Yoid Tree Management Protocol (YTMP) Specification

Paul Francis

ACIRI
francis@aciri.org

April 2, 2000

Contents

1 Changes **4**

2 Introduction **4**
2.1 Document Status 4

3 Overview **4**
3.1 Basic Tree Concepts 5
3.2 Frame Delivery Modes 6
3.3 Locally Scoped IP Multicast "Links" 7
3.4 Basic Tree Forming Algorithms 9
3.4.1 Planned Topology Changes 9
3.4.2 Forced Topology Changes 11
3.4.3 Member Cluster Configuration 11
3.5 The Rendezvous 12

4 Position of YTMP **12**

5 YTMP Message Formats **12**
5.1 Record Header Fixed Part 14
5.2 Get Member Information Message (1) 15
5.2.1 Get Member Information Query Record (1) 15
5.2.2 Get Member Information Reply 16
5.2.3 New Neighbor Capacity Record (2) 16
5.2.4 Member List Records (3/4/5/17/18) 16
5.2.5 Statistics Record (11) 18
5.2.6 Group Information Record (10) 19
5.2.7 Reverse Path Source Codes 20
5.2.8 Child New Neighbor Capacity Records (17/18) 20
5.3 Join Message (2) 20

5.3.1 Join Query Record (6) . . . 20
5.3.2 Reservation Record (20) . . . 21
5.3.3 Replacement Record (25) . . . 21
5.3.4 Join Reply Message Subtype . . . 21
5.3.5 Root Path Record . . . 22
5.3.6 Error String Record (7) . . . 22
5.4 Quit Message (3) . . . 22
5.4.1 Reservation Record (20) . . . 23
5.4.2 Replacement Authorization Record (26) . . . 23
5.5 Switch Message (4) . . . 23
5.5.1 Join Target Record . . . 23
5.5.2 Switch Message Reply . . . 23
5.5.3 RTT Change Record (14) . . . 24
5.6 Intent to Join Message (5) . . . 24
5.6.1 Join Path Record (8) . . . 24
5.6.2 Intent to Join Reply Message Subtype Values . . . 24
5.7 Root Path Trace Message (7) . . . 25
5.8 Error Message (8) . . . 25
5.8.1 Type Unrecognized (1) . . . 25
5.9 Root Announcement Message (9) . . . 26
5.10 Pairwise Knowledge Message (10) . . . 26
5.10.1 Message Subtype . . . 27
5.10.2 Lifetime Record (15) . . . 27
5.10.3 Head Record (24) . . . 27
5.11 Member Down Message (11) . . . 28
5.11.1 Member ID Record (16) . . . 28
5.12 Extend Reservation Message (12) . . . 28
5.13 Cluster Announce Message (13) . . . 29
5.13.1 Cluster Announce Record (19) . . . 29
5.13.2 Head Parent Record (24) . . . 30
5.13.3 Cluster Pairwise Knowledge Records (22, 23) . . . 30
5.14 Summary of Records . . . 30

6 Major Member States **31**

7 Rendezvous Algorithms **33**
7.1 Newcomer Query . . . 33
7.2 Destroy Pairwise Knowledge Reply from Member . . . 34
7.3 Root Announce . . . 34
7.4 Get Member Information Reply from Rendezvous . . . 34
7.5 Member Discovery . . . 34

8 Tree Maintenance Algorithms **35**
8.1 Obtaining and Changing Parents . . . 35
8.1.1 Independent Parent Change . . . 36
8.1.2 Dependent Coordinated Parent Change . . . 36
8.1.3 Dependent Emergency Parent Change . . . 38
8.2 Neighbor Capacity Search . . . 39
8.3 Relatives . . . 41
8.4 Member Switch . . . 41
8.4.1 Emergency Switch . . . 41
8.4.2 Coordinated Switches . . . 42
8.5 Join Group . . . 44
8.6 Root . . . 45
8.6.1 Orphan . . . 45
8.6.2 Root . . . 45
8.7 Background Parent Search . . . 46
8.8 Quit Group . . . 47
8.9 Excessive Hop Count Expired . . . 47
8.10 Neighbor Reachability Detection . . . 47
8.10.1 Application-Frame Time Detection . . . 47
8.10.2 Pre-Application-Frame Time Detection . . . 48

9 Cluster Algorithms **48**
9.1 Cluster Discovery . . . 48
9.2 Foot . . . 49
9.2.1 Head Reachability . . . 49
9.2.2 Foot Announce . . . 50
9.3 Become Head . . . 50
9.3.1 Selecting Cluster Values . . . 51
9.3.2 Selecting a Parent . . . 51
9.3.3 Obtaining Other Children . . . 51
9.3.4 Head Announce . . . 51
9.4 Head Election . . . 52
9.5 Joining a Cluster . . . 53
9.6 Foot Quit . . . 53
9.7 Head Quit . . . 53
9.8 Echoing YTMP Messages to the Cluster . . . 54
9.9 Forwarding Frames over a Cluster . . . 54
9.9.1 Multicast Delivery Mode . . . 54
9.9.2 Unicast Delivery Mode . . . 55
9.9.3 Broadcast Delivery Mode . . . 55
9.9.4 Tree and Mesh Anycast Delivery Modes . . . 55

9.10 Multiple Clusters 55
9.11 Generating Pseudo-random Port and Address Values 56

10 Mesh Maintenance Algorithms **56**
10.1 Non-Cluster Member Mesh Maintenance 57
10.2 Cluster Member Mesh Maintenance 57

11 Open Issues **58**
11.1 Full Tree 58
11.2 Dependence on Rendezvous 59
11.3 Discovery of Infrastructure Members 59
11.4 Security 59
11.5 Group With Identical Group IDs 59
11.6 Determining Spare Capacity 59
11.7 Excess "Background" Activity 59
11.8 Infrastructure-based Members 60

1 Changes

April 2, 2000 Updated for name change from Yallcast to Yoid.

2 Introduction

This document specifies the Yoid Topology Management Protocol (YTMP). An overview of TYMP and its role in the Yoid architecture can be found in "Yoid: Extending the Internet Multicast Architecture". This specification assumes the reader is familier with that document.

2.1 Document Status

This spec is very much a snapshot of a work in progress. In particular, we can expect YTMP to evolve considerably as we try it under different applications and find what works and what doesn't. what doesn't. Except for a few additions, this spec is considerably older than "Yoid: Extending the Internet Multicast Architecture". The latter represents the author's latest thoughts on Yoid, so where the former differs from the latter, the latter can be assumed to more accurately reflect the author's intentions.

Having said that, this spec (or parts thereof) is a better representation of what has been implemented. In particular, this spec contains a more general form of the intent-to-join freezing mechanism.

As for implementation, the following major parts of this spec have not been implemented as of this writing:

- Mesh topology (and broadcast transmission modes).
- Clustering
- Intent-to-join loop detection
- Member switching

3 Overview

YTMP automatically builds reasonably efficient shared tree and mesh topologies among a collection of hosts (presumably hosts running a given application). A separate protocol, YDP (Yoid Distribution Protocol) is used to transmit application data (content) over those topologies. YDP does not rely on IP-multicast, though it can use localized IP-multicast, such as over a single LAN, where appropriate. Instead, the hosts themselves replicate and forward packets over the topologies they have built.

Packets are transmitted host-by-host over the unicast "links" of the tree-mesh using either UDP (*datagram transmission mode*) or over TCP or some TCP-variant running over UDP (*stream transmission mode*). Stream mode should be used wherever possible, mainly because of its flow control attributes, but also because of its reliability. Unicast is used either when two hosts in the tree are more than one (router) hop apart, or when the two hosts are communicating over a point-to-point link.

Packets are transmitted over IP-multicast when multiple hosts are on a single shared-media such as a LAN. Datagram transmission runs directly over UDP, and stream runs over a reliable IP-multicast transport protocol optimized for small and local IP-multicast groups[1]. (This is relatively easy to do, compared to reliable IP-multicast over a wide area.)

Note that stream transmission mode over YDP is not end-to-end reliable nor does it necessarily preserve end-to-end ordering, though it does come close. In either case, YDP forwards content in units of application layer frames.

Each group is associated with a DNS name. Each group also has an identifying string associated with it. This string is unique for the DNS name only. There are no special IP addresses (unicast or multicast) associated with the group per se. All unicast frames are sent via each host's unicast IP address. All IP-multicast frames are sent over a locally assigned multicast IP address.

Each group has one or more *rendezvous*. The rendezvous run on the hosts named by the DNS name of the group, and can therefore be learned with a DNS query for the group's DNS name. Rendezvous are not necessarily members of the group per se, but must know of some small number of members that are (where the term member refers to a yoid host in the group). New members joining the group (*newcomers*) find a rendezvous through DNS, and then find several members already in the group (if any) through the rendezvous. From these members they can find other members, and in particular an appropriate member to join.

Thus, the bootstrapping, or group discovery, mechanism is DNS. This naming/discovery scheme allows any host to individually create as many groups as it likes without coordination with other hosts.

Hosts can join the group either as transit or as stub members. Where possible, hosts are encouraged to join as transit members. They should be stub members only in the cases where 1) they will not be in the group long enough to justify being a transit, or 2) they do not have the resources (processing, bandwidth) to be transit members. An example of the former is a dial-up host logging in to a mail distribution group to get the latest news. An example of the latter is a laptop at the end of a dial-up link joining an audio/video conference.

The purpose of the tree topology is to transmit content as efficiently as possible. The tree, however, is fragile in that the loss of any given member will partition it, albiet temporarily. The mesh topology is built to make the group robust. The loss of any member or even small number of members will not partition the mesh. Therefore, broadcast over the mesh is highly reliable, if not very efficient. Application content can be broadcast over the mesh where appropriate. YTMP also uses mesh broadcast for various tree maintenance purposes (mainly partition discovery).

Figure 1 shows all of the components of the YTMP architecture. DNS is used to find one or more rendezvous, which in turn know about members in the group.

3.1 Basic Tree Concepts

Figure 2 shows the structure of the tree. Again, each member in the tree is a host. The tree is a single shared tree[2] for the entire group. The "links" between tree or mesh neighbor members are either 1) unicast multi-router-hop paths through the internet (the arrows joining pairs of members), or 2) local IP-multicast groups (the arrow joining the LAN

[1] In the remainder of this document, the generic term multicast generally refers to Yoid multicast. Where the context is not otherwise clear, the term IP-multicast is used to refer specifically to IP multicast.

[2] A shared tree is one whereby all members transmit and receive over the same tree.

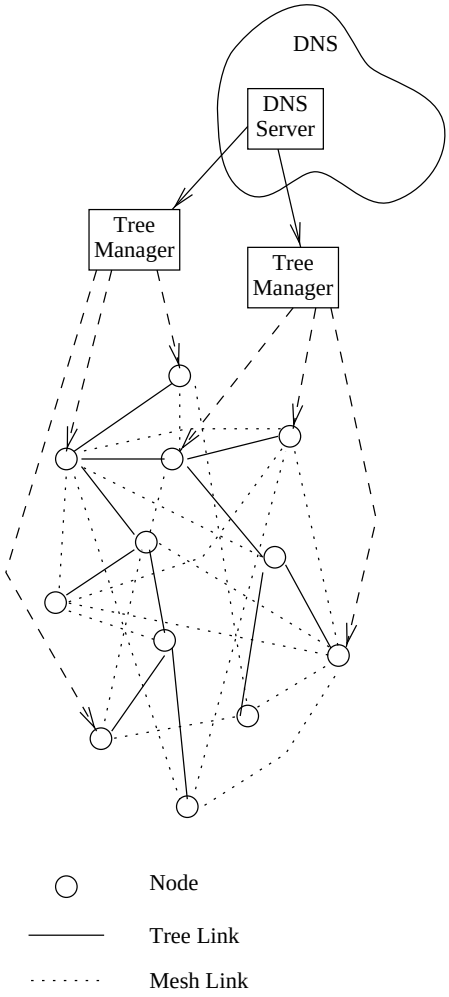


Figure 1: YTMP Architecture

with the member). The tree has no loops. When a member receives a yoid multicast frame, it (generally) forwards the frame to all tree neighbors except that from which the frame arrived. (Note that use of the term "neighbor" alone generaly is taken to mean "tree neighbor".)

The arrows shown on the links in the Figure do not imply that frame forwarding is unidirectional: frames can go in either direction. Rather, they are related to the control relationships between members. The automatic tree creation and maintenance algorithms require that there be a parent/child relationship between members. The member at the pointy end of the arrow in the Figure are the parent members. The member with no parent is the root member, and members with no children are leaf members. Stub members are always leaf members.

