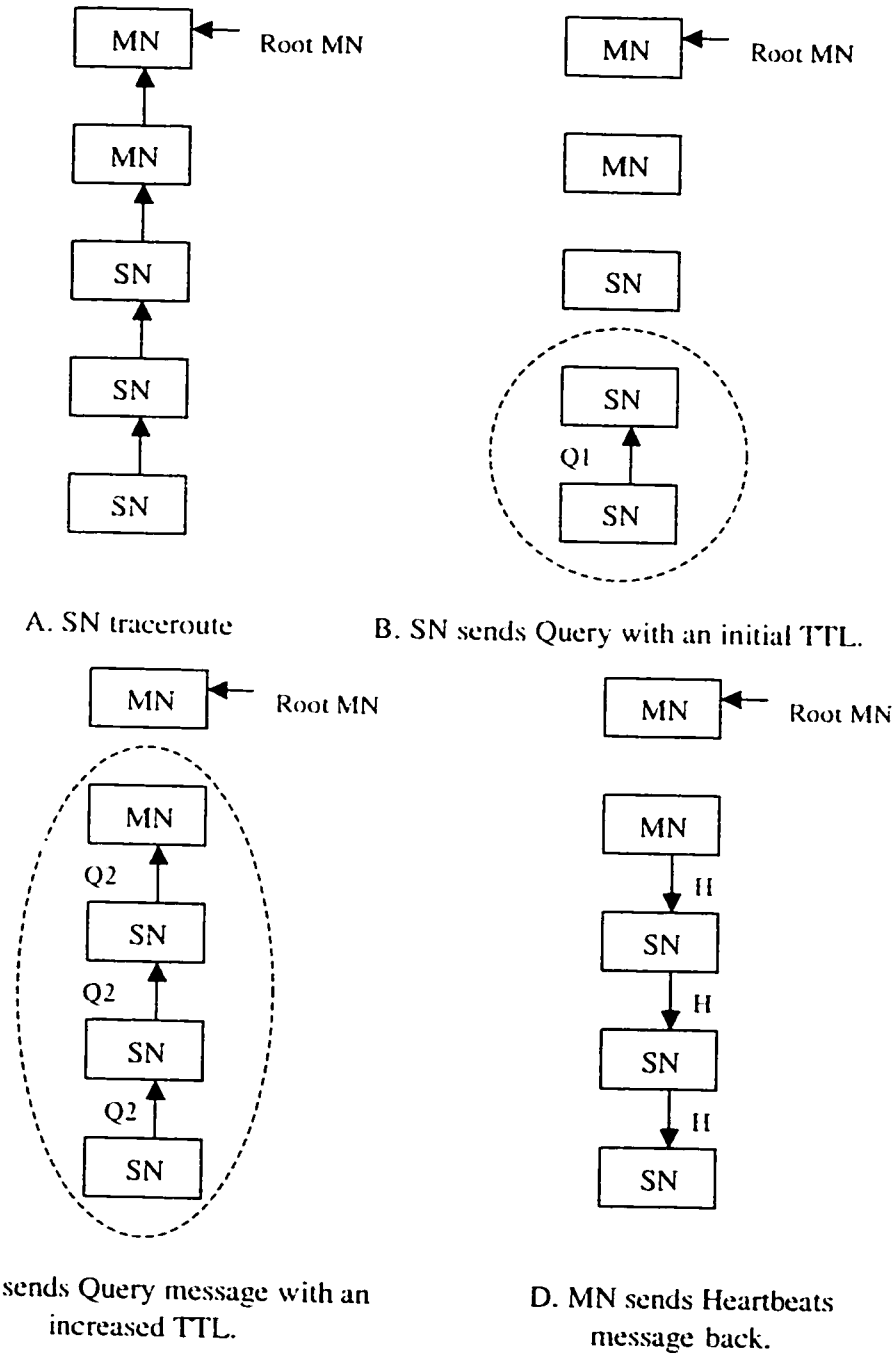


Figure 18. An example of Controlled ERS algorithm.

There is an example of controlled Expanding Ring Search algorithm in Figure 18. In Figure 18A, a new SN finds out all routers on the path to the root by the traceroute function. Then in Figure 18B, the SN sends a Query (Q1) message with an initial TTL to the routers it found in the first step. The dotted circle is the search ring that a message with such a TTL can reach.

We assume that there is no reply at all after a SolicitPeriod. So, in Figure 18C, the SN sends Query (Q2) messages with an increased TTL. The search ring is enlarged. If there is a service node (MN or SN) that can accept another child and is within the search ring, a Heartbeat (H) is sent back to the SN, as in Figure 18D.

This algorithm has some advantages over the expanding ring search (ERS) algorithm recommended in the IETF draft, Reliable Multicast Transport Building Block: Tree Auto-Configuration. In the IETF draft, the new node sends Query messages in the multicast channel. ERS floods the query message over the whole multicast tree. Expanding ring search (ERS) is an effective technique in a local subnet or intranet (especially when the IP multicast routing protocol is dense-mode based). On the other hand, it is not practical in a multi-domain network. It is not effective when the routing protocol is sparse-mode based. It can add

significant control traffic overhead [6].

The controlled ERS algorithm queries only the nodes that are on the path to the root. Other nodes will not be involved in this process. This feature can significantly avoid unnecessary control messages. It can also avoid some inefficient tree branch as created by the ERS algorithm. We can see an example in Figure 19. We have seen a sample MBone multicast network with a combination of IP-encapsulated tunnels and native multicast link in Figure 2.

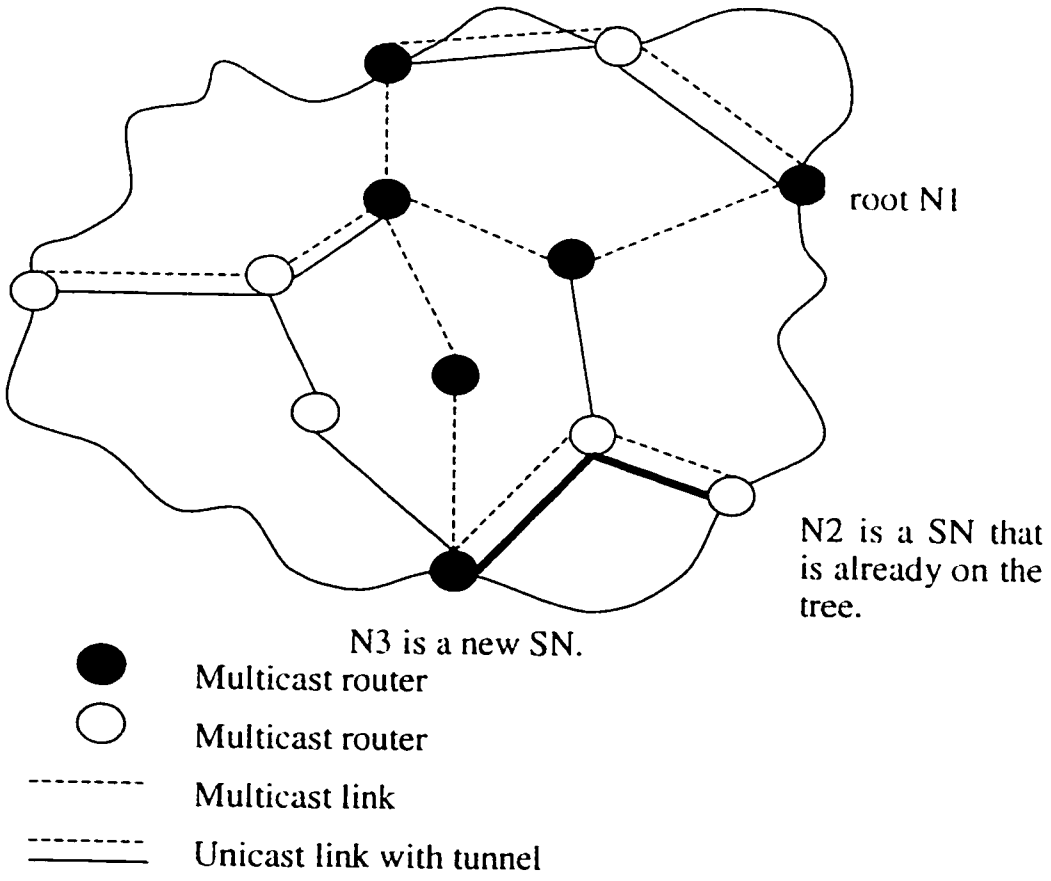


Figure 19. An example of inefficient tree branch created by ERS.

In this topology, we assume that there is a multicast tree rooted at N1 and there is already a service node on the tree, N2. When a new service node, N3, wants to join a tree, it uses ERS and multicasts Query message with an initial TTL. If the TTL is not long enough to allow Query message to reach the root, but it is long enough to the reach N2, the new SN may consider N2 as the best parent candidate and bind to it. Clearly, N2 is not the best choice and it even needs to get multicast data via N3. The controlled ERS algorithm can avoid such inefficient connections by only querying routers on the shortest path to the root.

After the parent node is chosen, the SN node sends a BindRequest to the parent. It sends BindConfirm message to its children after receiving a BindConfirm message from its parent and binding to the parent. If it receives a BindReject message, it tries to find another parent and binds to it, following the process above. It maintains a child list that contains information about its children. If it accepts a new child, it makes a new entry in its child list. It deletes a child entry if this child leaves the session tree.

Each SN maintains a child list for a session. This list stores the IP address, child count, and other information. The local SN periodically sends

Heartbeat to its parent and all its children to inform them of its existence. If a SN crashed, its parent node will delete the entry for this node in the child list and its children will try to bind to another SN.

A SN can leave a session only when all its children have left the session tree. The SN sends a LeaveRequest message to its parent, and waits for the LeaveConfirm message.

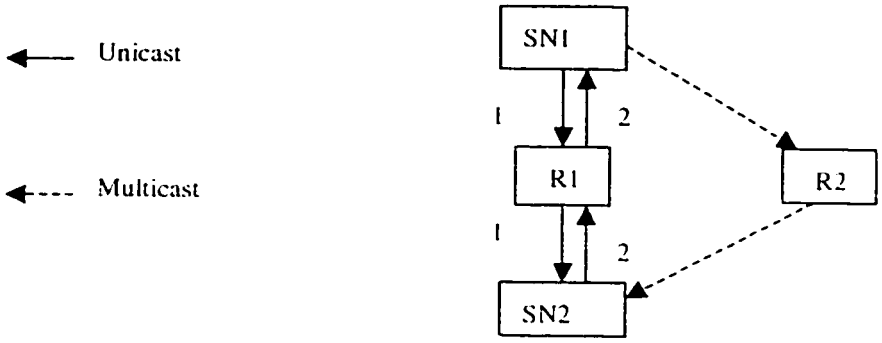


Figure 20. Unicast and Multicast have different paths.

In this step, the unicast control information path from a SN to its parent can be different from the multicast path. There is an example in Figure 20. The tree links between service nodes are logical links in the transport layer. The service node will maintain the connection state of both unicast and multicast. The SN1 can still build an entry for SN2 in its child list. However, the entry should include two fields indicating next-hops information of unicast and multicast. In this cast, the next-hop for unicast is R1, and next-hop for multicast is R2. The SN2 just has the IP address

of SN1. The SN2 will send control information (e.g., BindRequest, LeaveRequest, or TRACK) via R1. SN1 sends multicast data packet and retransmission packet via R2.