Generally speaking, the child member is responsible for seeing to it that it, and by association all of its offspring, remain attached to the rest of the tree. In particular, each member divides the set of all other members into two groups: parent-side members and child-side members. Parent-side members are all members reachable via the parent. Child-side members are all other members. It is the responsibility of each member to make sure that, when it decides to or is forced to get a new parent, the new parent is among the set of parent-side members. Otherwise, the tree would become partitioned.

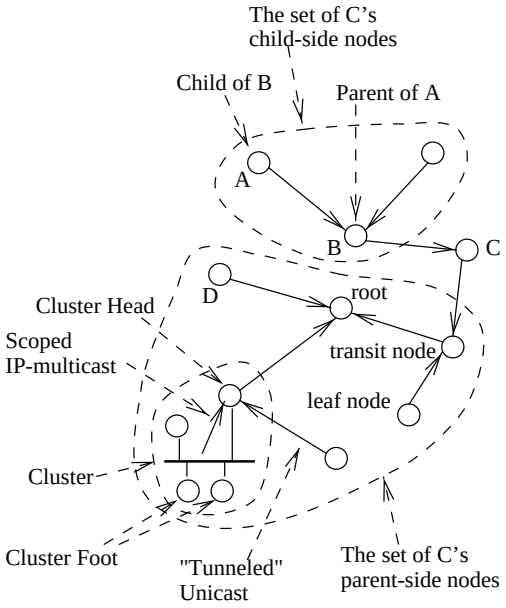


Figure 2: Basic YTMP Tree

3.2 Frame Delivery Modes

There are five *delivery modes* for sending frames to other members. (Note that, strictly speaking, this is part of YDP. It is included here primarily for the purpose of defining terms.)

Multicast: Frames are sent to all tree neighbors (parent and children) except the one from which it was received. They may be sent either datagram or over a stream.

Unicast: Frames are transmitted to a single member in the tree. In the case of datagram, it is transmitted directly to the member, even if the sender is not a neighbor of the member. In the case of stream, it is transmitted neighbor-to-neighbor along the tree to the recipient. The reason for this is to prevent the recipient from having a very large number of TCP connections, for instance because all members in the tree regularly send it frames.

Tree Anycast: This frame is randomly routed either to a neighbor other than the one from which it was received, or delivered to the member's upper layer — either the application or YTMP.

Mesh Anycast: This frame is randomly routed to any known member (neighbor or not), other than the one from which it was received, for a certain number of hops, after which it is delivered to the member's upper layer. This mode may be run datagram or stream.

Broadcast: Each member transmits frames to all mesh and tree neighbors. Broadcast mode is available for datagrams only (not streams). Frames are tagged with a unique identifier and remembered to prevent continuous forwarding of the same frame. This mode is used, among other things, by the tree control protocol to detect tree partitions.

The main *delivery mode* is of course multicast. Unicast is useful in those cases where all application data should be relayed through a single member (or its application) before being multicast. Examples of this include: a controller for an audio/video multicast (to turn on and off the ability of listeners to transmit), and the moderator of a mailing list.

The information required in members for unicast delivery is obtained through the destinations multicasting their member names, with all members remembering the name and, if stream mode, the reverse path. For scalability reasons, there should not be too many members actively advertising themselves as unicast destinations.

The anycast modes have a role in the maintenance of the tree and mesh, in essence as a way to randomly find a member in the topology, and is included solely for this purpose. One can, however, imagine various creative ways this mode could be used by applications. For instance, email could be sent to a few random members to be screened (by humans) for spamming.

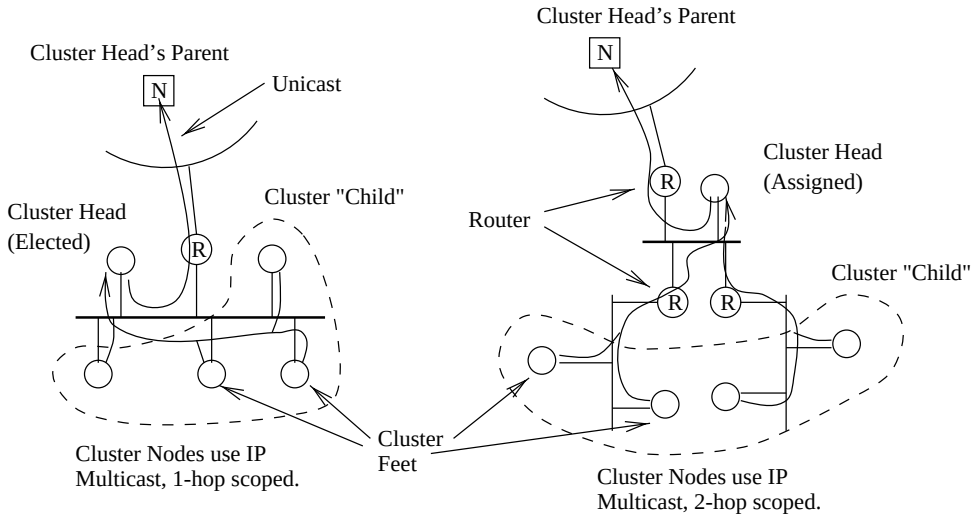


Figure 3: Use of Local IP-Multicast with Cluster Members

3.3 Locally Scoped IP Multicast "Links"

A general goal in Yoid is, to the extent possible, to minimize the load on any given member or (physical) link. One way to do this is to have relatively small member degrees in the tree. This way, each member only has to replicate and forward a few frames for each received, and the local (physical) topology is not flooded with traffic.

In the case of shared-media, such as LANs, small member degrees alone will not insure that the wire itself is not overwhelmed with frame traffic. For instance, consider a case where 20 members are on the same wire, with one of the members the child of a member off of the wire, and all others the children of other members on the wire. Even if the maximum number of children for any given member is only two, a single frame transmitted over the tree would require 20 distinct transmissions of the same frame over the same wire (with only one of them actually being forwarded off the wire by the router).

The solution to this problem is to use IP-multicast over a very local area—normally confined to the single shared-media itself. The left side of Figure 3 shows a group of members belonging to the same group, and attached to the same shared media (say, a LAN). This is called a *cluster*. One of the members is dynamically elected to be the parent member, and is called the *cluster head*, or just *head*. It and only it finds a parent outside of the cluster. The remaining members are by default children of the cluster head, are collectively called the *cluster child*, and are individually called *cluster feet* or just *feet* (or foot if singular).

All members in the cluster transmit frames to each other over an IP multicast group. The time-to-live of the group is set at 1, thus limiting transmission of the frames to the shared media itself. If reliability is required (as is normally the case), a reliable multicast transport protocol is run over IP. While reliable multicast transport protocols over a wide area and for large groups is an open research problem, reliable multicast over a single LAN is well-understood and has been operational in many environments.

Generally speaking, the cluster acts as a single logical member in the topology. For instance, the cluster feet are prevented from having children of their own. The cluster head counts the cluster child (the collection of feet) as a single child, and is allowed to have additional children off the shared media. In this configuration, any given frame croses the wire once for the cluster, once for the cluster head's parent, and once each for any additional children the cluster head may have.

In addition, the job of maintaining the mesh topology is spread out over the members of the cluster, in such a way that the cluster, in total, has roughly the same number of mesh neighbors that a single non-cluster member would have.

The right side of Figure 3 shows an alternative cluster configuration. This configuration is used for shared media where the "uplink" bandwidth is small or non-existant, making it inappropriate or impossible for a member on the shared media itself to be an effective transit member. Examples of such shared media include cable and satellite.

In this case, the IP multicast group associated with the cluster is allowed a time-to-live of 2 or possibly more, allowing the transmissions to cross one router. The head of the cluster is administratively assigned. Its hello messages inform the other members that it is an assigned cluster, and also what the IP time-to-live setting should be. In this configuration, each frame crosses each shared-media segment only one time (excepting retransmissions due to frame loss). Note that there can be more than the two shared-media segments shown in Figure 3.

Note that no other cluster configurations are allowed. In particular, we do not allow dynamically-elected clusters of more than two router hops, and allow this only in the limited case of an administratively-assigned parent. There are two primary reasons for this limitation. First, dynamic configuration of clusters beyond a single wire is complex, especially considering that scoped groups generally have asymmetric connectivity (not all members of the group can hear all others). Secondly, we have concerns about the throughput of reliable multicast protocols as groups become large and widespread. Monitoring a given group for throughput, and trying to adjust the group size to keep it acceptable, appears too complex to justify at this time.

This issue, however, can be revisited as experience is gained both with Yoid and with wide-area reliable multicast. In particular, the idea of a flexible boundary between Yoid and IP multicast, dynamically adjusting IP multicast usage to be more or less as appropriate, is attractive.

3.4 Basic Tree Forming Algorithms

The primary goal in forming the tree is to keep it connected. A secondary but important goal is for members that are near each other in "internet distance" to be also near each other in the tree (in tree hops). Yet another goal is to connect members so that they do not become bottlenecks in the topology. For instance, members with limited bandwidth should be at the edges of the tree rather than in the middle.

If members never crashed and never left the tree, then building the tree would be trivial. Each member simply becomes a neighbor with an existing member that does not already have a full complement of neighbors. This way, the tree would be built-up one branch at a time, would never have loops in the topology, and would never be partitioned.

And in fact, this is our basic approach. *Newcomers* (new members joining the tree) simply find a suitable member already in the tree and attach. The newcomer considers its initial neighbor to be its parent (see Figure 2). Its parent, in turn, considers the newcomer to be its child.

For various reasons, however, the topology needs to change from time to time, even when no newcomers are joining. One reason is that members in the tree may wish to change parents in order to optimize the topology. Another reason is that members may crash or for other reasons abruptly leave the tree. The former case represents a planned topology change. This is described in the following section (3.4.1). The latter case requires a different algorithm, and is described in the subsequent section (3.4.2).

3.4.1 Planned Topology Changes

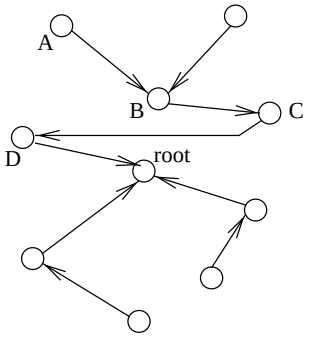
In order to maintain a connected tree when changing parents, each member follows a simple pair of rules:

1. Each member at all times keeps one and only one parent (with the exception of the root member itself).
2. When a member replaces its current parent with a new parent, it selects as its parent a member that is not one of its descendents (a child, a child's child, etc.).

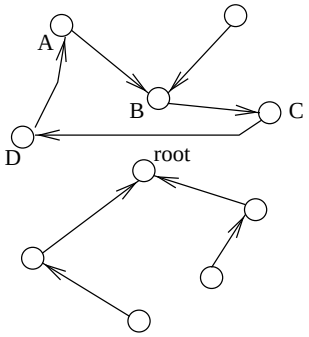
So for instance, looking at Figure 2, if member C were to attach to a new parent, it should select one of the members from the set of members labeled as its parent-side members. As long as it selects one of these, the tree will remain loop-free and connected (upper part of Figure 4).

The tricky part comes when we consider that multiple members may be changing parents at the same time. For instance, say that C decided to attach to member D (for instance, because member D is much closer to it than its current parent). If at the same time member D were to select as its new parent member A (which it theoretically could do because member A is on member D's parent-side), then the result would be the topology shown in the lower part of Figure 4. This tree is partitioned.

One way to prevent this is reduce the valid set of new parents to only those of a member's ancestors (parent's parent, etc.). With this rule, D would not have considered A to be a valid replacement parent, and the partition would not



C reattaches to parent-side
node D. Tree remains connected.



C and D reattach to parent-side
nodes at the same time. This
results in a partition of the tree.

Figure 4: Examples of Changing Parents

have occurred. By the same token, C would also not have considered D to be a valid replacement. Which illustrates the problem with this constraint—the flexibility for optimizing the topology, or staying within a small member-degree constraint.

Another approach is to simply disallow more than one change at a time in the tree. For instance, if member C changes parents first, after the change member A is no longer on D's parent-side, and D would not select it as a new parent. Likewise, if member D changes first, member D would then be on member C's child-side, and member C would not select it as a new parent.

This is more-or-less the approach we take, but rather than only allow a single change at a time for the whole tree, we instead prevent only those changes that would in fact result in loops/partitions. Looking again at the case where C gets a new parent, we can see that the problem occurs only when any member from C's parent-side tries to select a parent from C's child-side. Simultaneous changes limited to within the set of C's parent-side members, or to within the set of C's child-side members, would not result in a loop/partition.