3.2.5. Session sub-tree on the mesh.

The MN that accepts BindRequest messages should try to bind to its "next-hop" MN, if it is not a node on the session tree yet. The mesh approach uses the next-hop neighbor as the selection for the best neighbor to bind to for MN-MN bindings. MN may accept a node as a child, as long as selecting it would not cause a loop in the tree. This process ends when a MN on the session or the root accepts a bind request. After the tree on the mesh is built, all MN nodes should respond to the BindRequests that they received. Then the whole session tree is built.

The result of this step is shown in Figure 9. The solid lines show the binding built in the tree auto-configuration process. The dashed lines are connections that are not used in the session tree.

3.3. Tree Formation.

This section is a detailed description of the tree formation process and

tree level computation defined in IETF Internet-Draft, Reliable Multicast Transport Building Block: Tree Auto-Configuration [6]. It defines the Sender's tree level as 0, the tree level of session tree's root MN as 1, the child's tree level is one more than its parent's tree level. When a node is not connected to the tree yet, it has a tree level value greater or equal to 128. The reason for reserving part of the space (of tree levels) for indicating off-tree status is that a special technique can be used to prevent forming loops. The largest value is 255, so the range of off-tree levels is in the range 128 - 255. At the beginning, any new node assumes that its tree level is 128.

Because an MN selects and binds to the next-hop in its forwarding entry for root, and the next-hop node is on the shortest path to the root according to the Mesh Construction mechanism described in section 3.2.1, we can be sure about a fact that there is no loop in the sub-tree on Mesh.

However, the situation in lower level sub-tree is more complex. Once a SN has been selected as a service node's (SN) parent, the node sends a BindRequest message to the selected SN. In the BindRequest message, the requesting node should inform the selected node of its current tree level and number of children.

If the SN has an outstanding request to bind to another service node, it must refuse the incoming bind request in case it would form a loop. Otherwise, it may accept the node as a child, as long as selecting it would not cause a loop in the tree. Loop freedom is guaranteed by these two rules [5]:

1. If the requesting node does not have children, the SN can accept it as a child as long as the SN has no outstanding bind requests. If it does have an outstanding binding request, the SN can accept the node as a child if its tree level is less than the child's tree level.

2. If the requesting node has children, the SN can accept it as a child if
   a. The SN's level is 128, i.e., it is the top of a sub-tree not yet connected to the Sender, or
   b. The SN's level is less than 128, i.e., it is connected to the session tree.

The second rule prevents a node from selecting one of its own children as its parent. Two nodes at level 128 are prevented from selecting each other using the tie-breaking criteria described step 1 above.

If the node is not connected to the session tree and accepts a child, it

sends a BindACK message to its child. The parent should include its current tree level in the BindACK message. The child node should change its tree level to parent's tree level plus one. If the child does have its own children, it informs its children about the change in its tree level. Its children will change their own tree level value accordingly.

After being connected to the session tree, a requesting node will get its parent's tree level in the received BindConfirm message from its parent, change its tree level to parent's tree level plus one, and inform its children about this change in its BindConfirm message. This process ends when a leaf node (receiver) corrects its tree level accordingly.

We can see an example of this process in Figure 21. At time 1 in Figure 21A, a receiver R1 sends BindRequest to SN1. At this time, SN1's tree level is 128. Because R1 is a receiver and does not have any children, SN1 accepts it as a child and send the BindACK to R1. R1 changes its tree level to 129 as shown in Figure 21B. We assume that SN2 is interested in this session and is not connected to session tree yet (with tree level 128). If SN2 is on the shortest path from SN1 to the root, SN1 will choose SN2 as its parent. So at time 2, SN1 sends BindRequest to SN2. If SN2 has no outstanding request, it can accept SN1 as a child. Otherwise, it has to refuse SN1's request. We assume that SN2 does not

have an outstanding request at this time. So SN sends a BindACK to SN1. SN1 will change its tree level to 129 and inform R1 of this change in a BindACK message. R1's tree level is changed to 130 as shown in Figure 21C. Then SN2 sends a BindRequest at time 3, Figure 21C. Figure 21D shows the result that all nodes are connected to the session tree.

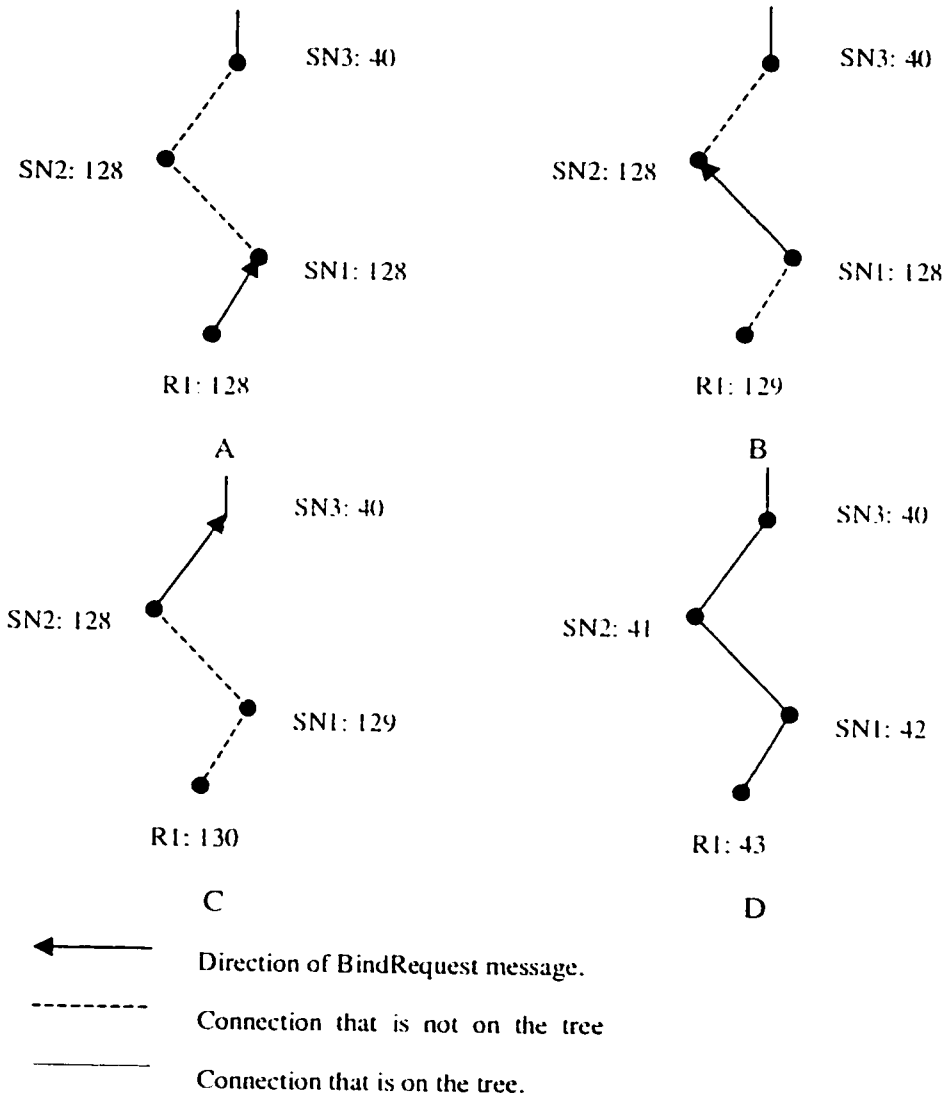


Figure 21. Tree Formation.

If an SN cannot accept a requesting node as its child, it sends a BindReject message to the requesting node. In the BindRequest message, SN should indicate the reason for rejecting this bind request. The requesting node will try to select another node as its parent and try to bind to it.

There is another reason that a SN rejects a child, except for forming a loop on the tree. SN may limit the number of children they support depending on their capacity. Once an SN has accepted its maximum number of children, MAXChildren, it stops accepting new children until a change in membership that causes its count of children to go below this limit.

Chapter 4
Implementation of Tree Auto-Configuration Algorithm.

In this chapter, an implementation of Tree Auto-configuration algorithm will be given. This implementation is based on a protocol base class library, Meta-Transport Library, designed by Infrastructure and Networking Research, Sandia National Laboratories, Livermore, California. The system state representation and architecture will be given.

4.1. Meta-Transport Library (MTL)

The Meta-Transport Library (MTL) is a set of C++ base classes designed to present an infrastructure for building transport protocols. The classes represent the necessary protocol components, and the member variables and functions of these classes represent the state each component must keep and the work each component must do [10]. The current MTL version number is 1.5.1.

The goal of MTL is to allow transport protocol designers to prototype a protocol rapidly. The designer does not need to modify any kernel, or

support special hardware. The MTL also allows the designer to reduce use of root privilege as much as possible. So MTL has these design characteristics: portability, adaptability, configurability, and readability [10].

Portability: MTL has been ported to most major Unix varieties, including SGI IRIX, Sun SunOS and Solaris, HP HP-UX, DEC Ultrix and OSF/1, IBM AIX, FreeBSD, Linux, and BSDI BSD/OS [10]. This feature allows an MTL-derived protocol to be easily ported to all those platforms.

Adaptability: The modularity of the MTL design allows for easy replacement of the underlying data delivery service [10]. MTL has two different delivery service modules for IP and UDP. The module for IP needs some root privilege, but the module for UDP does not. It is easy to build modules for other services, such as a connection-oriented network service. This feature allows the derived transport protocol to run over a variety of networking technologies.

Configurability: In addition to changing the underlying data delivery service, replacement of various protocol control algorithms is easily done when the protocol implementation is modular [10].

Readability: This approach is designed to enhance the understanding of how protocol components interoperate, and to allow a designer to replace components without undo effort. C++ separates interfaces from implementations, and encapsulates concepts into modules. This makes the implementation process both easier to understand and easier to

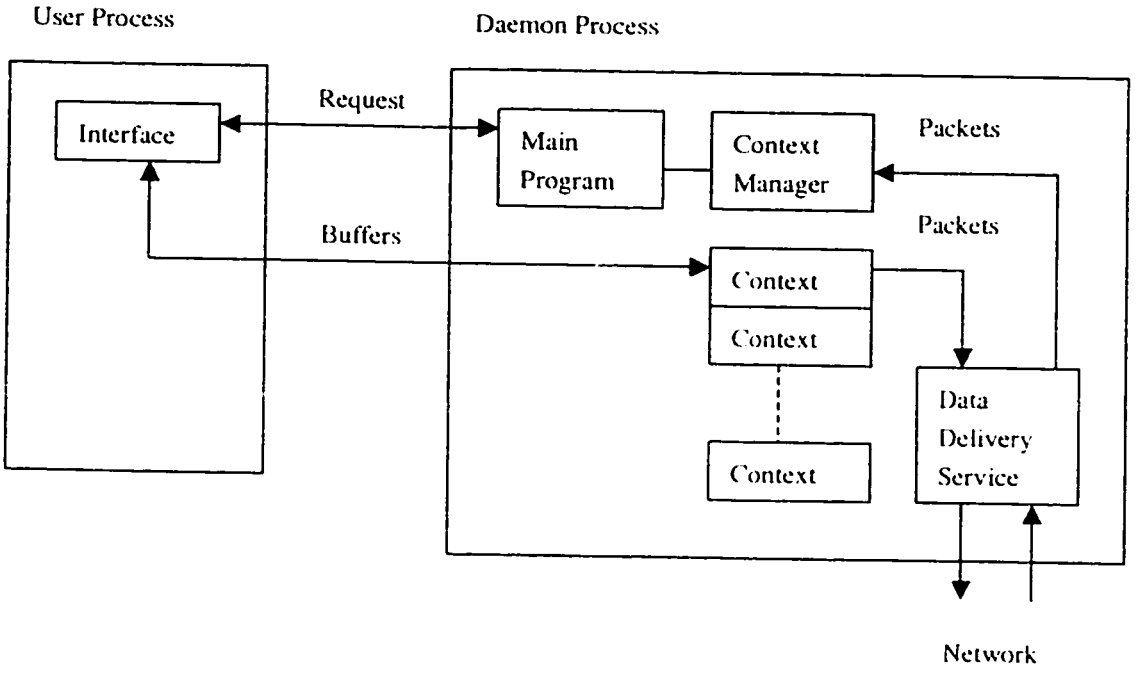


Figure 22. MTL User/Daemon Model.

manipulate [10].