Observing that the only path on the tree from C's parent-side to C's child-side goes through C, there is a relatively simple way to prevent such simultaneous changes:

1. We require all changes to be communicated through the tree, and
2. a changing member prevents any other changes from being communicated from its parent to itself.

So, getting back to our example, D would need to communicate its intent to change parents through the tree to member A. This path includes the root, C's parent, member C, member B, and finally member A itself. Since C is

already in the process of changing, however, it would not allow D's parent change notification to pass from its parent to itself, but instead would return a message to D disallowing the change. If members C and D try to communicate intent to change messages along the tree at the same time, they would both be rejected, and both members would back-off for a short random time before trying again.

This short description gives the basic approach for maintaining the tree. Of course many details are left out, and these can be found in the protocol description itself.

The only other thing we should mention here is the method for optimizing the tree. When a newcomer joins the tree, it puts a premium on joining as quickly as possible, versus finding the nearest member to join. This way, it can start receiving data, albeit over a less efficient path than it might otherwise have, as soon as possible. Once it joins, then it periodically searches among its parent-side member for a better (closer) parent.

This is done by randomly selecting a parent-side member through anycast frame delivery, and pinging that member to determine the round-trip delay. If it is smaller than its current parent, then it changes to the new parent. (If a host doesn't have adequate clock resolution to measure delays, it could send a frame to both its parent and the selected member and see which returns first.) Over time, as it gets harder and harder to find better parents, the member tries less and less frequently. A member's activity then is fairly heavy at first, as it finds and joins new parents, but quickly tapers off as it approaches it's optimal place in the tree.

3.4.2 Forced Topology Changes

The above parent-changing algorithm clearly won't work if a member is getting a new parent because its previous parent crashed. The intent-to-change message would not have an intact path to traverse. One could consider sending the intent-to-change instead to the next member after the (now down) parent on the path to the intended parent, but this member may also have crashed. In general, it is easiest to assume that almost any arbitrary set of failures may have taken place, and to have a general solution for repairing an arbitrarily broken tree.

Towards that end, we note that an intact tree has the following characteristics:

1. Exactly one member is a root (has no parent).
2. Every other member has exactly one parent.
3. The *root path* for any given member (the path consisting of the member's ancestors) terminates at the single root (is loop free).

Thus, each member strives to make sure that these three characteristics are maintained. In particular, whenever a member is parent-less, it considers itself to be the root of the tree, but always assumes that the tree may be partitioned and that other roots may exist. Therefore, it periodically tries to determine if there are other roots/partitions, and tries to join a member in another partition if one is discovered.

Exactly how it does this depends on the likelihood that other partitions exist. For instance, if a member was formerly not a root, but suddenly became one because its parent crashed, then it knows that there are probably other partitions. It also knows one or more members from the other partitions because of the mesh topology. It tries to join one of these members.

To insure that a loop will not form, it transmits a frame along the root path of the intended parent. If there is a loop, the frame will return to itself. In this case, the member waits for a short, random period, and tries again (with either the same prospective parent or a different one). (This is a rather over-simplistic description, but gives the basic flavor.)

If on the other hand a root member cannot find another partition after a reasonable effort, then it assumes that with high (though not 100%) probability it is the "true" root. In this case, it periodically sends an I-am-the-root message to the rendezvous. If there are in fact multiple roots, the rendezvous will detect it, and can inform the roots accordingly.

This safeguard obviously depends on the rendezvous being up, which may not always be the case. (In general, the weak-link of YTMP is its dependence on the rendezvous and their DNS servers.) An additional way to check for partitions is for the root to broadcast an I-am-the-root message over the mesh. Because of the rich connectivity of the mesh, this message has a high probability of reaching other partitions. Roots in the other partitions then recognize that there is a partition.

3.4.3 Member Cluster Configuration

Auto-configuration of shared-media member clusters works roughly as follows. For each group, there is an algorithmically generatable IP multicast address (produced via a hash of the group ID and related information). Each member, when it joins a group, joins the multicast group associated with the generated IP multicast address. It periodically transmits a hello message over the multicast group (with time-to-live 1), and listens for such messages as well.

In the case of dynamically selected cluster heads, when three or more members discover each other, one of them (generally the one with greatest capacity and longest up time) selects itself as the cluster head, and the cluster is formed. In the case of administratively assigned cluster heads, the head's hello message indicates that it has been administratively assigned, and indicates the IP time-to-live that the feet should use as well.

3.5 The Rendezvous

The purpose of the rendezvous is to bootstrap newly joining members (newcomers) into the group. They do this by continuously maintaining knowledge about some number of existing members in the group. If an application on the same host as the rendezvous has itself joined the group, then the rendezvous can know about other members on the group simply through the normal process of being a member in the group. If no application on the same host has joined the group, however, the rendezvous must make specific efforts to monitor the group and maintain knowledge about members in the group.

The rendezvous initially learns about newcomers because they query it for information about the group. The rendezvous periodically queries the members in knows about to learn of still more members. When a member quits the group, it should also inform the rendezvous of this, so that the rendezvous can remove it from its list of known members. Given that members may quit the group abruptly, however, the rendezvous should time out members from its list as it learns of new members in any event.

4 Position of YTMP

The position of YTMP, relative to other protocols as it might appear in a typical implementation, is shown in Figure 5. Note that, while YTMP is certainly effected by actions of the application (create group, etc.), it may be convenient to limit the API with the application to YDP, and let YDP pass these actions onto YTMP where appropriate. As such, YTMP interacts only with YDP (and uses YDP to transmit YTMP messages to other members).

5 YTMP Message Formats

This section gives the formats for the various YTMP control messages. Each control message is transmitted in a single YDP frame, either over datagram or stream depending on the message type.

YTMP messages are denoted in the YDP frame header by the value of 0 in the Protocol field. The YTMP message itself immediately follows the YDP frame header. The YDP frame header of all YTMP messages must contain the frame source option indicating the initiator of the message.

All YTMP messages have a fixed part, formatted as shown below:

These fields are defined as follows:

Q/Message Type: Indicates the type of message. The first bit (Q) is 0 if the message is a query, and 1 if the message is a reply.

Message Subtype: Certain messages set this field to some non-zero value to convey additional information. Except for these cases, it is set to 0 and ignored upon receipt.

Flags: The following flags are defined:

Member Type (bit 17): Set to 1 if transit member. Set to 0 if stub member.

Root (bit 18): Set to 1 if the member is a root. Set to 0 otherwise.

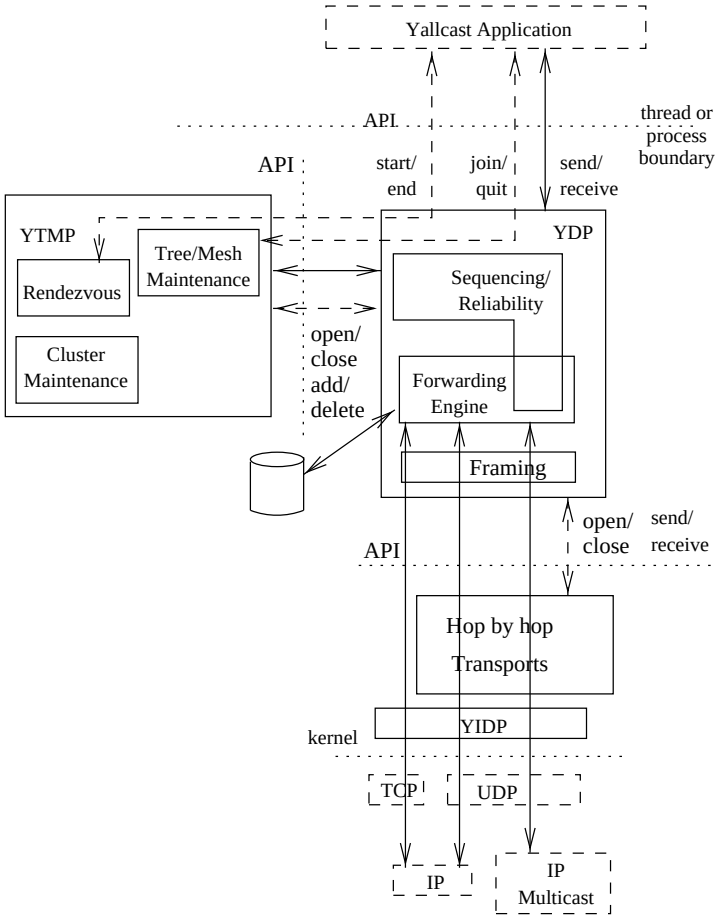


Figure 5: Position of YTMP

```
                    1                   2                   3
 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+---------------+---------------+---------------+---------------+
|Q|Message Type |Message Subtype|             Flags             |
+---------------+---------------+---------------+---------------+
|     Receiver Relationship     |            Reserved           |
+---------------+---------------+---------------+---------------+
|                                                               |
+                          Message ID                           +
|                                                               |
+---------------+---------------+---------------+---------------+
```

Foot (bit 19): Set to 1 if the member is currently a foot. Set to 0 otherwise.

Head (bit 20): Set to 1 if the member is currently a head. Set to 0 otherwise.

Reserved (bits 21 - 32): Transmit as zero, ignore upon receipt.

Receiver Relationship: This indicates what the sender considers the relationship of the receiver to the sender to be. This field is used primarily to detect when two members don't agree on the status of the other. This can happen due to race conditions resulting from network delays. The following values are defined:

None (0): The relationship is undefined.

Parent (1): The receiver is the parent of the sender.

Child (2): The receiver is the child of the sender.

Rendezvous (3): The receiver is the rendezvous.

Prospective Parent (4): The receiver is a prospective parent of the sender.

Relative (5): The receiver is a relative of the sender.

Prospective Relative (6): The receiver is a prospective relative of the sender.

Mesh Neighbor (7): The receiver is a mesh neighbor the sender.

Message ID: This is an opaque field, set by the member transmitting the query, and returned unchanged by the member replying to the query. The values of all zeros and all ones are reserved, and should not be used in queries or replies. The value 1 is used in uninitiated replies, and should not be sent in queries. All other values can be used in the query (and corresponding reply).

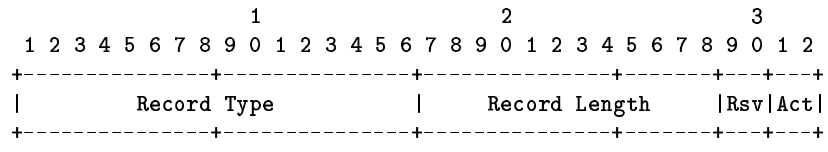
Replies received with an unrecognized Message ID, or with a recognized Message ID attached to the wrong Message Type or transmitted member, are silently discarded. (This can happen, for instance, when a foot receives an YTMP message meant for another foot over the cluster stream.)

The Message Type values are listed in the following table:

Message	Value
Get Member Information	1
Join	2
Quit	3
Switch	4
Intent to Join	5
Header Option Acknowledge	6
Root Path Trace	7
Error	8
Root Announcement	9
Pairwise Knowledge	10
Member Down	11
Extend Reservation	12
Cluster Announce	13

5.1 Record Header Fixed Part

Following the YTMP Message fixed part is a series of zero or more records containing the information required for the message. Each record starts with the fixed part shown below:



The fields are defined as follows:

Record Type: Indicates the type of record. Note that every record has a unique Record Type value. Strictly speaking, this is not really necessary, since records can always be, and usually are, parsed in the context of a given message, and therefore only have to be unique for a given message. But, we do it this way anyway.

Rsv: Reserved, transmit as zero, ignore upon receipt.

Act: This indicates what action should be taken if the option is not recognized, as follows:

Ignore Silently(0): Ignore the option, but continue processing other options and process the message. Don't notify the sender.

Ignore With Notification (1): Ignore the option, but notify the sender that the option was not recognized with the YTMP Error Message of type Type Unrecognized.

Drop Silently (2): Drop the entire message, don't notify the sender.

Drop With Notification (3): Drop the entire message, and notify the sender.

Record Length: The length of the record, in units of 32-bit words, including the fixed part.

This fixed part allows members to parse all records even if they do not recognize some of the record types. In this way, the contents of a given message type can evolve.

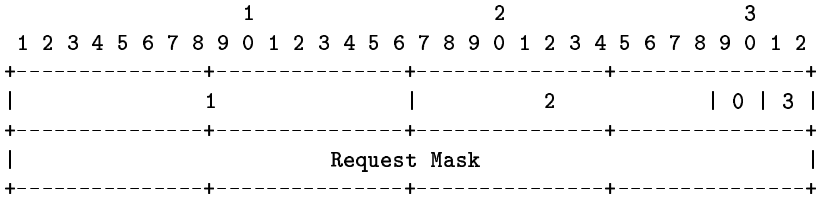
5.2 Get Member Information Message (1)

This query/reply message gives various information about the member being queried. It is used for various purposes, including when determining if a member should be considered as a new parent, and when pinging an existing neighbor to make sure it is still alive. It can be sent over either datagram or stream, depending on its purpose. The reply should generally be returned the same way, however a stream reply can follow a datagram query in the case where the reply is relatively long (would require several IP packets to transmit). Also depending on its purpose, it may be transmitted either as unicast or anycast. A query received via anycast, however, must be returned using unicast.

The Message Type field of the YTMP fixed part is 1.

5.2.1 Get Member Information Query Record (1)

The only record in the Get Member Information query is the Get Member Information Query Record, and is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 1, the Action is 3 (drop with notification), and the Record Length is 2.

Request Mask: A bit-mask specifying the information being requested. A 1 in the bit field indicates that the defined information should be returned in the reply. The bits are defined as follows:

New Neighbor Capacity (1): The number of new neighbors (transit and stub), in addition to the ones that it already has, that can be obtained by the queried member. (Note that the term "neighbor" alone generally refers to a tree neighbor, though it can sometimes refer to both tree and mesh neighbors.

Transit Neighbor List (2): A list of the queried member's transit neighbors and related information. When this call is made to the rendezvous for the purpose of bootstrapping into the group, then arbitrarily selected members are listed instead of neighbors.

Stub Neighbor List (3): A list of the queried member's stub neighbors and related information.

Root Path (4): The list of members on the tree path from the queried member to the tree root.

Statistics (5): Various statistics about the group (as known by the member being queried), such as the number of frames transmitted, frequency of joins/quits, age of the group, and so on.

Reverse-Path Source Codes (6): The reverse-path enabled Source Codes for all sources reachable via all other neighbors.

Group Information (7): Basic configuration information for the group, such as the minimum buffer size and minimum throughput rate.

Child New Transit Neighbor Capacity (8) A list of zero or more children that have reported new transit neighbor capacity for themselves or for their own children.

Child New Stub Neighbor Capacity (9) A list of zero or more children that have reported new stub neighbor capacity for themselves or for their own children.

Bits 10 - 32: Reserved. Transmit as zero, ignore upon receipt.

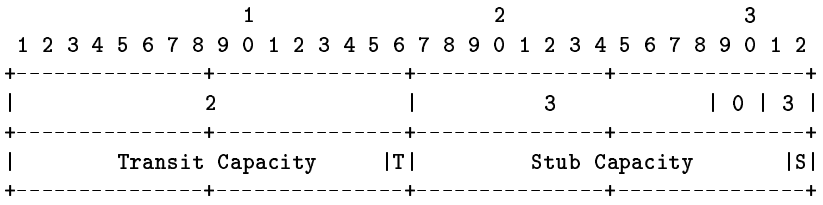
Reserved: Transmit as zero, ignore upon receipt.

5.2.2 Get Member Information Reply

A Get Member Information Reply can be sent with or without being initiated by a Get Member Information Query. For instance, a parent may send an unrequested Get Member Information Reply anytime information sent in a previous Get Member Information Reply has changed. The various records included in the Get Member Information Reply can be attached in any order.

5.2.3 New Neighbor Capacity Record (2)

If New Neighbor Capacity was requested, then the New Neighbor Capacity Record is included in the reply. It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 2, the Action is 3 (drop with notification), and the Record Length is 3.

Transit Capacity: The number of additional transmit members that can still be accepted as neighbors.

T: This field is set to 1 if at least one child has reported either non-zero Transit Capacity, or a non-zero T bit. Otherwise it is set to 0.

Stub Capacity: The number of additional stub members that can still be accepted as neighbors.

S: This field is set to 1 if at least one child has reported either non-zero Stub Capacity, or a non-zero S bit. Otherwise it is set to 0.

5.2.4 Member List Records (3/4/5/17/18)

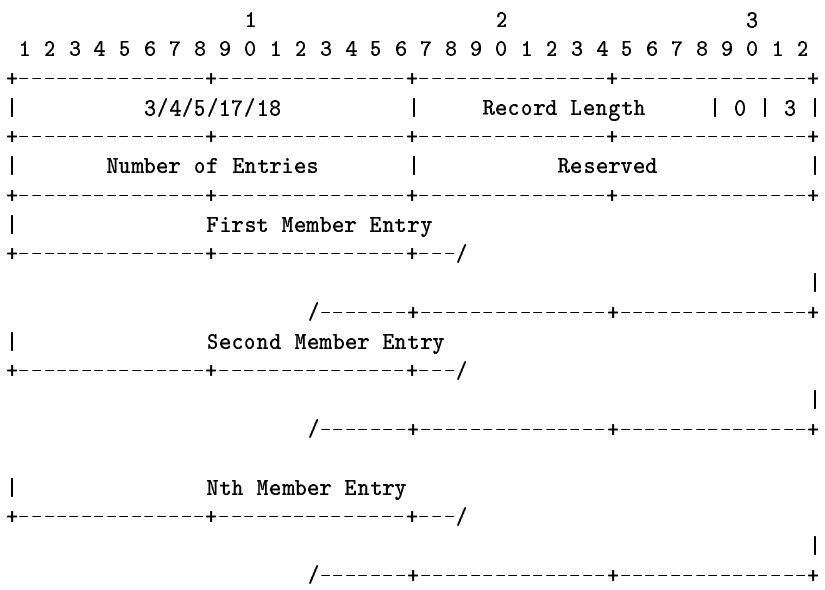
This section gives the format for the five types of member lists that can be returned in the reply: the Transit Neighbor List (3), the Stub Neighbor List (4), the Root Path (5), the Child New Transit Neighbor Capacity List (17), and the Child New Stub Neighbor Capacity List (18).

These correspond to bits in the Request Mask of the query. If the list is a Transit Neighbor List, then the first listed must be the parent. If the member has no parent (either because it is the root, or because it temporarily has no parent, that is, it is an *orphan*), then the first listed member must be a null entry, as defined below.

If the list is a Root Path, the entries are listed in order of parent, parent's parent, etc., up to and including the root member.

In the case of the Child New Transit Neighbor Capacity and Child New Stub Neighbor Capacity Records, a child may be included in this list if and only if it has either 1) reported that it itself has new neighbor capacity of the corresponding type (transit or stub), or has reported that its offspring has capacity of the corresponding type (through the T or S bit of the New Neighbor Capacity Record). The replying member is not required to list all such children. Typically one or a few will suffice.

The five records have the same format, differing only by their record types. They are formatted as shown below:



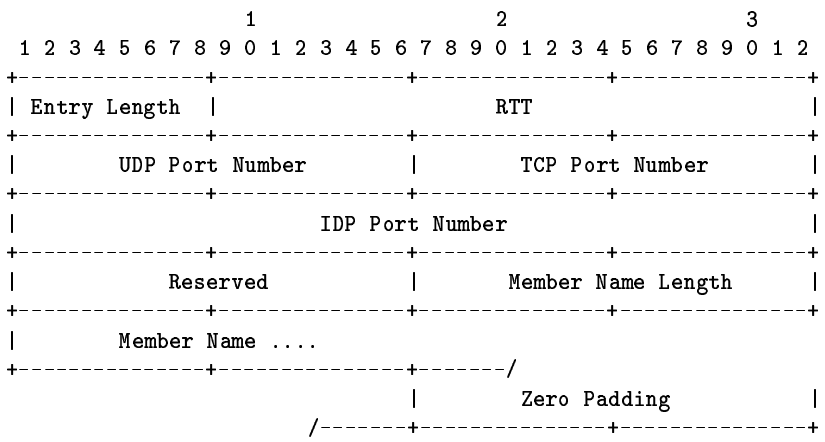
The fields are defined as follows:

Fixed Part: The Record Type one of 3, 4, 5, 17, or 18. The Action is 3 (drop with notification), and the Record Length is variable.

Number of Entries: The total number of entries listed in this record.

Reserved: Transmit as zero, ignore upon receipt.

Each member entry listed has the format shown below:



The fields are defined as follows:

Entry Length: The length of this entry, in 32-bit words.

RTT: The estimated average RTT time, in microseconds, for this neighbor, measured over a reasonably recent time period. The value 0 is used for any RTTs of less than one microsecond. The value 16777215 (all ones) is reserved to mean that the RTT is unknown. The value 16777214 is used to encode any RTTs equal to or greater than 16777214 microseconds (slightly less than 17 seconds).

UDP Port Number: This is the UDP port number over which the listed neighbor will receive datagram frames for the given group.

TCP Port Number: This is the TCP port number over which the listed neighbor will accept connections for stream frames for the given group. This field must be set to 0 if the member prefers using yTCP over TCP.

IDP Port Number: This is the IDP port number over which the listed neighbor will accept frames. This IDP Port Number is learned from the Listen IDP Port of the IDP header.

Reserved: Transmit as 0, ignore upon receipt.

Member Name Length: The length, in bytes, of the Member Name (not including padding).

Member Name: The DNS name of the listed member. It is padded out, if necessary, to an integral 32-bit boundary by zeros.

A null entry contains the following values:

Entry Length: 1.

RTT: 16777215 (unknown).

UDP Port Number: 0.

TCP Port Number: 0.

IDP Port Number: 0.

Member Name Length: 0.

Member Name: None.

5.2.5 Statistics Record (11)

If Statistics is requested, then a Statistics Record is included in the message. It is formatted as shown below:
The fields are defined as follows:

Fixed Part: The Record Type is 11, the Action is 3 (drop with notification), and the Record Length is variable.

Number of Entries: The total number of entries listed in this record.

Reserved: Transmit as zero, ignore upon receipt.

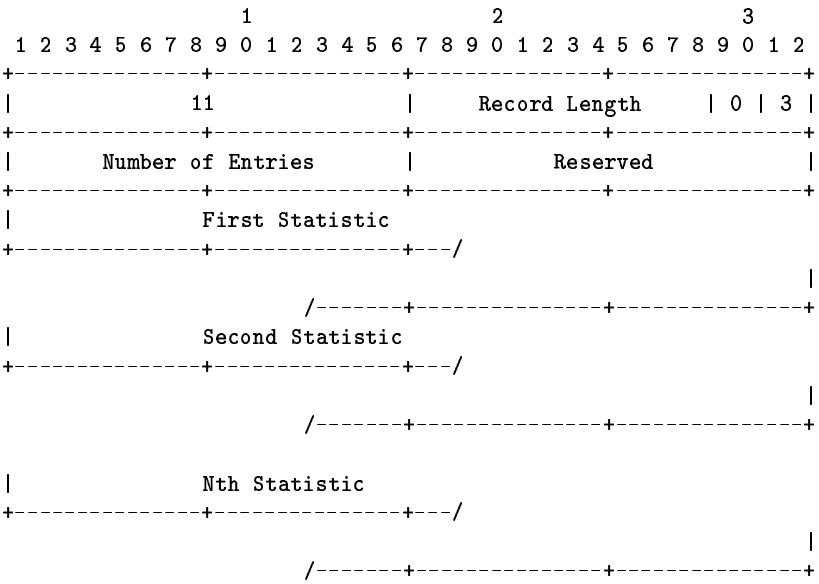
Entries: Each Statistic starts on an integral 32-bit boundary, and is formatted as shown below:

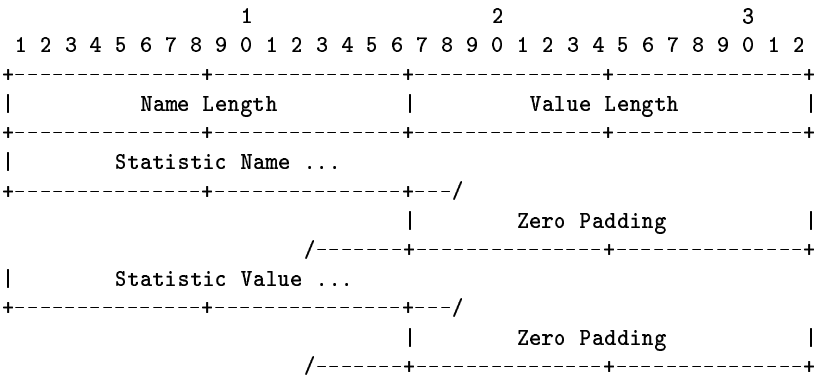
These fields are defined as follows:

Name Length: The length, in bytes, of the Statistic Name, not including padding.

Value Length: The length, in bytes, of the Statistic Value, not including padding.