In general, a transport protocol has five components: packets, data delivery service, context, context manager, and user interface. A transport protocol implementation sends and receives packets via the use of some services of the underlying data delivery service. The state of the communication is maintained by some context. A context manager is an

agent that de-multiplexes incoming packets and incoming user requests, delivering these to the proper contexts. The user interface is the abstraction through which access to the functionality of the protocol is granted to the user.

Figure 22 shows the general MTL model. A MTL derived protocol usually has at least a daemon process, a context manager object, an interface object, a data delivery object, and several context objects. An MTL-derived protocol implementation is instantiated as a user-space daemon process. A user process uses the interface object to send requests to the daemon process via an IPC facility, which is a facility that enables the daemon process to communicate with its users. The main loop of the daemon process accepts a user request and uses the context manager object to direct the request to the proper context object. Some of the requests may cause the context to generate packets. These packets are constructed and sent through the data delivery service object to underlying network. The daemon process sends the result of a user request via IPC back to the interface object as well. The daemon also listens for the incoming packets, and uses the context manager objects to steer them to the proper contexts for processing.

There are six main classes within the MTL library package, five of which

correspond to the main abstractions named above: a data delivery service class, a packet class, a context class, a context manager class, and a user interface class. The sixth class is a daemon class that wraps everything into an entity that can be handled by the operating system. Each of these classes except for the data delivery service class is designed to be a base

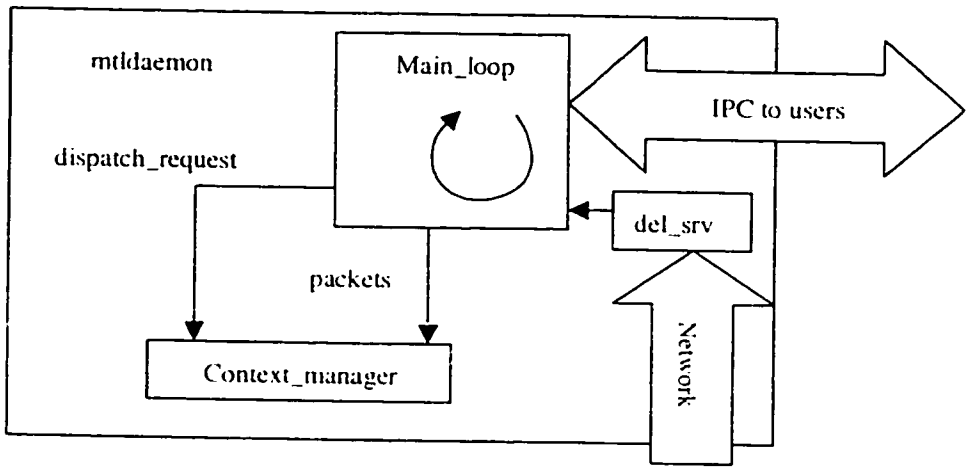


Figure 23. Main Loop of mtldaemon.

class for a protocol-specific class.

The main program of the protocol implementation initiates a global daemon object through which the protocol processing is conducted. The basic daemon class, mtldaemon, provides some fundamental functionality and access to several objects used throughout the system.

A daemon must determine if another daemon is running in order to avoid colliding with some of its IPC mechanisms, parse the arguments given to

the daemon process, initialize the daemon, start the main loop of getting packets and user request and processing them, and shutting down when is told to do so.

A context is the collection of all state information for an endpoint of an association. Certain state information is common to all transport protocols, and MTL reflects this in the context base class. Much of the information is just clerical such as identification and priority values. The context class provides access to these variables and reference through member functions. The context also holds the state of communication. Most of this information is protocol-specific.

The context manager manages contexts. The context manager class is a container class for all of the contexts in the system. It has a linked list linking all context objects, which allows them to be manipulated easily. The main purpose of the context manager is to match user requests and incoming packets to the appropriate context, so that the contexts can do the necessary protocol processing. To aid in manipulating the contexts, the context manager is a friend class to the context class. A context is identified by its key value. The key value is a 64-bit handle by which the context manager can find the context among all contexts in the system.

Half of the context manager's job is to dispatch user requests; the other half is steering incoming packets to their contexts. Both jobs require matching the key value and the proper context. For a user request, this is straightforward: MTL dictates that all user requests have the same header format and include the key value in this header.

There is a command value in *cmd* field of the user request header format. This command value is protocol-specific, and identifies what the request is. For example, the REGISTER user command instructs the user request dispatcher to initialize a new context for use by a user. The user sends the request via the user interface; the protocol-specific dispatcher must call a virtual function init_context() with the user request and the user's address for sending results and information back to the user. The *key* field is used to match this request to the appropriate context. The *upid* value is the process identifier for the user process. The *result-code* is either one of the command error codes defined in MTL or an addition error code defined in the protocol-specific type files. The *snd_shmid* and *rcv_shmid* values are used for setting up shared send and receive buffers between the daemon and the user, and *snd_buf_size* and *rcv_buf_size* are the sizes of these buffers. The *len* field holds the total length of this user request.

After initializing a context, placing a context on the context manager's active list puts the context into the processing stream. The context manager goes through each of the active contexts cyclically allowing them to do work. In MTL, an active context is just one that is on the active list, the derived protocol's definitions of states notwithstanding.

The context becomes the user's representative during communication, so it must hold information about the user process. The user process ID and the user's return address are part of this information. When a user requests that a context be initialized, the user PID (from the user request structure) is installed into the context. The return address is obtained from the IPC facility, and is also installed into the context. Knowing the user PID allows the derived protocol to send the process a signal when some event happens, if the daemon has sufficient permission. While the user makes requests via an IPC mechanism hidden by the user interface, the second way for exchanging information between the context and the user is via data buffers. Both user and the context hold two buffer managers each, one for the send buffer and one for receive buffer. When the user sends the request to initialize a context, the context establishes two buffers according to the size given in the user request. The context can block a user request. According to the semantics of the user request, the context may not respond to the user request with a result or other

information immediately, but rather may hold the user's request until some condition is met.

The context manager has a virtual function *handle_new_packet* as an entry point to specific protocol's packet handling code. The simplest thing to do is to put the packet on a packet FIFO queue and deliver it to the proper context later. If the packet's destination context is easily discovered, this function may put the packet on the destination context's received packet FIFO queue. The context manager also has a virtual function to be used when the daemon is shutting down. This call should return any resources that the context manager gathered, and possibly notify each user that the daemon is shutting down.

All other classes are designed to be compiled directly into the program that implements the protocol. The user interface class is targeted to be part of a library that will be compiled into the user's application code. The user instantiates the interface object; it provides the user's application with a means of issuing requests to the daemon and managing the data in the user's send and receive buffers. Two user requests are essential: registering with the daemon, and release from the daemon. The user application sends a request with a REGISTER command value to register with the daemon, and RELEASE to release

from the daemon. The user interface controls the user's end of the two buffer managers. The user writes data into these buffers and issues the send command. The presence of data is made known to the context, and the context's buffer manager pulls the data into the protocol. As the user application asks for data through the receive request, the context informs the interface's receive buffer about the size and location of the received

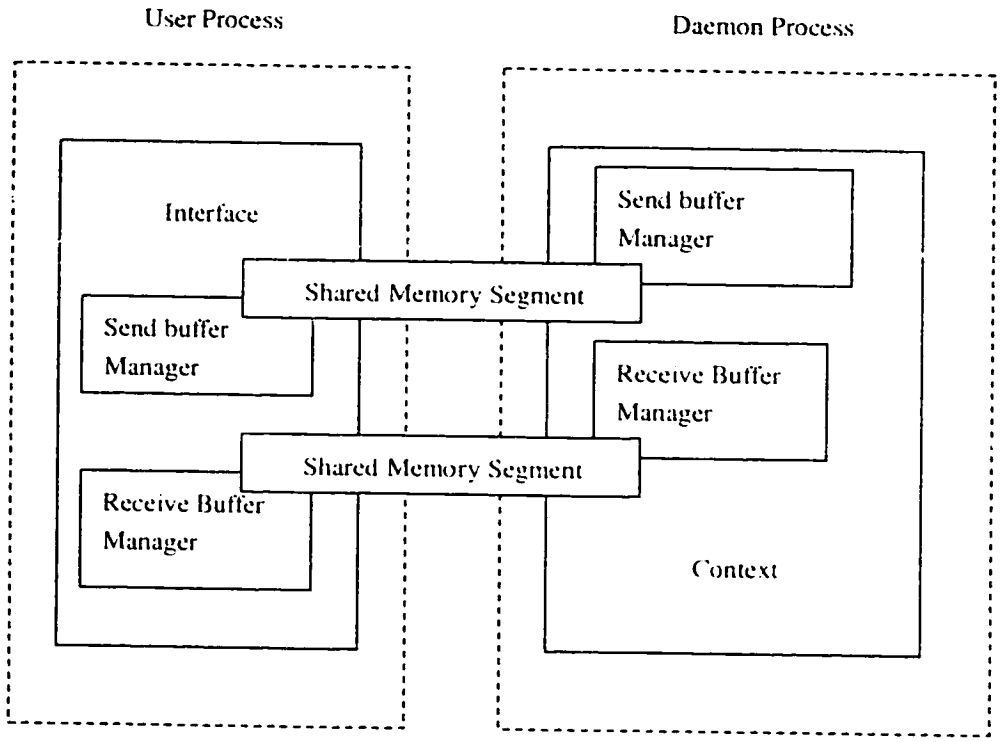


Figure 24. Buffer Manager

data.

Each user has two buffers, one for sending and one for receiving, as shown in Figure 24. A context has two buffer manager objects. Each buffer manager object controls each of the user's buffers. Data are

written into and read from the buffers through the buffer manager interface routines. A shared memory segment is used to reduce the amount of copying required to send or receive data. Internally, this data buffer is a piece of shared memory that two or more process gain access to using the buffer manager method. Consequently, there can be multiple buffer manager objects granting access to a single data buffer.

One process must create the physical data buffer using the *creat()* method with a integer *index*, and the buffer manager will use this value as the key to the shared memory allocation system. If *index* is not included in the calling parameters, the buffer manager will attempt to find a unique shared memory identifier. Once a data buffer has been created, another process can attach to the buffer. So data written into the buffer by one process can be read by another process. To do that, the second process must instantiate a buffer manager object, and then get the shared memory id from the first process (communicated by TPC facility). The second process attaches its buffer manager to this shared memory. Now both processes are attached to the shared memory, and a buffer manager will immediately see any activity on the buffer from another buffer manager.