Statistic Name: The name of the statistic, padded out to an integral 32-bit boundary with zeros. The name consists of the printable subset of ASCII characters, including spaces.





Statistic Value: The value of the statistic, padded out to an integral 32-bit boundary with zeros. The name consists of the printable subset of ASCII characters, including spaces. If the value is a number, the encoding is the same as that expected by the C library `atof()` function.

The following table gives the currently defined statistics:

Name	Value Format
`Outgoing Frames per Second, Lifetime Average`	Number
`Outgoing Frames per Second, Hour Average`	Number
`Outgoing Frames per Second, Minute Average`	Number
`Incoming Frames per Second, Lifetime Average`	Number
`Incoming Frames per Second, Hour Average`	Number
`Incoming Frames per Second, Minute Average`	Number
`Neighbor Changes per Hour, Last Hour`	Number
`Neighbor Changes per Hour, Lifetime Average`	Number
`Member Age, in Seconds`	Number
`Maximum Available Bandwidth (Bytes per Second)`	Number

The Names given here are case insensitive, but otherwise must be encoded exactly as shown, with exactly one space between text, and no white space leading or trailing.

5.2.6 Group Information Record (10)

If Group Information was specified in the query, this record is included in the reply. This record is formatted identically to Statistics Record, except that the Record Type is 10 rather than 11. This record is typically used by the Rendezvous to convey basic configuration information to a newcomer (see Section 7.1).

The following table gives the currently defined types of group information:

Name	Value Format
`Discard Threshold, Bytes per Second`	Number
`Minimum Buffer Size, Bytes`	Number
`Frame Discard Age, Milliseconds`	Number

5.2.7 Reverse Path Source Codes

If Reverse Path Source Codes was specified in the query, the Header Option Record, defined in "Yoid Distribution Protocol (YDP) Specification", is used to transmit the Reverse-Path Source Codes. One record is included for each Reverse-Path Source Code.

Each record contains the exact Header Option that would normally be attached to a content frame to convey the Source Code.

5.2.8 Child New Neighbor Capacity Records (17/18)

The Record Type of 17 refers to the Child New Transit Neighbor Capacity Record. The Record Type of 18 refers to the Child New Stub Neighbor Capacity Record. These records are returned if Child New Transit Neighbor Capacity or Child New Stub Neighbor Capacity, respectively, were requested in the query.

The format for both of these records is the same as that of the Member List Records (Record Types 3 and 4, Section 5.2.4). A child may be included in this list if and only if it has either 1) reported that it itself has new neighbor capacity of the corresponding type (transit or stub), or has reported that its offspring has capacity corresponding type (through the T or S bit of the New Neighbor Capacity Record). The replying member is not required to list all such children. Typically one or a few will suffice.

5.3 Join Message (2)

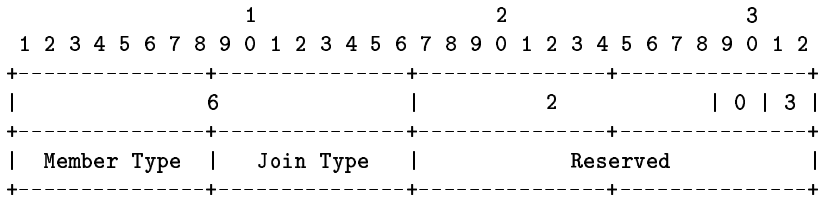
The Join Query is sent by a member that wishes to become the child neighbor of another member. It is sent only after the prospective child has verified that the join will not result in a loop/partition of the tree, and will be accepted with high probability (see Section 8.1). It is always sent over a frame stream — one that has usually already been established.

The Reply either accepts or rejects the request. It is carried over the same stream.

The Message Type field of the YTMP fixed part is 2.

5.3.1 Join Query Record (6)

The Join query always contains a Join Query Record, which is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 6, the Action is 3 (drop with notification), and the Record Length is 2.

Member Type: Set to 1 if the child is a transit member, set to 0 otherwise.

Join Type: This field specifies the context of the join. The context impacts the criteria for accepting or rejecting the join. The following values apply:

Coordinated (0): Joining as part of a coordinated neighbor change, or as a newcomer.

Emergency (1): Forced emergency (non-coordinated) join.

Reserved: Transmit as zero, ignore upon receipt.

5.3.2 Reservation Record (20)

The Join query may contain a Reservation Record. This is used by the joining member to identify the reservation under which it is joining. This is used in the case where a head is joining the same parent used by the previous head.

It is formatted identically to the Member ID Record (Section 5.11.1), except that the Record Type value is 20.

5.3.3 Replacement Record (25)

The Join query may contain a Replacement Record. This causes the receiver of the Join to simply replace the child named in the record with the sender of the Join Message. It is used when a new head (the sender of the Join) is replacing its previous head (the member named in this record) as the child of their parent (the receiver of the join).

When the receiver of the Join accepts the Join, it (directs YDP to) all at once stop sending frames to the replaced child and start sending them to the new child. As a result, no frames in the direction of parent to child are dropped. YDP also stops receiving frames from the replaced child and starts receiving frames from the new child. Since the two children are not exactly synchronized, this may result in dropped or duplicated frames, but it does not result in a loop. It also sends a Quit reply to the replaced child, and drops the stream connection.

It is formatted identically to the Member ID Record (Section 5.11.1), except that the Record Type value is 25.

5.3.4 Join Reply Message Subtype

The Message Subtype of the Join Reply is interpreted as an acknowledgement code. Its values are defined as follows:

Accept (1): The Join request is accepted.

Other Reason (2): The join request was rejected for a reason other than those listed below.

No Remaining Neighbor Capacity (3): The join request was rejected because there was no remaining neighbor capacity. This could happen if the queried member obtained more children between the Get Member Information and Join messages.

Root Path Invalid (4): The join request was rejected because the requesting member is in the root path of the requested member.

Replacement Invalid (5): The member named in the Replacement Record is not a current child.

Orphan (6): The queried member is in a persistent Orphan state. That is, it has run the Neighbor Capacity Search, but was unable to find capacity or arrange a switch.

A Join query is never accepted if the querying member is in the root path of the replying member. If the Join Type of a query is Coordinated, the replying member should not accept the Join if:

- it has no new neighbor capacity of the same type as the querying member, and

- it is not holding a reservation for the querying member. The querying member can claim a reservation if either it has the same name as the reservation, or it has attached a Reservation Record with the same name as the reservation.

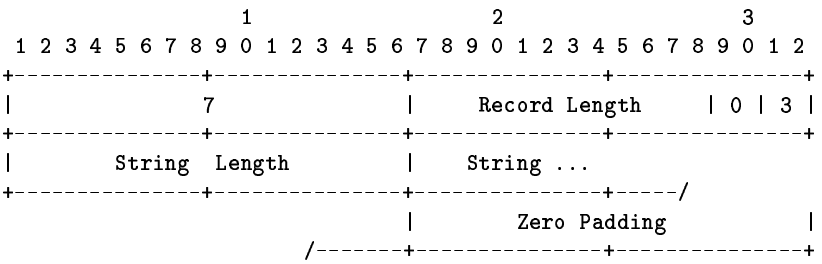
If on the other hand the Join Type of the query is Emergency, the replying member should accept the query even if it has no new neighbor capacity. In this case, however, it should send an Anticipated Quit back to the new child, causing it to find another parent quickly.

5.3.5 Root Path Record

If the Join is accepted, then the Root Path of the replying member must be included with a Root Path Record. The syntax for the Root Path Record is shown in Section 5.2.4. If the Join is rejected, this record is not attached.

5.3.6 Error String Record (7)

If the Join is rejected, a text string describing the reason can optionally be attached to the Message using this Record. It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 7, the Action is 3 (drop with notification), and the Record Length is variable.

String Length: Gives the length, in bytes, of the string (not including padding).

String: The text string, consisting of the printable subset of ASCII characters. It is padded out, if necessary, to an integral 32-bit boundary by zeros.

5.4 Quit Message (3)

This message has three purposes. First, it is used by a child member to inform its parent that it is disconnecting from the parent. This may happen either because the child has found a new parent, or because the child is leaving the group altogether. Second, it is used by a child member to inform its parent that another member will replace it as the parent's child. This is used when a head is being replaced by another. Finally, it is used by a parent member to inform its children of an anticipated quit, in order to give the children a chance to find new parents in a coordinated way.

Unlike most other messages, the Quit message has no Reply form. This is because the receiver of a Quit message has no choice but to accept it. Reception of the Quit query, which must be transmitted via a stream, is acknowledged directly by TCP or yTCP.

The Message Subtype field of the Quit Query must be set to one of three values:

Anticipated (1): This is sent from the parent to the child as a warning that the parent plans to leave the group, and so the child should immediately find a new parent.

Completed (2): This is normally sent from the child to the parent to inform the parent that it is quitting. Normally, the child will have already gone through the entire process of finding a new parent, informing its own children of its new root path, and starting the exchange of data traffic with the new parent. This message simply informs the (now former) parent of the changed status. This message may also be sent from a parent to a child, for instance because errors were received from the child, or because the application related to the parent requested a hard quit. Obviously this should be avoided where possible.

Replaced (3) This is sent from the child to the parent, essentially authorizing the parent to accept the Join from the replacement child. Application frames are continued to be sent from the parent to the child until the replacement child Joins.

5.4.1 Reservation Record (20)

The Completed form of the Quit Message, when sent from child to parent, may contain a single record, the Reservation Record. The record contains the Member ID of the member holding the reservation. This causes the parent, for a short time (on the order of 2 minutes) to reject any join requests from Members other than the one holding the reservation. It also causes the parent to not advertise the reserved capacity in New Neighbor Capacity records.

It is formatted identically to the Member ID Record (Section 5.11.1), except that the Record Type value is 20.

No acknowledgement is received as to whether or not the reservation will in fact be made.

5.4.2 Replacement Authorization Record (26)

This record names the member that is authorized to replace the sender of the Quit message. It is formatted identically to the Member ID Record, with the exception that it has a Record Type of 26.

5.5 Switch Message (4)

The purpose of this message is for a member to arrange to take the place of another member (attach to the other member's parent in lieu of the other member). It is sent to the member being replaced. This is done in order to optimize the topology in the case where the new parent has no new children capacity. The member being replaced normally, but not necessarily, attaches to the member that initiated the switch.

The Message Subtype is treated as a bitmask in the query, as follows:

Coordinated (Bit 9): Set to 1 if this is a coordinated switch. Set to 0 if it is an emergency switch.

Preliminary (Bit 10): Set to 1 if this switch request is preliminary, meaning that the replyer should not actually make the switch, but rather should only indicate if it is willing to make the switch. Set to 0 otherwise.

Reservation (Bit 11): Set to 1 if the replyer should make a reservation for the queryer when the replyer quits its parent. Set to 0 otherwise.

5.5.1 Join Target Record

This record indicates to the replyer which member will become available for it to join. The replyer is not required to join this particular member. In a tree with little remaining new neighbor capacity, however, this is useful information. Note that the join target may not have new neighbor capacity at the time this query is made. Inclusion of the join target in this record implies that the queryer intends to free the new neighbor capacity and reserve it for the replyer.

It is formatted identically to the Member ID Record, with the exception that it has a Record Type of 21.

5.5.2 Switch Message Reply

There may be a significant delay between the switch query and its corresponding reply. This is because the queried member should go through the process of determining if it can find a satisfactory new parent, and joining that parent. The Switch Reply Message assigns the following values to the Message Subtype:

Preliminary Accept (1): The replyer is willing to accept the preliminarily requested switch, but otherwise has done nothing.

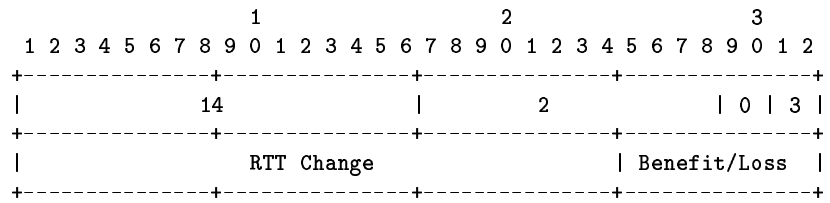
Accept (2): The switch request has been accepted, and the replyer has quit its parent.

Reject (3): The switch request has been rejected. The replyer will not quit the parent.

5.5.3 RTT Change Record (14)

Both the Switch Message query and reply may contain an RTT Change Record. This record gives the change in RTT latency expected by the member transmitting the query or reply were it to make the suggested switch. This is used to help determine the value of the switch, which should result in an overall benefit if it is to be made.

It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 14, the Action is 3 (drop with notification), and the Record Length is 2.

RTT Change: The estimated change in RTT time, in microseconds, that the sender of the record would benefit/lose were it to make the suggested change. The value 0 is used for any changes of less than one microsecond. The value 16777214 is used to encode any changes equal to or greater than 16777214 microseconds (slightly less than 17 seconds).

Benefit/Loss: This field is set to 1 if the RTT Change is a benefit (gets lower), and 0 if it is a loss (gets larger).

5.6 Intent to Join Message (5)

This message is used to insure that a planned parent change will not result in a tree partition/loop. The query is sent HxH along the tree path from the joining member to the prospective parent.

If there is any member along that path that has frozen joins with its neighbor, or if the path itself is specified incorrectly, the reply to the Intent to Join is negative. Otherwise, it is positive, and the joining member can subsequently join its new parent. The reply is transmitted directly from the replier to the querier.

5.6.1 Join Path Record (8)

The syntax of the Join Path Record is identical to that of the Root Path Record (Section 5.2.4), and is not repeated here. The differences are that 1) the Record Type of the Join Path Record is 8 (rather than 5 for the Root Path Record), and 2) the path itself is not to the root but rather to the target of the planned join. The message itself is forwarded member-to-member along the path specified in this record.

Note that the information in the YDP Frame Source Option and in the query message fixed part refers to the original source of the message, not to the member that happens to be forwarding the message.

5.6.2 Intent to Join Reply Message Subtype Values

The reply can be sent by any member in the query's Join Path Record. If it is sent by any member other than the final entry of the Join Path Record (that is, the prospective parent), it must be negative. The prospective parent may

send either a negative or positive reply. The reply is transmitted directly to the original querying member, as specified by the contents of the YDP Frame Source Option and the message fixed part of the query.

The values for the Message Subtype fields are as follows:

Accept (1): Set to value 1 if the Intent to Join is successful.

Other Reason (2): The Intent to Join request was unsuccessful for a reason other than those listed below.

Joins are Frozen (3): The Intent to Join was unsuccessful because the intended join would conflict with a join already in progress.

Join Path Invalid (4): The Intent to Join was unsuccessful because the path in the Join Path Record is invalid (adjacent entries in the Join Path Record are not neighbors).

5.7 Root Path Trace Message (7)

This message is used during an emergency parent change to test for loops or partitions before actually joining the new parent (see Section 8.1). In this case, the query form is initially sent from the member that wishes to initiate the Root Path Trace (the parent-less member) to the member from which the root path will be traced (the prospective parent).

This message may also be initiated by a member that suspects that it is in a topology loop, and wishes to check (for instance, because of an excessive number of Hop Count Expired messages, Section 8.9). In this case, the initiating member is also the member from which the root path is traced, and the member treats it as though it had received it from another member, as described below.

The Message Subtype field must be set to one of the following two values:

Parent Only (1): Forward the message to the current parent only (the parent with which application frames are being exchanged). The message must not be forwarded to a prospective parent.

Parent/Prospective Parent (2): Forward the message to the parent or the prospective parent.

Each member that receives a Root Path Trace query transmits it either 1) to its parent if it has one, or if the Message Subtype allows, 2) to its prospective parent. If the member has no parent/prospective parent to forward the query to, then it considers itself the root, and transmits a Root Path Trace reply to the initiator of the query.

The YDP Frame Source Option and the fixed part of the Root Path Trace message are set by and contain information about the initiator of the query. These parts are not changed as the message progresses along the root path.

The Root Path Trace message contains no records at the time it is initiated. The first member that receives it adds a Root Path record, with itself as the sole listed member. Each member that subsequently receives the message appends itself to the Root Path record. The Root Path Trace reply sent to the initiating member also contains the compiled Root Path record.

In addition to adding itself to the Root Path record, each member also checks it to see if it is already listed. If it is, it transmits a reply to the initiating member (and of course does not forward the query).

5.8 Error Message (8)

This message is used to convey notification of various kinds of errors. It only has the reply form (the message that triggered the Error Message can be thought of as an implicit query). It is always sent datagram transmission mode. A message must never be sent in response to an Error Message.

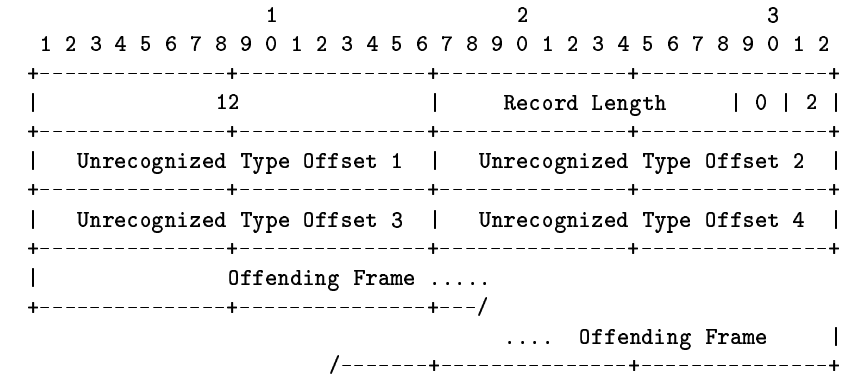
Members must wait some short period of time, on the order of one to a few seconds, before sending the same Error Message to the same recipient.

The Message Subtype field indicates the type of Error Message. There is at this time only one type of error message.

5.8.1 Type Unrecognized (1)

The Type Unrecognized Error Message is sent when a member does not recognize a given YTMP message or record type, and the action for that message or record was either Ignore With Notification (1) or Drop With Notification (3).

The Message contains a single record, the Unrecognized Type Offset record, formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 12, the Action is 2 (drop silently), and the Record Length is variable.

Unrecognized Type Offset: This is the offset, in units of 32-bit words, of the start of the unrecognized Message or Records. The 0th word of the offset is the first word of this (the Unrecognized Type Offset) record. A value of 0 indicates an unused Unrecognized Type Offset field. The first Unrecognized Type Offset is always non-zero.

Up to four offsets can be included, allowing up to four unrecognized items. Multiple unrecognized records could occur if, for instance, the Action of each record was Ignore With Notification.

Offending Frame: This is the frame that contained the unrecognized type, starting from the Frame Header. The whole frame does not need to be included, but everything up to and including the entire last unrecognized record must be included.

5.9 Root Announcement Message (9)

This message is used to discover tree partitions (multiple roots). It is sent by members that believe themselves to be the root of the tree. A member will believe itself to be a root after it finds that all known members register it as their root in their root paths (Section 8.6.2).

This message has both a query and reply form. The query form is used to by a root to announce itself to other potential roots. It is sent unicast, datagram, to a rendezvous if the rendezvous is up. The rendezvous responds with the reply form. Otherwise, the query form is sent broadcast, stream, to all members in the mesh. In this case, there is no reply.

The Message Subtype for the query must take on one of two values:

Root (1): Set to 1 if the member sending the message considers itself to be the root.

Not Root (2): Set to 2 if the member sending the message no longer considers itself to be the root.

There are no message subtypes for the reply form (that is, it is sent as 0).

The reason for using a datagram reply as an acknowledgement rather than a stream is because a host may potentially be a rendezvous for a very large number of group. One reason for this might be because the system where the Rendezvous resides has high reliability and availability. Making this message datagram reduces the number of connections that such a system would have to maintain.

5.10 Pairwise Knowledge Message (10)

This message is used to create or destroy pairwise knowledge between two systems. It is used both by transit members to maintain mesh neighbors (Section 10), and by rendezvous to maintain knowledge about members in the group (Section 7.5). In the former case, the purpose is to insure a well-connected mesh topology (non-partitioned with high probability). In the latter, to insure that the rendezvous know of at least one active group member with high probability.

To create the pairwise knowledge, the query form of the message is sent mesh anycast, datagram. Its purpose is to find a random other member. When done by a non-foot member, it is done frequently (once every few seconds) until the member has a small set number of mesh neighbors (say, 3) that were discovered by its anycast transmission. After this, it is done less frequently—once every few minutes or longer—just to keep the mesh topology well mixed.

Upon receipt of such a query, the recipient checks to see if the sender is already a tree or mesh neighbor. If it is not, then the recipient establishes a connection back to the sender and replies. Subsequently, periodic low-level keep-alives are used to maintain correctness of the pairwise knowledge.

To end the pairwise knowledge relationship, one or the other members/rendezvous transmits a reply message with a subtype of destroy, and tears down the connection.

5.10.1 Message Subtype

The Message Subtype field must take one of three values:

Create Mesh (1): This indicates that the sending system is a transit member, and wishes to establish pairwise knowledge for the purpose of forming a mesh link with the recipient.

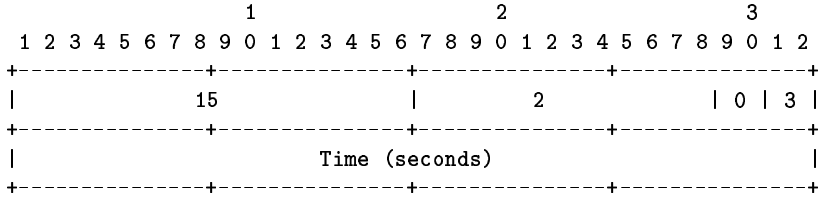
Create Rendezvous (2) This indicates that the sending system is a rendezvous, and wishes to establish pairwise knowledge for the rendezvous's benefit.

Destroy (3): This indicates that the sending member has erased its pairwise knowledge with the recipient. This is only sent if the pairwise knowledge is erased prematurely (before the timeout defined by the Lifetime Record). It is primarily used for the case where the member leaves the group. It is never sent by a rendezvous.

5.10.2 Lifetime Record (15)

The Lifetime Record gives the time, in seconds, that the system transmitting the record will remember the system receiving the record. This record is sent in both the query and reply messages. The actual time that both systems use is the minimum of the two times. (The replying member obviously has no reason to choose a time greater than that of the querying system.)

It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 15, the Action is 3 (drop with notification), and the Record Length is 2.

Time: This is the maximum number of seconds that the sending system will remember the receiving member.

5.10.3 Head Record (24)

This record is included in any Pairwise Knowledge Message with Subtype Create Mesh that is transmitted by a member that is attached to a cluster. It contains the name of the current head.

When a cluster member receives a Pairwise Knowledge Message with this record attached, it compares the name with that of its own head, and ignores the message if they match.

The format of this record is identical to that of the Member ID Record, with the exception that the Record Type is 24.

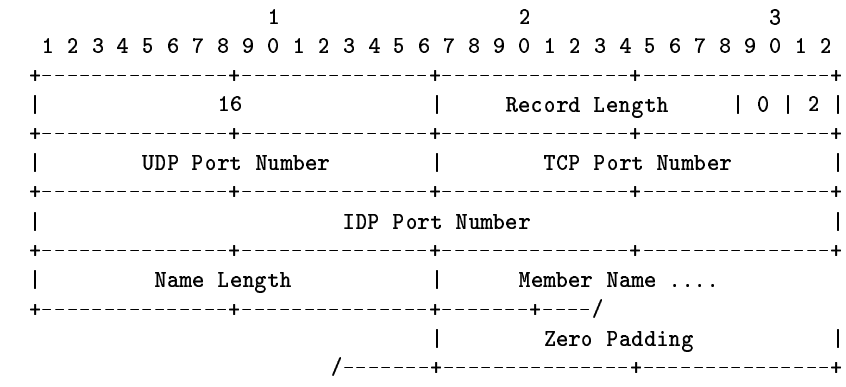
5.11 Member Down Message (11)

This message is used to indicate that a member has been discovered to be down. The discovery will normally have been made on the occasion of a parent member trying to send an application frame to its child, over a stream, and not receiving any packets in acknowledgement. The message is broadcast over the mesh, and lets the children of the child know that their parent is possibly down.

This message has only the reply form.

5.11.1 Member ID Record (16)

This record must be attached to the message. It contains the ID of the member that is down. It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 16, the Action is 2 (drop silently), and the Record Length is variable.

UDP Port Number The UDP port number used by the member to receive datagram frames for this group.

TCP Port Number The TCP port number used by the member for transmitting stream frames for this group. This field must be set to 0 if the member prefers using yTCP over TCP.

IDP Port Number: This is the IDP port number over which the listed neighbor will accept frames. This IDP Port Number is learned from the Listen IDP Port of the IDP header.

Name Length: The length, in bytes, of the Member Name (not including padding).

Member Name: The DNS name of the member. It is padded out, if necessary, to an integral 32-bit boundary by zeros.

5.12 Extend Reservation Message (12)

This message is used by the querying member to extend the reservation being held for it at the queried member. The reservation is extended to last for 2 minutes from the time that the queried member replies. Only the member for which the reservation is being held may request an extension.