In MTL, the two parties interested in the data buffer are the user and the

appropriate context in the protocol daemon. The context creates the data buffers, and the user interface attaches to them. This avoids that the user's demise will remove the shared memory buffer. The buffers are created and attached to in pairs, one for sending, and one for receiving. The user writes data to the send buffer, and the context reads data from it. The context also keeps track of what data has been acknowledged, so that portion of the data buffer can be reused. The buffer manager also provides the methods for reading, writing, and manipulation of the data buffers.

The data delivery service class *del_srv* is an abstract class specifying the interface to a data delivery service system used to send and receive packets. Classes derived from del_srv implement this abstract interface by employing a particular data delivery service, such as *ip_del_srv* for IP and *udp_del_srv* for UDP. An instantiated derived data delivery service object is the daemon's access point to the network. There are three functions that return data delivery service characteristics: *get_maxpdu*() returns the maximum protocol data unit size possible on this delivery service; *get_rate*() returns the number of bytes per millisecond; and *get_burst*() returns the number of bytes in a single burst. The addresses used by the system calls do not necessarily follow the same structure, so the data delivery service addresses are abstracted by the class

dds_address.

The data delivery service addressing structure *dds_address* is also an abstract class. It hides the internal structure of the addresses, and provides instead a common set of functions on the address. Specific data delivery services derive an address class from dds_address, providing an implementation for each of the pure virtual functions. For example, the *ip_dds_address* class is based on the dds_address class and the sockaddr_in addressing structure.

Packets are the vehicle for data and information exchange between endpoints. Packets are sent and received by a data delivery service that treats the contents of the packets as un-interpreted payload. The derived protocol defines the structure of its packets, and information is placed or extracted only with knowledge of the structures. Therefore, the packet class provided by MTL does not impose a structure on the packet, but rather provides packet shells; packet shells are the manipulated by both MTL and the derived protocol as is appropriate.

The packet class is not designed to be a base class. A packet object is just a repository for data to be sent and received via the data delivery service; packet classes from the derived protocol should contain a member

variable of MTL packet type, and their member functions should impose some order onto the raw data contained within the packet.

There are two packet manipulator classes in MTL: the *packet_pool* class and the *packet_fifo* class. The packet pool class is a general repository for packet objects and manages packet objects. The packet pool is responsible for allocating all of the packets in the system, and de-allocating them when the daemon terminates. For incoming packets, the context manager asks the packet pool for a new packet shell in which to put the data from the network. For outgoing packets, the context will ask for a packet shell, and then fill in the packet information according to the structure imposed by the protocol. The *packet_fifo* class builds a FIFO list of packets. There is no restriction on the size of this FIFO except for the number of packets in the system. Each context has two packet FIFO objects, one for holding unprocessed receiver packets, and one for holding outgoing packets that have been sent.

There are two ways to view a packet, as monolithic contiguous memory or as a vector of scatter-gather elements. Methods that operate on a monolithic packet use a function to get a pointer to the start of the contiguous data. Protocol-specific agents then use this pointer to read and write to offsets within the packet. The scatter-gather vector is a set of

pointer, length pairs. The packet class provides member functions to set and retrieve the scatter gather elements. This style is intended for constructing packets with minimal data copying. The data delivery service object provides some methods that can get address information from incoming data packets, and put address information to outgoing packets. A packet object is designed to be handled by reference to avoid excessive data copies. When a packet is initialized, it use count is 0. Each time a copy of the reference to the packets is given out, the use count increases by 1. When the entities complete their use of the packet, the use count decreases by 1. The packet cannot be returned to the packet pool until its use count is 0.

MTL defines some data types: for example, byte8 for an 8 bit integer, short16 for a 16 bit integer, word32 for a 32 bit integer, and word64 for a 64 bit integer.

Because original MTL version 1.5.1 did not take account the timer management issue, Jiang Yonglin modified the original source code of MTL and introduced two new classes, *cctimer* and *cctimer_link*, to manage the timer in a linked list. The idea of managing linked timer list is introduced by Prof. Atwood. This modification can improve the efficiency of timers that are used by any synchronized service [13].

4.2 Implementation.

4.2.1. Assumption.

We introduced MTL concept and infrastructure above. The next question is how to implement our protocol based on the MTL. Before we introduce the implementation, we introduce some assumptions on which we build our algorithm implementation.

First, we assume that the service nodes (MNs) are always functional, and never change or crash after the Mesh is constructed. There is no new node joining the session tree, and no node leaving the tree later. So we can use the static method to configure the forwarding tables of the MNs. The dynamic way is more complex to implement than the static way. For more detail, please review the section 3.2.1 Mesh Construction. This assumption can simplify our implementation. Because our goal of this implementation is to build a multicast session tree to prove that the proposed algorithm is feasible and efficient, we restrict this condition of the scenario. Although this assumption will reduce our implementation's capability of reflecting real network conditions, the implementation can still verify and validate the proposed tree auto-configuration algorithm.

The dynamic way to build mesh and forwarding table will be implemented and added in the future work.

Second, we assume that the Point-Of-Contact (POC) is not available in the scenario in which our implementation is built. Although the POC method can help nodes get information about the Mesh and simplify routing computation, it does not reflect the current technique in real networks. Server from which applications can retrieve information about multicast service nodes and topology do not exist.

Third, we assume that all nodes can provide UDP data delivery service, rather than IP service. Because some of the IP services need root privilege, it is more realistic to implement our algorithm based on UDP. Because this algorithm proposes a topology that is different from the topology in RMTP-II, we modify something in RMTP-II. First, every node uses two ports to communicate with others: one port for unicast control information, and one for multicast data channel. A multicast session is identified only by the root's multicast port. So the multicast session ID has two parts: root's IP address, and the port number of root multicast channel. Second, we add some packet formats to adapt to the modifications in topology. The detailed packet formats used are described in the next section.

4.2.2. Common Structure and Classes.

MTL reflects some common characteristics of transport protocols. We derive our implementation from the MTL classes to achieve the features providing by MTL.

Some classes in our implementation directly derive from MTL classes. The basic structure of our implementation is same as the User/Daemon model in MTL.

There are four types of node in this implementation: Sender, Service node in the Mesh (MN), Service node (SN), Receiver. The functionality of these node types is discussed in Chapter 3. The design details will be given in the next section. Here, we just discuss the general structure and class division of those node types.

Every node in this implementation has a daemon process derived from the MTL daemon process. The class definition of our daemon and is shown below:

```
class TACDaemon : public mtldaemon
```

```
{
    private:
        int Node_Type;

    public:
        time_t start_t;
        TAC_SESSION_ID d_sid[MAX_SES_NUM];

        TACDaemon();
        TACDaemon(int protocol_num);
        ~TACDaemon();

        int parse_args(int argc, char** argv);
        int init_daemon(int protocol_num, char* req_addr);
        req_action dispatch_request(user_request* request,
        req_addr_struct* user_addr);
        int change_node_type(int new_type);
};
```

The daemon process has a private integer member variable, *Node_Type*, which indicates the node type of the application running on the host. This variable is set when the daemon process is initialized. A public function,

int change_node_type(int new_type), can change the node type value only if a receiver node is selected as a new local SN in the SN election. The start_t is variable used to store the beginning timestamp of the daemon process. The integer array *d_sid* is a multicast session ID list that the node joined. The structure *TAC_SESSION_ID* has two fields: a 64 bits structure *SESSION_ID* variable Session_ID in which the multicast session ID is represented, and a character *root* indicating if it is the root node of a session. The structure *SESSION_ID* has two fields: a four byte IP address and a 2 byte UDP port number. There are three implementations of virtual functions defined in mtldaemon class. The function *parse_args* parses command arguments and initializes the daemon according to the arguments. We ask the user to specify the node type in the command line. The function *init_daemon* initializes the daemon process after the daemon object is constructed.

The TACContext_manager is derived from context_manager. The following is the TACContext_manager class definition.

```
class TACContext_manager : public context_manager {
public:
    Forwarding_Table* mn_forw_table[MESHSN_NUM];
```

```
    TACContext_manager(word32 num_contexts);
    ~TACContext_manager();

    init_context(user_request* request, req_addr_struct* user_addr);
    int release(word64 key);
    void handle_new_packet(packet* pkt);
    word32 satisfy();

    //  quiescent_routine function which was introduced by yjiang
    void quiescent_routine(TACContext * c);

    TACContext* search_context_list(short16 context_port) ;
};
```

The public *mn_forw_table* is the forwarding table, if the node is an MN. This forwarding table is set up when an MN start up. The structure includes a destination node address and a next_hop address.

The constructor instantiates a new context manager object with a number (*num_contexts*) of contexts. The function *init_context* initializes a context and binds it to a socket with a fixed port number. The function *release* releases contexts when the user terminates a context. The function

handle_new_packet steers an incoming packet to proper context. If a child sends service node a BindRequest message to join a new session, the context_manager will initialize a new SN context and register this session in the daemon process's session list. The function *search_context_list* is used to find a context with a specific port number in the context list managed by the context manager. The function *quiescent_routine* is an implementation of a virtual function introduced by Jiang Yonglin in context_manager class. The function is called when a context is inactive and a specific event times out.

Each context corresponds to a receiver, a service node (SN), a Mesh node (MN), or a Sender. The class definition is:

```
class TACContext : public context {

    friend class TACContext_manager;

private:
        void unblock_user();

public:
    TACState_machine* c_state_machine;
```

```
  TAC_NODE_TYPE NodeType;
  udp_del_srv* delsrv;
  udp_dds_address* c_addr;

  TACSender* sender;
  TACReceiver* receiver;
  TACSN* service_node;
  TACMN* mesh_node;
  cctimer* c_timer;

// Constructor/Destructor
  TACContext();
  virtual ~TACContext();

  int process_packet();
  int send(user_request* request);
  int receive(user_request* request);
  packet_fifo* send_fifo() { return c_snd_fifo; }
  packet_fifo* recv_fifo() { return c_rcv_fifo; }

  // Implementation of virtuals
  int is_quiescent();
```

```
void go_quiescent ();
int initialize(user_request* request);
int bind();
short16 set_port(short16 n_port);

// Functions to handle the timeout and routine
void handle_timeout(word32 ttype);
void routine();

// Misc
void log_status();
};
```

A variable *c_state_machine* points to a state machine object that records the current protocol-specific state of the context. The variable *NodeType* specifies the node type of the context. *TAC_NODE_TYPE* is an enumerated type given names of node types. There are some pointers to some node objects. The application will initialize a node object and make one of the context's pointers to point to this node object. Each context has a *cctimer* object for timing some events.

Function *unblock_user* activates a blocked user requesting service from

this context. Functions *send* sends packets via the data delivery service object *delsrv*, and *recv* extracts incoming packets from *delsrv*. Function *process_packet* processes incoming packets and delivers the packets to node object. Function *send_fifo* and *recv_fifo* return pointers to the sending packet FIFO queue *c_snd_fifo* and the receiving packet FIFO queue *c_rcv_fifo*. Implementations of MTL context virtual function *is_quiescent* and *go_quiescent* check and set context blocking state. Function *initialize* initializes class member variables and initial states of the context. Function *bind* binds the context with a socket. Function *set_port* sets the socket port number. Function *handle_timeout* handles a time out event, sets the proper state of the context, and calls the proper functions to process it. Function *routine* is the idle function, which is called when a context is waiting for some event. Function *log_status* is a miscellaneous function recording events in a log file.