This message contains no records. The Message Subtype of the query must be set to 0. The Message Subtype of the reply is set as follows:

Extension Accepted (1): The extension is accepted.

Extension Rejected (2): The extension is rejected.

The message may be sent over a stream or datagram.

5.13 Cluster Announce Message (13)

This message is used by members to announce themselves over their local cluster, for the purpose of auto-configuring the member cluster. It is transmitted over the cluster IP-multicast group, with a time-to-live of 1 in the case of dynamically-configured clusters, and with a time-to-live of 1 or more in the case of statically-configured clusters.

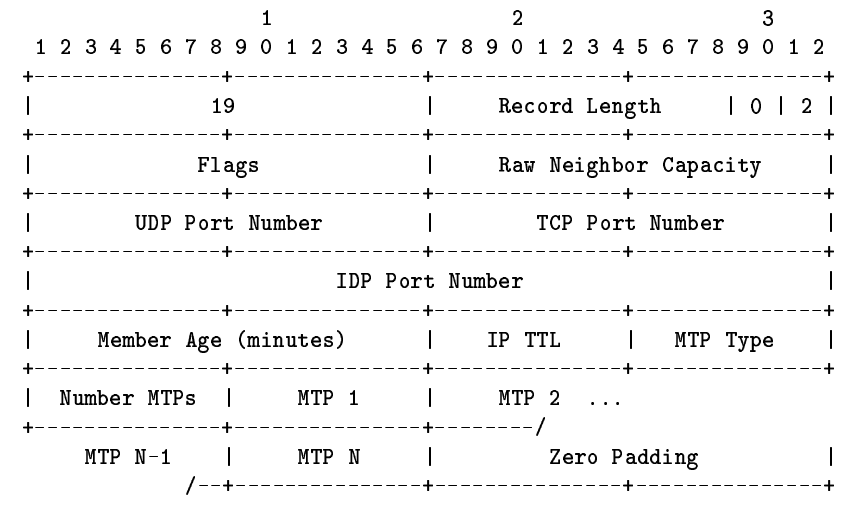
This message is periodically transmitted by the head. It is also periodically transmitted by certain feet. The message has only the reply form.

This message is always transmitted datagram, using the underlying connectionless multicast protocol. Typically that will be IPv4, but may be another protocol where IPv4 is not used. (Note that this is the only message that deals directly with IP-level information. Everything else simply runs over TCP or UDP, and is silent about what internet protocol may or may not be in use.)

The time-to-live used in the IP header of the transmitted message is 1 unless a different value has been advertised by the head (in the IP TTL field of the Cluster Announce Record).

5.13.1 Cluster Announce Record (19)

The Cluster Announce Message contains a single record, the Cluster Announce Record. It contains various information about the announcing member. It is formatted as shown below:



The fields are defined as follows:

Fixed Part: The Record Type is 19, the Action is 2 (drop silently), and the Record Length is variable.

Flags: The following flag bits are defined:

Election State (1): Set to 1 if the cluster is currently electing a head. Set to 0 if the head is considered to be already elected.

Member Status (2): Set to 1 if the member considers itself to be the head. Set to 0 otherwise.

Head Status (3): Set to 1 if the member considers itself to be an administratively assigned head. Set to 0 otherwise.

Anticipated Quit Status (4): This is set to 1 if the member has been a member of the cluster, and is planning to quit it.

Quit Status (5): This is set to 1 if the member has been a member of the cluster, and is now quitting it.

One-way Media (6): Set to 1 if feet must not or cannot transmit frames over the shared media. Set to 0 otherwise.

Bits 7 - 16: Reserved. Transmit as zero, ignore upon receipt.

Raw Neighbor Capacity: This is the total combined raw capacity for both Stub and Transit neighbors. By raw, we mean the sum of both used and available capacity. This is used as the primary criteria in electing a head.

UDP Port Number: The UDP port number used by the member to receive (unicast) datagram frames for this group.

TCP Port Number: The TCP port number used by the member for transmitting (unicast) stream frames for this group. This field must be set to 0 if the member prefers using yTCP over TCP.

IDP Port Number: This is the IDP port number over which the listed neighbor will accept frames. This IDP Port Number is learned from the Listen IDP Port of the IDP header.

Member Age: This is the number of minutes that the member has been a group member. When the value reaches all-ones (representing an age of roughly 45 days), it remains at that value there-after.

IP TTL: This is the value that was used in the time-to-live field of the IP header of the transmitted frame. In the case of static clusters, this value is established by the head and used by the feet.

MTP Type: The multicast transport protocol in use for the purpose of transmitting application frames. This field is valid only when transmitted by the cluster head. The values for this field are identical to those used by the YIDP Protocol field.

Number MTPs: The number of available multicast transport protocols listed. These are the multicast protocols that the member transmitting the message can use. The values are those listed under MTP Type below.

MTP X: The list of multicast transport protocols.

5.13.2 Head Parent Record (24)

This record is attached by a head that itself has a parent. It contains the name of its parent. This informs feet of the parent, so that they can attempt to join the same parent if they later are elected head. It is formatted identically to the Member ID Record, with the exception that it has a Record Type of 24.

5.13.3 Cluster Pairwise Knowledge Records (22, 23)

The head attaches a list of the pairwise knowledge maintained by members in the cluster (see Section 10.2). Each entry is represented by three distinct records, in the following order:

1. The cluster-member end of the pair.
2. The remote (non-cluster) member end of the pair.
3. The remaining lifetime of the pairwise knowledge.

The first two records are encoded using the same syntax as the Member ID Record. They have Record Types of 22 and 23 respectively. The third record is the Lifetime Record (15).

5.14 Summary of Records

The Record Type values are summarized in the following table:

Record	Value	Corresponding Message
Get Member Information Query	1	Get Member Information (query)
New Neighbor Capacity	2	Get Member Information (reply)
Transit Neighbor List	3	Get Member Information (reply)
Stub Neighbor List	4	Get Member Information (reply)
Root Path	5	various
Join Query	6	Join (query)
Error String Record	7	various
Join Path	8	Intent to Join (query)
Group Information	10	Get Member Information (reply)
Statistics	11	Get Member Information (reply)
Unrecognized Type Offset	12	Error (reply)
RTT Change	14	Switch (query or reply)
Lifetime	15	Pairwise Knowledge (query and reply)
Member ID	16	Member Down (reply)
Child New Transit Neighbor Capacity	17	Get Member Information (reply)
Child New Stub Neighbor Capacity	18	Get Member Information (reply)
Cluster Announce	19	Cluster Announce (reply)
Reservation	20	Join (query), Quit (reply)
Join Target	21	Switch (query)
Cluster Pairwise Knowledge (cluster end)	22	Cluster Announce (reply)
Cluster Pairwise Knowledge (remote end)	23	Cluster Announce (reply)
Head	24	Pairwise Knowledge (query and reply)
Replacement	25	Join (query)
Replacement Authorization	26	Quit (query)

6 Major Member States

Figure 6 shows the major member states. A member can act in concurrent multiple capacities, and Figure 6 shows which states a member can be in at any given time. Figure 7 shows the transitions between the major states.

States exist either in the context of the tree and its algorithms, or the context of a cluster and its algorithms. The two algorithm operate independently, with the sole exception that, if a member is a (cluster) foot, then it does not participate in the tree algorithms at all.

There are three transient states:

Newcomer: This is the initial state of a member joining the group. The member is only in this state once.

Orphan: This is the transient tree state. This is the state of a member when it does not have a parent, but neither does it consider itself to be the root of the tree. While in this state, the member is actively searching for a parent.

Candidate: This is the transient cluster state. This is the state of a member when it knows of no cluster head, and is participating in the election of a new head.

There are six steady states:

Root: This is a steady tree state. A member considers itself to be the root when it has exhausted its search to find a parent. In this state, it periodically searches for another root (in case the tree is partitioned).

Child: This is a steady tree state. A member is a child when it has a parent. A member cannot be a child and a root.

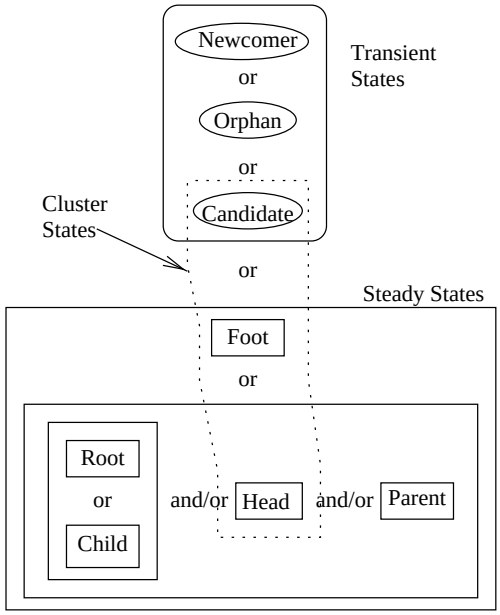


Figure 6: Concurrent Major Member States

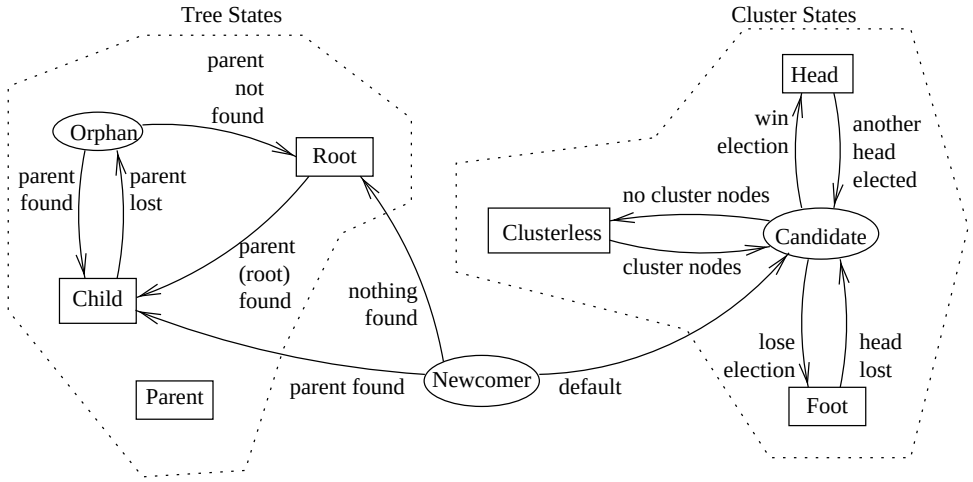


Figure 7: Transitions Between Major Member States

Parent: A member is a parent when it has a child. Note that only a member designated as a transit member can be a parent.

Clusterless: This is a steady cluster state, of sorts. If a member is unable to find any other cluster members, then it considers itself to be clusterless. All it does in this state is listen to the cluster, and periodically transmit to it in search of another cluster member. It also participates in the tree algorithms. The other two cluster steady states exist only when there are two or more members active in the cluster.

Head: This is a steady cluster state. A member is a cluster head when it has won an election. As a cluster head, it is responsible for insuring that the cluster is attached to the tree, and therefore participates in the tree algorithms.

Foot: This is a steady cluster state. A member is a cluster foot when it has not won an election, and another member has. A foot does not participate in the tree algorithms, because the cluster head does this on its behalf.

7 Rendezvous Algorithms

The first thing required to create a group is that one or more rendezvous be invoked. Rendezvous are the first point of contact for newly joining members (newcomers). A rendezvous is invoked by the group's controller application, using the createGroup API call.

The rendezvous must reside on a host with a DNS name matching that of the Group ID's group name. If there are multiple rendezvous, each of their hosts must have the same DNS name. In addition, the DNS reply for the name must include A Records for all of the hosts. Among other things, this allows the rendezvous to find each other.

The rendezvous must communicate knowledge of newcomers to each other, so that all rendezvous can monitor the group membership. They do this through a separate group, called the *rendezvous group*, whose members consist only of the rendezvous. By convention, the group name and port number of the rendezvous group are the same as the group being managed (sporadically called the main group where a distinction is necessary). The subname of the rendezvous group is the concatenation of the main group's subname with the string ".rendezvous".

For example, if the Group ID for the main group is:

```
foo.bar.com/yoidChat/7654
```

then the Group ID for the rendezvous group is:

```
foo.bar.com/yoidChat.rendezvous/7654
```

Each rendezvous keeps a database of (main) group members, called the member list. The member list need not contain all group members, but should contain enough that the rendezvous will know of several alive members at any given time with extremely high probability. If the member list contains no valid members, then newcomers effectively cannot join the group (or, more to the point, a new tree, partitioned from the existing tree, would be created). Probably a rendezvous should maintain knowledge of roughly 10 members at any given time.