The user interface class is the user's access point to the protocol services. The following is the class definition:

```
class TACIf: public mtlif {
    public:
        ~TACIf();
        TACif();
```

```
        int reg(int req);
        int release(int no_answer);
        void perror(int res, char* usr_msg);
        void init_req(user_request* command);

    //IPC routines.
        int issue(reqbuf* req);
        int accept(reqbuf* req);
        int inform(reqbuf * req);
};
```

Function *reg* builds a user request with REGISTER command value and sends it to the daemon process. The user register request will make the daemon process initialize a context for this user and put the context in the active context list. Function *release* releases the context that provides services to the user. Function *perror* takes the result code returned from the context and prints out the standard error textual meaning of the result. Function *init_req* builds a user request in a standard user request format according to the user command. There are three IPC facility functions: *issue*, *accept*, and *inform*. The user's requests are issued using function *issue*. The request requires a response from the daemon, so *issue* waits

until the daemon responds with the result of the request. The *issue* rewrites the user request *req* with the results of the request. The *inform* method tells the daemon of some changes with user, but does not require a response, so this call does not block. Function *accept* is used when more than one response is required by an *issue* call. The *accept* call blocks until the daemon sends back a response.

4.2.3. State Representations and function description

There are four kinds of Node types in this implementation: Service Node on the mesh (MN), Service Node (SN), Receiver (RN), and Sender. The basic functions of each kind of node are described in Section 2.2. The State Representations and detailed function descriptions of each node type are given in this section.

4.2.3.1. The Service Node in the mesh (MN).

Because the static method is used to build the mesh, the MN sets up the forwarding table by accessing a specific file when it starts up. Each MN maintains a child list for each session. The state representation of MN is shown in Figure 25.

The state P0 is the start state. At this moment, MN is waiting for an incoming packet. If the incoming packet is a BindRequest packet, MN's state changes from P0 to P1.

In state P1, if SN is the root, it sends BindConfirm or BindReject message to the request node according to the fact that it can accept another binding or not. If SN is not the root, there are two possible conditions: it has been on the tree already and can accept more children, it sends BindConfirm message to the request nodes. If the SN is not on the tree yet and can accept another binding, it sends BindACK to the requesting node. Then it checks the forwarding table entry of the session tree root, and sends BindRequest message to next hop node. It sets a timer for this outgoing BindRequest packet. Its state changes to P2. If it cannot accept another child, it sends BindReject to the requesting node and changes its state to P1. If the BindRequest is received from a Sender, the MN just sends a BindConfirm, if it can establish a multicast session; otherwise it sends a BindReject message.

In state P2, if SN receives BindConfirm or BindReject message from the next hop, MN's state changes to state P3. If the timer times out, the MN changes its state to P1 and sends BindRequest packet again. In state P3, SN sends BindConfirm or BindReject message to requesting node

according to the BindConfirm or BindReject message it received in P2. If the binding is successful, it accepts the binding to request node and binds itself to the next hop. It puts the requesting node into its child list. Then

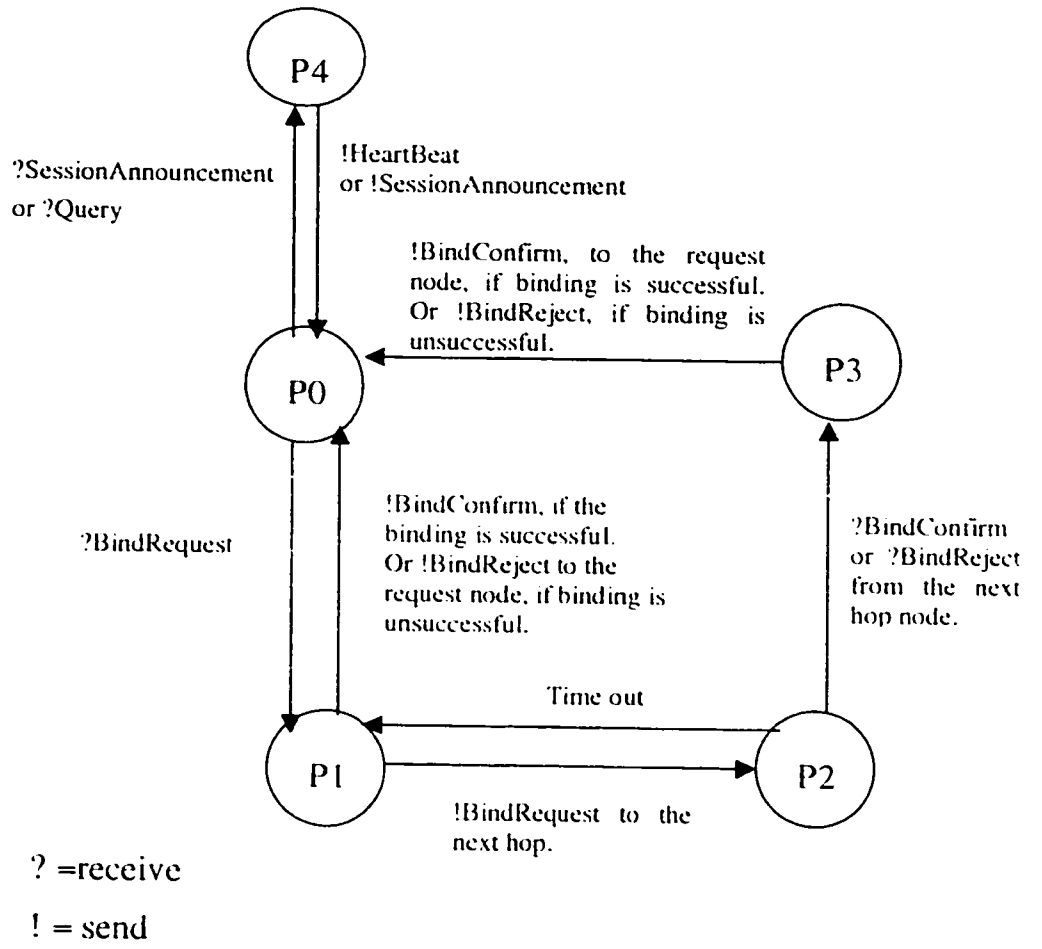


Figure 25. State Representation of Mesh Node (MN).

MN's state changes to state P1.

In state P0, if MN receives a Session Announcement from the Sender, it sends this Session Announcement to its neighboring MNs. If it receives a service node Query message from a SN, it responds to the Query message

with a Heartbeat message. In P0, MN may periodically send Heartbeat messages to all its children and neighboring MN, and wait for Heartbeat_response messages from its children. This process is not shown in Figure 25.

The class *TACMN* is a class providing MN services. The *TACMN* class definition is shown below:

```
class TACMN {

    friend class TACContext;

    protected:
        Children_List* children[MAX_CHILDREN_NUM];
        Children_List* c_head;
        int children_num;
        packet_pool* pool;
        TAC_SESSION_ID sid;
        byte8 root;
        Forwarding_Table* mn_forw_table;

    public:
```

```
    udp_dds_address* parent;
    byte8 bind;
    byte8 treelevel;

    TACMN();
    ~TACMN();
    byte8 is_root();
    int bind_to_tree(context* context);
    int leave_tree();
    int accept_child(context* context);

Children_List* search_children_list(udp_dds_address* child_addr);
    int send_bindrequest(context* ctxt, udp_dds_address* mn_add);
    int recv_bindrequest(context* ctxt);
    int send_bindack(context* ctxt);
    int recv_bindack(context* ctxt);
    int send_bindconfirm(context* ctxt, udp_dds_address* child);
    int recv_bindconfirm(context* ctxt);
    int send_bindreject(context* ctxt, udp_dds_address* child, int
reason);
    int recv_bindreject(context* ctxt);
    int recv_query(context* ctxt);
```

```
        int recv_sessionannouncement(context* ctxt);
        int send_sessionannouncement(context* ctxt, udp_dds_address*
mn_add);
        int send_heartbeat(context* ctxt, udp_dds_address* rec);
        int recv_heartbeat(context* ctxt);
        int send_heartbeat_response(context* ctxt, dds_address* rec);
        int recv_heartbeat_response (context* ctxt);
};
```

•

In this class, an MN maintains a child list, *children*, of *Children_List* structure. This list is a linked list. A *Children_List* structure includes three fields: *udp_dds_address child_addr*, an 8 bit *bind*, and a 16 bit *next*. The *bind* indicates when the child is accepted. The *next* is pointer to the next child. The variable *c_head* is the head of this child list. The *children_num* is the number of children. A MN object maintains a packet pool, *pool*. The *sid* is the session number of the TACMN object. Each MN object serves one session. A MN node can have multiple TACMN to serve multiple sessions. The variable *parent* is the address of the parent node in this session tree. The variable *bind* indicates whether the node is bound to the tree. The variable *treelevel* is the node's value of level in the tree. The pointer mn_forw_table points to the forwarding table maintained in the *TACContext_manager* object.

The method *bind_to_tree* starts the process of binding to the session tree. The MN object selects the next hop node for the destination and tries to bind it. The method *leave_tree* defines the process of leaving a session tree. The method *accept_child* is called, when a BindRequest message arrives, to deal with the new request. The method *search_children_list* returns a pointer to a child with a specific address in the child list. The *send_bindrequest* method sends a BindRequest to the selected possible parent node. The *recv_bindrequest* receives the BindRequest packet from a potential child node. When a BindRequest packet is received, the *send_bindrequest* method analyzes the packet and changes the state of MN. The state machine object *c_state_machine* defined in the context. Other functions send or receive a kind of packet and change the MN state according to Figure 25.

4.2.3.2. Sender.

At the beginning of session tree construction process, Sender should locate the closest MN node in the mesh and try to bind to the MN. In this implementation, the sender tries to bind to a pre-assigned MN. Then the Sender sends a bind request to the selected MN and waits for the bind confirmation from the closest service node on the mesh. After being

accepted by the MN, Sender sets the MN as the root of the session tree, and sends Session Announcement packet to the root. The Session Announcement should contain the Multicast address, session ID, the Sender's address, any specific port numbers to be used, and any global information useful for tree construction.

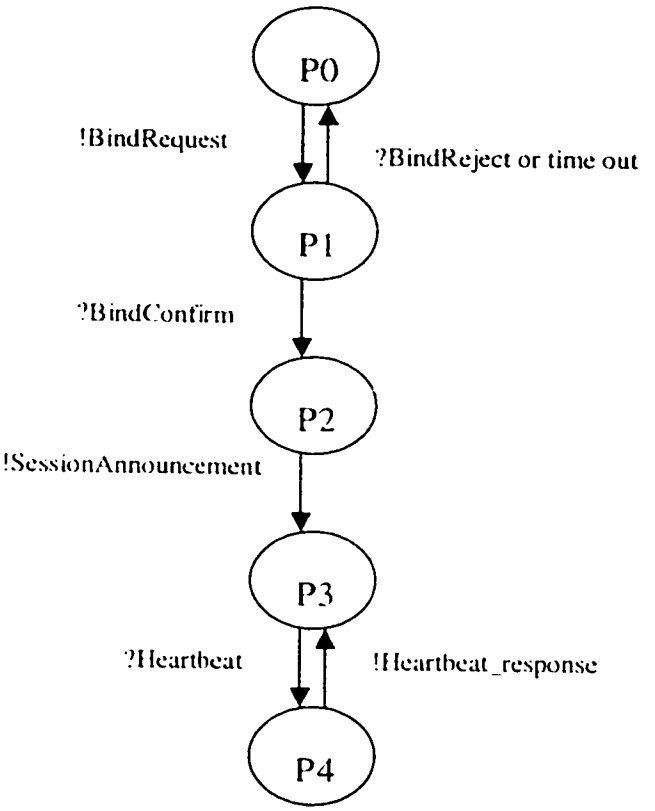


Figure 26. State Representation of Sender node.

In the state representation in Figure 26, state P0 is the start state. In state P0, Sender sends a BindRequest message to selected MN, waits for the response, and changes the state to P1.

In state P1, if Sender receives a BindConfirm message, it sets the MN as the root of the session tree and changes its state to P2. If Sender receives a BindReject or times out, it changes its state to P1 and tries to bind to another MN.

In state P2, Sender sends SessionAnnoucement message to the root of the session tree. The root will send the session announcement to all MNs. Then Sender changes its state to P3. Now, the Sender is bound to the tree and ready to multicast data packet. Sender sends data packet to the tree and receives ACK from the root, which are not shown in Figure 26.

In P3, if Sender receives a Heartbeat packet from the root, it changes its state to P4. In P4, it sends Heartbeat_response packet to the root.

The class TACSender is the class providing Sender's service. The class definition is:

```
class TACSender {

    friend class TACContext;

    protected:
```

```
        packet_pool* pool;
        TAC_SESSION_ID sid;

    public:
        udp_dds_address* root;
        byte8 bind;

        TACSender();
        ~TACSender ();

        int bind_to_tree(context* context);
        int leave_tree();
        int send_bindrequest(context* ctxt, udp_dds_address* mn_add);
        int recv_bindconfirm(context* ctxt);
        int recv_bindreject(context* ctxt);
        int send_sessionannouncement(context* ctxt, udp_dds_address*
            mn_add);
        int recv_heartbeat(context* ctxt);
        int send_heartbeat_response(context* ctxt, dds_address* rec);
};
```

The meanings of variable *sid* and *pool* are the same as in the TACMN

class. The variable *root* is address of the root. The variable bind indicates whether the Sender is bound to the root.

The method *bind_to_tree* defines the process of binding to the root. When a Sender finishes its session, it calls leave_tree to terminate the session. Other methods send or receive packets, and change Sender's state appropriately.

4.2.3.3. Service Node (Group Manager or local router)

The Service Node (Group Manager or Local Router) can accept a limited number of children, SNs or receivers. The state representation is shown in Figure 27.

P0 is the start state in the state representation. In state P0, if this SN receives a bind request from a node, another SN or receiver, then it changes its state to P1.

In state P1, if the SN is already on the tree, it should respond to this message by sending a BindConfirm or a BindReject message according to the fact that it can accept another child or not, and then it changes its states to P0. If the SN is not on the tree, it should try to bind to the tree

before accepting any child, send BindACK to the requesting node, and

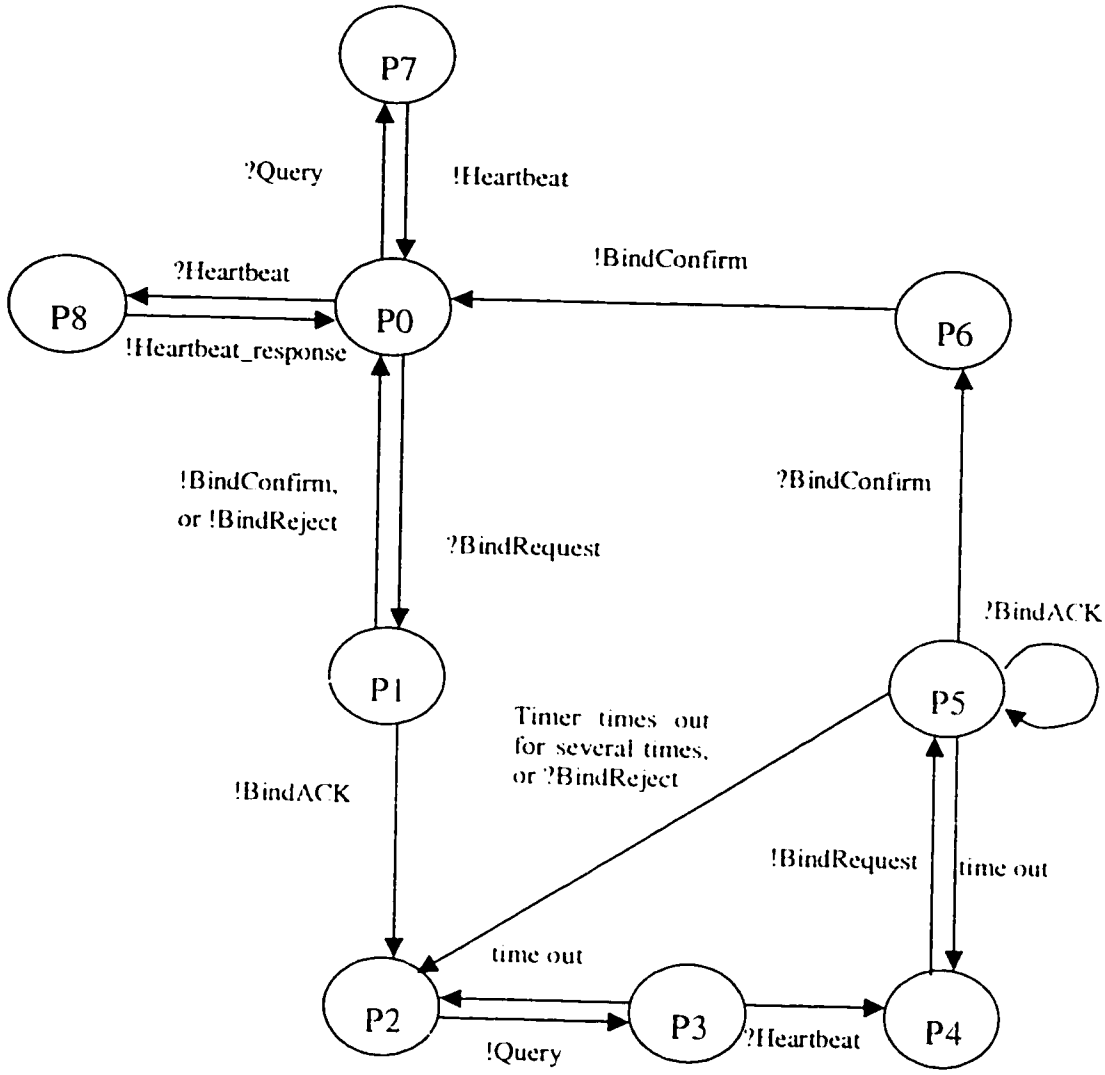


Figure 27. State Representation of Service Node (SN).

change its state to P2.

In state P2, SN uses the *traceroute* method, which is discussed in section 3.2.4, to find some potential parent nodes. Then SN sends Query messages to these potential parent roots and changes its state to P3.

In state P3, if the SN receives Heartbeat messages from other nodes as the response to Query, it changes its state to P4. If timer times out, the SN changes its state to P2 and send a Query message again. If timer times out for a MAX_TIMEOUT times, SN changes its state to P2 and tries to bind to another node.

In state P4, the SN chooses the closest neighbor, and then tries to bind to the chosen one by sending a BindRequest message, and then it changes its state to P5.

In state P5, if it receives the BindConfirm from the chosen node, SN sets the chosen node as its parent SN, and changes its state to P6. Now, it can accept the binding request from other nodes by sending BindConfirm to all requesting nodes. If timer times out, SN changes its state to P4 and sends a BindRequest. If timer times out for a MAX_TIMEOUT times or SN receives a BindReject message, SN changes its state to P2 and tries to bind to another node. If SN receives a BindACK message, it remains in its state until a new event occurs.

In state P6, SN sends BindConfirm messages to all nodes that request to bind to it, and puts these nodes in its child list.

In state P0, if SN receives a Query message from an SN, it responds with a Heartbeat message. The SN should respond to other SN's Query message, if it can accept another child. This function is important to build the sub-tree for multicast. If receiving a Heartbeat from its parent, it responds with a Heartbeat_response message if necessary.

In P0, SN may periodically send Heartbeat messages to all its children, and wait for Heartbeat_response messages from its children. This process is not shown in Figure 27.

The TACSN is the class that defines the SN node. The class definitions are shown in below:

```
class TACSN {

    friend class TACContext;

    protected:
        Children_List* children[MAX_CHILDREN_NUM];
        Children_List* c_head;
        int children_num;
```

```
    packet_pool* pool;
    TAC_SESSION_ID sid;

public:
    udp_dds_address* parent;
    byte8 bind;
    byte8 treelevel;

    TACSN();
    ~TACSN();

    int bind_to_tree(context* context);
    int leave_tree();
    int accept_child(context* context);
    int send_bindrequest(context* ctxt);
    int recv_bindrequest(context* ctxt);
    int send_bindconfirm(context* ctxt, udp_dds_address* child);
    int recv_bindconfirm(context* ctxt);
    int send_bindreject(context* ctxt, udp_dds_address* child, int
reason);
    int recv_bindreject(context* ctxt);
    int send_bindack(context* ctxt, udp_dds_address* child, int
```

```
reason);
    int recv_bindack(context* ctxt);
    int send_query(context* ctxt);
    int recv_query(context* ctxt);
    int send_heartbeat(context* ctxt);
    int recv_heartbeat(context* ctxt);
    int send_heartbeat_response(context* ctxt, dds_address* rec);
    int recv_heartbeat_response (context* ctxt);
    int traceroute();
};
```

Most of member variables and methods have the same meaning as in class TACMN, except for traceroute method, which is used to find all intermediate routers on the path from this SN to the root of the multicast session tree.

4.2.3.4. Receiver.

The receiver initiates a BindRequest to the session tree. If local router can be Service Node and is not on the session tree yet, it should join the tree before accepting the child, as in the state representation shown above. If the local router cannot be a Service Node, receivers on the LAN should

elect one of themselves as the Group Manager. The group manager becomes a Service Node first, joins the session tree, and then accepts

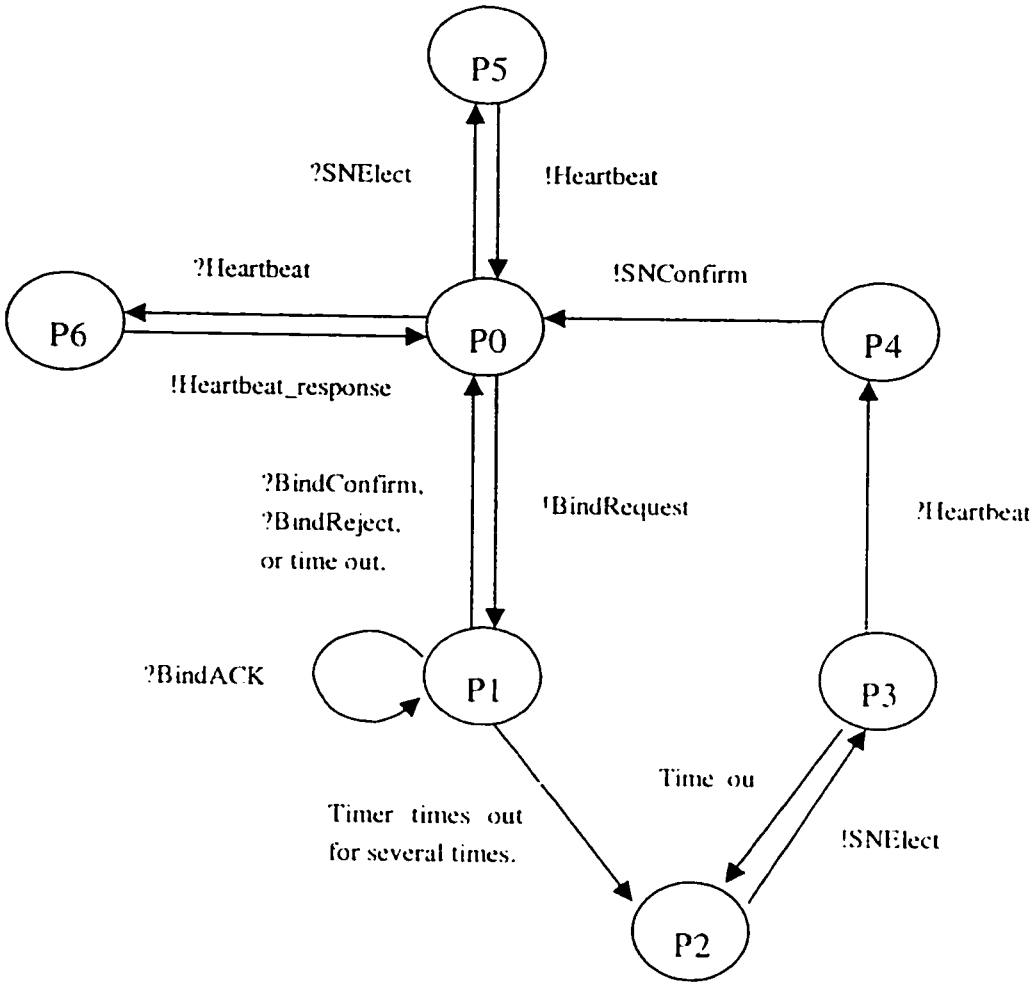


Figure 28. State Representation of Receiver Node.

other receivers as children.

In the state representation in Figure 28, P0 is the start state. A Receiver gets multicast session information by some out-of-band mechanism and broadcasts BindRequest on LAN in state P0. Then it changes state to P1. In state P1, if this receiver gets a BindConfirm message, it binds to the

SN. It sets parent node address and tree level value, and its state goes back to P0. If it receives BindReject message, the binding has failed. If it receives BindACK, it stays in state P1 and waits for the next event. If time out, it goes back to state P0 and resend BindRequest message. If the timer times out for MAX_TIMEOUT times, Receiver can be sure there is no SN on this LAN, and Receiver changes its state to P2.

In state P2, it broadcasts a SNElect message on the LAN to initiates a SN election, and goes to state P3. Any nodes on the LAN that receives this message should reply with a Heartbeat message to the node that initiates the election.

In state P3, if the Receiver receives one or more Heartbeat messages, it changes its state to P4. If time out, Receiver goes back to P2 and resends SNElect message for at most MAX_TIMEOUT times. If it does not receive any response at all, it makes itself an SN, terminates the Receiver object, and initiates a SN object.

In state P4, it chooses a node as the new SN, broadcasts the result in an SNConfirm message, and goes to state P0.

In P0, the Receiver should respond to the Heartbeat message received

from the parent with a Heartbeat_response message, and respond to the SNElect message with a Heartbeat message.

Also, receiver receives SNConfirm in state P0. If it is not the new SN, it sends BindRequest to join desired multicast sessions. If it is the new SN, it terminates the Receiver object, and initiates a SN object.

The class TACReceiver defines the services that a receiver can provide. The class definition is:

```
class TACReceiver {

        friend class TACContext;

    protected:
        TAC_SESSION_ID sid;
        packet_pool* pool;
        udp_dds_address* parent;

    public:
        byte8 bind;
        byte8 treelevel;
```

```
        TACReceiver();
        ~TACReceiver();

        int bind_to_tree(context* ctxt, user_request* request);
        int leaver_tree();
        int elect_sn(context* context, user_request* request);
        int send_bindrequest(context* ctxt);
        int recv_bindconfirm(context* ctxt);
        int recv_bindreject(context* ctxt);
        int recv_bindack(context* ctxt);
        int send_heartbeat(context* ctxt);
        int recv_heartbeat(context* ctxt);
        int send_heartbeat_response(context* ctxt, dds_address* rec);
        int send_snelect(context* ctxt);
        int recv_snelect(context* ctxt);
        int send_snconfirm(context* ctxt);
        int recv_snconfirm(context* ctxt);
};
```

Most of member variables and methods have the similar meaning with those in class TACSN. The method elect_sn defines the process of SN

election. The method *send_snelect* broadcasts SNElect message on the LAN, and *recv_elect* receives SNElect from the LAN.

4.2.4. Packet Formats.

There are several packet formats in this implementation: BindRequest, BindConfirm, BindACK, BindReject, LeaveRequest, LeaveConfirm, Query, Heartbeat, HeartbeatResponse, SNElect, and SNConfirm. In these packet formats, BindRequest, BindConfirm, BindACK, BindReject, LeaveRequest, LeaveConfirm, LeaveRequest, LeaveConfirm, Heartbeat, and HeartbeatResponse are defined in RMTP-II specification. Because the tree auto-configuration algorithm introduces some new process, three new packet formats are introduced: Query, SNElect, and SNConfirm.

4.2.4.1. Packet Header.

All kinds of packet have a fixed header:

<table>
<tr><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="3">VER</td><td colspan="3">Num</td><td colspan="2">Res</td><td colspan="8">TYPE</td><td colspan="16">Session ID</td></tr>
<tr><td colspan="32">Session ID</td></tr>
<tr><td colspan="32">Data or Optional Fields</td></tr>
</table>

Table 1. Packet Header.

VER: Specifies the version number of the protocol, the first field to be checked by a receiver

Num: a number that specifies the number of options present in this packet. If no options are present, then this field is set to 0.

TYPE: Packet type.

Value	Meaning
0	Reserved
1	Data
2	Retransmission
3	ACK
4	BindRequest
5	BindAck
6	BindConfirm
7	LeaveRequest
8	Heartbeat
9	NullData
10	BindReject
11	EOS
12	HeartbeatResponse
13	LeaveConfirm

14	Query
15	SNElect
16	SNConfirm
17-255	Reserved for Future Use.

Some of these packet types are not used in this algorithm.

Session ID: Tree ID is a six-byte identifier of the RMTP-II tree. Currently Tree ID consists of the four bytes IP address and a two bytes UDP port of the top node.

4.2.4.2. BindRequest.

BindRequest packet can be used to join a session. The packet format is shown below.

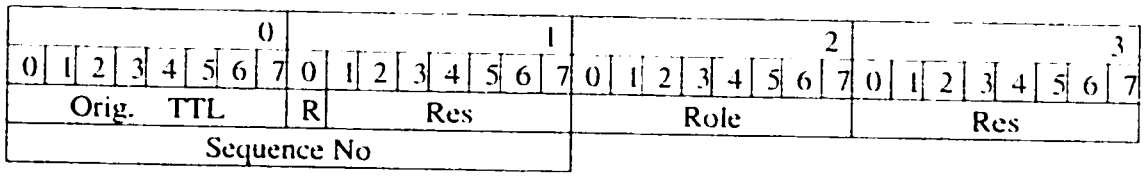


Table 2. BindRequest packet.

Orig. TTL: This is the original TTL sent from the originator. This field allows receiving node to assess roughly the distance from the source.

R: This flag is set to 1 to indicate a rejoin request.

Role: This specifies the role of the joining node in the Session tree.

Sequence No: This is the number of times the same request has been sent.

4.2.4.3. Bind Confirm.

BindConfirm packet is sent from a parent node to a child, when the parent accepts the child.

0								1								2								3							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Child index																Role						C	R	HB_TTL							
Sequence No.																Parent's Tree Level								Overhear							

Table 3. BindConfirm packet.

Child Index: This is the index of the child node in its parent's child list.

Role: This identifies or acknowledges the role of the parent.

C: This flag is set to 1, if the parent accepts the child, or to 0 if the parent rejects the child.

R: This flag is set to 1 to indicate confirmation for a Rejoin packet.

HB_TTL: This is the time-to-live for the Heartbeat packet.

Sequence No.: This is the sequence number of the BindRequest packet.

Parent's Tree Level: The level number for the parent node in the session tree.

Overhead: This is the response constant that specifies the number of HACKs that the answering parent expects for each Data packet.

4.2.4.4. BindACK.

BindACK packet is sent to a child, if the parent needs to process its bind request.

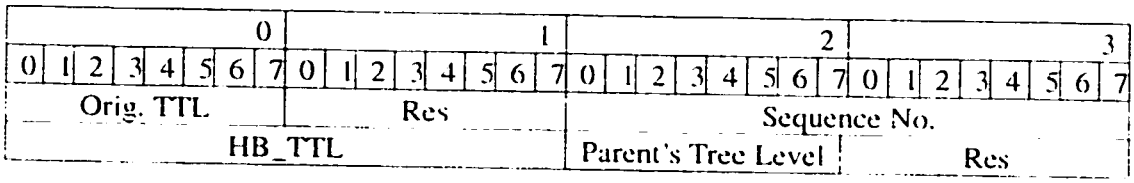


Table 4. BindACK packet.

Orig. TTL. This is the original TTL sent from the originator. This field allows a receiving node to roughly assess the distance from the source.

Sequence No.: This indicates how many times the same packet has been sent.

HB_TTL: This is the time-to-live for the Heartbeat packet.

Parent's Tree Level: The level number for the parent node in the session tree.

4.2.4.5. BindReject.

BindReject packet:

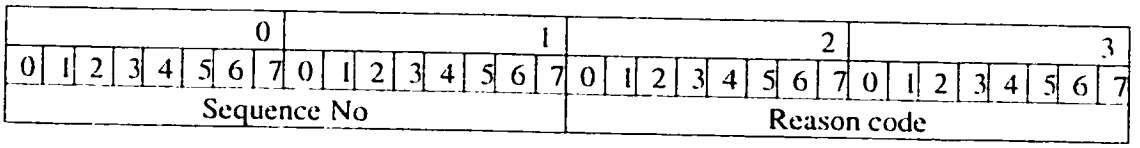


Table 5. BindReject packet.

Sequence No.: This is the sequence number of the BindRequest packet.

Reason Code: This code indicates the reason for sending the reject packet

4.2.4.6. LeaveRequest.

A child sends LeaveRequest packet to its parent, if it wants to leave the session tree.

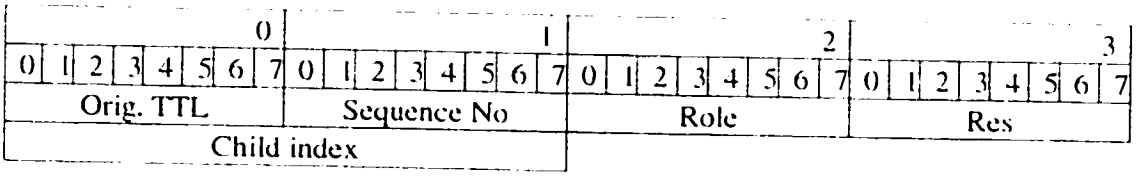


Table 6. LeaveRequest packet.

Orig. TTL: This is the original time-to-live sent from the originator.

Sequence no.: This indicates how many times the same packet has been sent.

Role: This is the role of the node sending the leave packet.

Child Index: This is the index of the child node in its parent's child list.

4.2.4.7. LeaveConfirm.

LeaveConfirm packet:

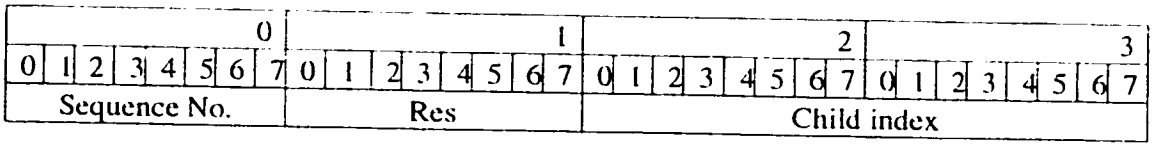


Table 7. LeaveConfirm packet.

Sequence no.: This is the sequence number of the LeaveRequest packet.

Child Index: This is the index of the child node in its parent's child list.

4.2.4.8. Query.

Query packet is sent when a node measures the distance to a service node.

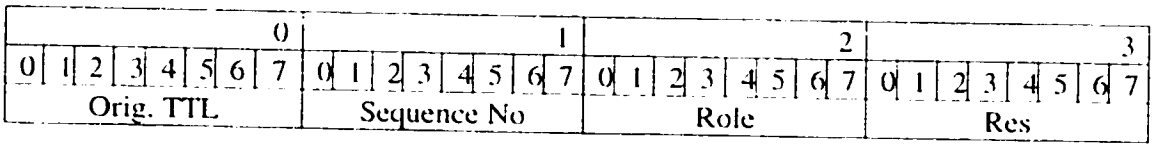


Table 8. Query packet.

Orig. TTL: This is the original time-to-live sent from the originator.

Sequence No: This indicates how many times the same packet has been sent.

Role: This specifies the role of the joining node in the Session tree.

4.2.4.9. Heartbeat.

Heartbeat packet:

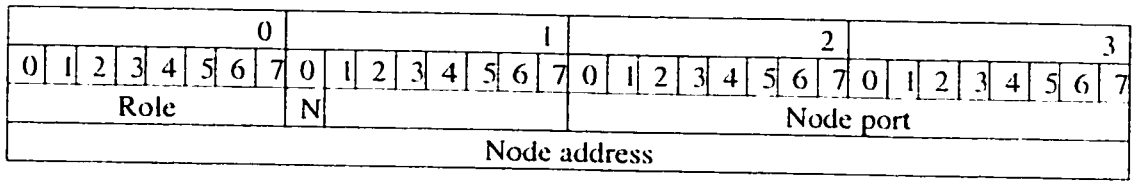


Table 9. Heartbeat packet.

Role: This specifies the role of the node in the Session tree.

N: This flag is set to 1 to indicate that the node must respond immediately.

Node Address: This is the IP address of the node.

Node Port: This is the UDP port number of the Node.

4.2.4.10. HeartbeatResponse.

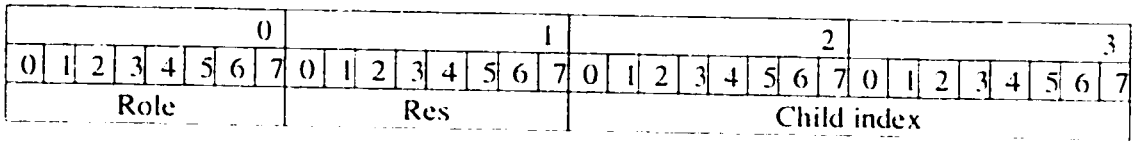


Table 10. HeartbeatResponse.

Role: This specifies the role of the node in the Session tree.

Child Index: This is the index of the child node in its parent's child list.

4.2.4.11. SNElect.

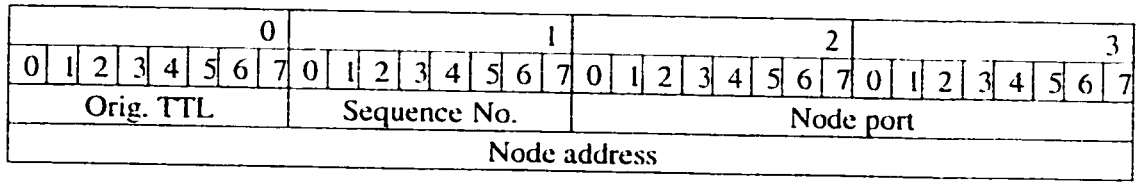


Table 11. SNElect packet.

Orig. TTL: This is the original time-to-live sent from the originator.

Sequence No: This indicates how many times the same packet has been sent.

Node Address: This is the IP address of the node.

Node Port: This is the UDP port number of the Node.

4.2.4.12. SNConfirm:

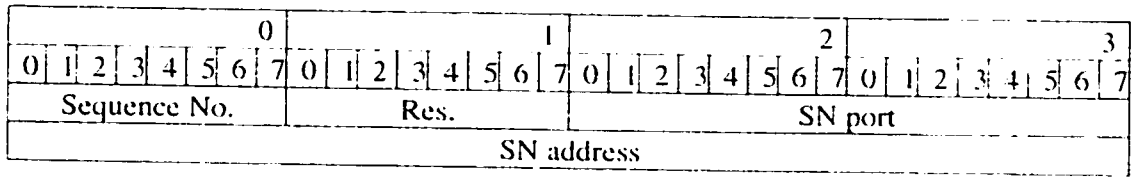


Table 12. SNConfirm packet.

Sequence No: This indicates how many times the same packet has been sent.

SN Address: This is the IP address of the new SN.

SN Port: This is the UDP port number of the SN.

Chapter 5 Conclusion.

In this chapter, a concluding overview of Tree Auto-Configuration is given, summing up its technical and theoretical background. Furthermore, an outlook for future development is provided.

5.1. Summary.

To provide multicast communication in a modern network, several multicast protocols have been developed and deployed, such as DVRMP, PIM-DM, and PIM-SM. Many multicast transport protocols have been proposed. The Reliable Multicast Transport protocol is a hierarchical multicast transport protocol for IP multicast that provides reliable data transmission from a few senders to a large group of receivers. This thesis proposes and implements a Tree Auto-Configuration algorithm for RMTP-II. This algorithm has been left out from RMTP-II, due to its complexity.

This algorithm generates a tree topology used for a tree based multicast protocol. This algorithm is motivated by and concerned with the requirements of the tree-based acknowledgement (TRACK) multicast protocol, and can be extended to provide reliable multicast transport and management. The generated hierarchical tree topology is comprised of

multiple sender nodes, a Mesh, multiple local multicast groups. The Mesh is the infrastructure of the multicast communication and is comprised of multiple pre-deployed service nodes. Local multicast groups are local units of the multicast communication and are comprised of a local service node and multiple receivers. The local groups are connected to the Mesh directly or indirectly. Senders are the data sources and directly connected to the Mesh. The algorithm provides a standard process to build the proposed topology. This process includes several sub-processes: Mesh construction, sender binding to the Mesh, local group constructions, local groups binding to the Mesh, sub-tree on Mesh construction. A set of different algorithms is used to fulfill the tasks of these sub-processes.

With this algorithm, a prototype implementation is built to show its feasibility and capability. This implementation is built based on the Meta-Transport Library, MTL, which is a set of C++ base classes designed to present an infrastructure for building transport protocols. We build four kinds of nodes in this implementation: sender, service node on the mesh, local service node, and receivers. These nodes are the fundamental components of the multicast session tree.

5.2 Future work

The most important component of the proposed topology in the algorithm is the Mesh. It is the multicast communication infrastructure on which the multicast management can be carried.

We can build a software framework on service nodes on the Mesh. The software framework can support emerging requirements of all kinds of multicast protocols, securing the multicast communication, and providing reliable multicast transmissions for all kinds of applications.

As we discussed in section 2.2.1.1, the Mesh has intra-domain and inter-domain capability. All protocols can use this feature and run on it. Multicast membership management can be distributed all around the Mesh. Distributed intelligence to manage multicast protocol can provide flexibility for all kinds of multicast protocol. Mesh can carry data stream and decrease routing algorithm complexity.

The Mesh can also be the infrastructure of multicast security management. The membership authentication system distributed on Mesh can avoid non-authorized people listening to the multicast channel. Additional priority mechanism can avoid a problem that any user can start their multicast session and send data in multicast channel, so called "anycast".

The congestion control and quality of service (QoS) algorithm can be added to the Mesh to fit requirement of different applications. Mesh nodes can cache the data packet to provide error recovery capability. This can make multicast more reliable and efficient.

Reference

[1] Sanjoy Paul "Multicasting: Empowering the Next-Generation Internet" IEEE Network January / February 2000 Vol. 14 No. 1

[2] Kevin C. Almeroth "The Evolution of Multicast" IEEE Network January / February 2000 Vol. 14 No. 1

[3] S. Deering "Multicast Routing in a Datagram Internetwork" Ph. D. dissertation, 1991.

[4] S. Deering and D. Cheriton "Multicast Routing in Datagram Internetworks and Extended LANs" ACM Trans. Comp. Sys., May 1990.

[5] Bin Wang and Jennifer C. Hou "Multicast Routing and Its QoS Extension: Problems, Algorithms, and Protocols" IEEE Network January / February 2000 Vol. 14 No. 1

[6] Miriam Kadansky, Dah Ming Chiu, Brian Whetten, Brian Neil Levine, Gursel Taskale, Brad Cain, Dave Thaler, Seok Joo Koh, Reliable Multicast Transport Building Block: Tree Auto-Configuration, IETF Internet-Draft, work in progress, March 2, 2001. http://www.ietf.org/html.charters/rmt-charter.html

[7] B. Whetten and J. Conlan, "A Rate Based Congestion Control Scheme for Reliable Multicast," GlobalCast Commun. Tech. White paper, Nov. 1998; http://www.talarian.com/rmtp-ii.

[8] Brian Whetten and Gursel Taskale, "An overview of Reliable Multicast Transport Protocol II". IEEE Network January / February 2000

Vol. 14 No. 1

[9] B. Whetten, M. Basavaiah, S. Paul, Lucent Technologies, T. Montgomery, N. Rastogi, J. Conlan, T. Yeh, "THE RMTP-II PROTOCOL", IETF Internet-Draft, working in progress, April 8, 1998.

[10] Andrew S. Tanenbaum, "Computer Network", Third Edition, Printice Hall, Inc. 1996.

[11] "Meta-Transport Library User's Guide", Infrastructure and Networking Research Sandia National Laboratories, Livermore, California.

[12] M. Handley, C. Perkins, E. Whelan, "Session Announcement Protocol", Internet Draft, Internet Engineering Task Force, March 2000.

[13] Yonglin Jiang, "Timer Management in Sandia XTP", M. Comp. Sc. major report, Concordia University, April, 2001.

[14] B. Whetten, D. Chiu, S. Paul, M. Kadansky, G. Taskale, "TRACK Architecture, A Scalable Real-Time Reliable Multicast Protocol," Internet Draft, work in progress, Internet Engineering Task Force, July 2000. http://www.ietf.org/html.charters/rmt-charter.html

[15] Markus Hofmann, Manfred Rohrmuller, "Impact of Virtual Group Structure on Multicast Performance", 1997. Lisboa, Portugal, Ed.: A. Danthine, C. Diot: From Multimedia Services to Network Services, Lecture Notes in Computer Science, No. 1356, Page 165-180, Springer Verlag, 1997