A rendezvous may receive the following YTMP messages:

	Message (Type)	Delivery	From	Purpose
1	Get Member Information (Query)	Unicast	Newcomer	Member discovery for group join
2	Pairwise Knowledge (Reply)	Unicast	Member	Maintain knowledge of members
3	Root Announcement (Reply)	Unicast	Member	Tree Partition Detection
4	Get Member Information (Reply)	Multicast	Rendezvous	Inform Rendezvous of members
5	Root Announcement (Reply)	Multicast	Rendezvous	Tree Partition Detection

These are discussed in the following sections.

7.1 Newcomer Query

The first thing a newcomer does is to send a Get Member Information Query to a rendezvous in order to learn about one or more members. The Request Mask of the query has the Transit Neighbor List and Group Information bits set. This message is received by the rendezvous via a stream.

Upon receiving this message, the rendezvous stores the newcomer in its member list. This will typically require deleting the oldest entry in the list to make room for the new one, though the exact management of the list is purely a local matter. Next, the rendezvous must both reply to the newcomer and inform other rendezvous of the newcomer. The sequence with which it does this may vary depending on the situation.

If the rendezvous' member list is empty when it receives the message, then it immediately multicasts a Get Member Information Reply to other rendezvous over the rendezvous group, with the newcomer as the sole entry in the Transit Neighbor List. The rendezvous then waits for a short time (several seconds at least) before replying to the newcomer.

The reason for this wait is to avoid the situation where two different newcomers query two different rendezvous at the same time. If both rendezvous have empty member lists, and both rendezvous reply as such to the newcomers, then the newcomers will not find out about each other, and the tree will start off partitioned. Waiting gives each rendezvous a chance to hear of simultaneous newcomers from other rendezvous.

If the rendezvous' member list is not empty when it receives the query, then it immediately replies to the newcomer. The reply consists of a Transit Neighbor List containing selected members from the member list. The main criteria for selecting the members are similarity of DNS name between the member and the newcomer, and distance between the newcomer and the member (if this is known). These things being equal, the members may be selected randomly, or, if the list is not large, all of the entries can be included.

If the member list is empty, even after waiting to hear from other rendezvous, then the Transit Neighbor List in the reply is empty.

7.2 Destroy Pairwise Knowledge Reply from Member

This message is sent to the rendezvous when a transit member, with which it has pairwise knowledge, has quit the group. After deleting the corresponding member from its member list, the rendezvous may send a Pairwise Knowledge query to find a new member according to the algorithm of Section 10.

7.3 Root Announce

This message is periodically sent by the root of the tree for the purpose of detecting a tree partition. Any member that considers itself to be the root periodically sends this message to a rendezvous. When it is no longer the root, it sends one or a few Root Announcement messages with Subtype Not Root.

The rendezvous maintains a list, called the root list, of roots (usually a list of one). When the rendezvous receives a Root Announcement with a Message Subtype of Root, it adds that member to its root list. When the rendezvous receives a Root Announcement with a Message Subtype of Not Root, it deletes that member from its root list.

Every Root Announcement message received from a member (not a rendezvous) is forwarded to the other rendezvous, multicast, over the rendezvous group.

Every received Root Announcement message (whether from a rendezvous or a member) is also forwarded to all other members in the root list.

7.4 Get Member Information Reply from Rendezvous

This message informs other rendezvous of newcomers, and is discussed in Section 7.1. It is sent as an unsolicited reply. When receiving this message, the rendezvous may add the listed member to its member list if appropriate (for instance, because the list is not full).

7.5 Member Discovery

The member list maintained by each rendezvous should be large enough that at least one of them is still current with very high probability, but not so many that the overhead of keeping them up to date is excessive. 10 entries or so seems appropriate. Note that, in particular, if all of the members in the rendezvous' member list and root list are no longer current, the rendezvous cannot successfully refer newcomers to group members until it receives a Root Announcement from the current root.

If the rendezvous' controller application is also a normal application (that is, has joined the tree proper), then the rendezvous need do nothing special to maintain its member list. The rendezvous, as a regular member, will naturally know of enough other members.

If the rendezvous' controller application is not a normal application, then the rendezvous may either choose to become a member, or may send Pairwise Knowledge queries into the mesh more-or-less as described in Section 10. It would choose the former where the amount of traffic over the group is similar to or less than that required to do the queries. The rendezvous can either first become a member and change strategies if the traffic is too high, or first not become a member, and use information in the Statistics record of Get Member Information Messages to determine if it should become a member.

For phase 1 of implementation, which does not have mesh anycast, the rendezvous should simply periodically find other members in the group in the same way that member members periodically find parent-side members (Section 8.7),

with the exception that the rendezvous should transmit the anycast Get Information Query messages to randomly selected members (rather than just the parent member).

8 Tree Maintenance Algorithms

In the algorithms described in this section, unless otherwise stated, the descriptions are refering to members acting outside the context of a cluster. This includes either transit members that have been elected as the head of a cluster, and members not on a cluster at all. The subsequent section (9) describes cluster operation.

8.1 Obtaining and Changing Parents

Every member considers every other known member to be one of the following:

1. A parent.
2. A prospective parent.
3. A relative.
4. A potential parent.
5. Not a potential parent.

A member considers another member to be its parent if it has either completed a join or become a foot, and it believes its parent to be still up and a member of the group, for instance because it has recently received frames from the parent.

A member considers another member to be its prospective parent if it is actively seeking to join that member. A foot never tries to seek a new parent (though it may of course become the head and then try to find a parent). Otherwise, a member will only seek to join the new parent if:

1. it believes that the new parent's root path does not contain itself, and
2. it believes that the new parent has new neighbor capacity.

These two criteria are called the parent validity criteria.

A member considers another member to be its relative if the parent validity criteria hold, and the relative has agreed to immediately inform the member of any changes in its root path or new neighbor capacity. This allows the member to select a relative as a prospective parent without having to first sending it a Get Member Information query to determine the information. A member will typically try to maintain two relatives. Two members can be relatives for each other.

A member considers another member to be a potential parent if, as far as it knows, the parent validity criteria hold. The potential parent, however, will not inform the member of any changes to its root path or new neighbor capacity, so the information may be out of date. Because of this, a member must send a Get Member Information query to a potential parent to update the parent validity criteria before making it a prospective parent.

Finally, a member considers another member to be not a potential parent if, as far as it knows, the parent validity criteria do not hold.

If a member has a parent and a prospective parent, then the sequence of events for the member are as follows:

1. Join the prospective parent.
2. Stop exchanging frames with the parent and start exchanging frames with the prospective parent.
3. Quit the (former) parent.

At the moment of the second step, the prospective parent becomes the parent, and the member no longer has a prospective parent.

Note that there is a period of time when two (or more) members believe themselves to be the parent of a given (child) member. As long as the child, at any given point in time, only considers one of the members to be its parent, and therefore only transmits and accepts application frames to and from that parent, no loops in the tree will form. Frames sent from the non-parent to the child will simply be dropped. (The child should not send an YTMP Error message for these frames unless it believes that the non-parent should know that it is no longer a child — for instance because it has already quit the parent.)

Note also that, even in the case of stream frame transmission, frames may be dropped or duplicated as a result of the switch. This is because the frames being sent by the two parents will not in general be synchronized in time. Normally, however, YDP is able to prevent lost or duplicated frames based on the byte sequence numbering.

There are three conditions under which a member may join a parent:

Independent: This is the case either where the member has no children, or where the only "child" is the feet for which it is the parent. This is the simplest case, because any transit member in the tree is a valid potential parent. This case occurs whenever a member is joining a group for the first time. It is also always the case for stub members.

Dependent Coordinated: The member has children and a current parent as it obtains a new (prospective) parent. In this case, the member can coordinate the change by sending an Intent to Join message to the prospective parent through its current parent, and by freezing Intent to Join messages coming from the parent. This would be the case where, for instance, the member found a potential parent that was closer than its current parent. It also occurs when a member receives an Anticipated Quit message from its parent.

Dependent Emergency: The member has children but no current parent. In this case, the member cannot coordinate the parent change using the Intent to Join message. It does, however, check for a partition by initiating a Root Path Trace message from the prospective parent. This case occurs when the (now former) parent of a member has crashed and it is detected by the member. It also occurs when a member receives a Completed Quit message from its parent before it has had a chance to obtain a new parent coordinated.

In the following sections, for each of the above three cases, the algorithm for how a member obtains the parent is given.

8.1.1 Independent Parent Change

In this case, the member simply transmits a Join query to the prospective parent as described in Section 5.3. Upon success, it transmits a Completed Quit Message to its old parent.

8.1.2 Dependent Coordinated Parent Change

There are two cases where a dependent coordinated parent change may take place. In the first, a member receives an Anticipated Quit Message from its parent. In the second, a member independently decides to select a new parent.

In the first case, upon receiving the Quit, the member should wait for a short, randomly selected period of time, between zero and ten or so seconds, before doing anything. This is because the member's siblings will presumably also be changing parents. Skewing the times at which each sibling changes parents smooths out the resulting activity a bit.

Except for this random wait, the procedure for obtaining the new parent are the same for either case. The basic steps are:

1. Using the Get Member Information query/reply, verify that the prospective parent is valid (parent-side), and optionally verify that it has capacity.
2. Freeze reception of Intent to Join messages from the (current) parent.
3. Send an Intent to Join message to the prospective parent via the path through the (current) parent.

4. Receive a positive reply for the Intent to Join directly from the prospective parent.
5. Send a Join directly to the prospective parent.
6. Receive a positive reply for the Join. Save the new parent and root path information.
7. Move traffic flow from the (now former) parent to the prospective (now current) parent.
8. Send a Completed Quit Message to the former parent.
9. Send an uninitiated Get Member Information reply to any children informing them of the new root path.
10. Transmit all reverse-path enabled Source Codes for sources on the child-side to the new parent.

The steps are described in more detail here:

Unless the member has very recently received a Get Member Information reply from its prospective parent, it should send a Get Member Information query requesting the prospective parent's neighbor capacity and root path. In particular, if the member was forced to abort and restart its attempt to obtain the prospective parent (for instance, because of a negative reply from the Intent to Join), it must get a new Get Member Information reply from the prospective parent. If the reply indicates that capacity exists, and that the root path does not contain the member, then the member can proceed.

Next, the member freezes Intent to Join messages coming from the parent. This means that any Intent to Join received from the parent are not passed on, and a negative reply to the Intent to Join is sent to its source.

Next, the member transmits an Intent to Join to its parent. The path in the Join Path Record consists of the concatenation of the member's root path and the prospective parent's root path, minus redundant hops. For instance, assume that the member's root path is N-A-B-D-R, where N is the member, A is N's parent, and R is the root. Assume that the prospective parent's root path (as learned from the Get Member Information reply) is P-C-B-D-R. Then, the join path is N-A-B-C-P. Member D and the root R are deleted from the concatenation because they are redundant.

If a negative reply is received from any recipient of the Intent to Join, the member must temporarily abort the procedure. Also, if no reply is received after a suitable time (1/2 minute or so is probably plenty), the member also aborts the procedure. To do this, it first unfreezes the Intent to Join block from its parent (i.e., it again allows Intent to Join messages from its parent). It sets a random timer, probably ranging up to 1/2 minute or so for quit-induced parent changes, longer for independent parent changes.

After the timer expires, it may again attempt to obtain the prospective parent, starting with the Get Member Information message.

If the Intent to Join is accepted, the prospective parent opens a stream with the member to send the Intent to Join reply. Future messages exchanged with the prospective parent take place over this stream.

At this point in time, the member may safely join the prospective parent. This is true even if the parent is now found to be unreachable, or if a Completed Quit is received from the parent. In other words, after this point in the process, there is no need to use the Dependent Emergency procedure even if the parent is lost. This is because there is no longer any need to send messages through the parent, and because the loop-free-ness of the prospective parent has by now been established.

The member then sends a Join query to the prospective parent. The Join Type of the query must be Coordinated.

If the Join reply is positive, then the member modifies its state to indicate the new parent, and changes its own root path to that of the parent plus itself. If the Join reply is negative, then the member aborts the attempt as described above with the Intent to Join message failure. The member should also probably pick a new prospective parent, or, if the change was not quit-induced, stop trying to change parents altogether.

After switching parents, the member should change traffic flow from the old parent to the new parent. This means that multicast or anycast frames received from a child should now go to the new parent and not the old one. Frames received from the new parent should be forwarded as normal. Frames received from the old parent should be silently discarded.

Next, the member sends a Completed Quit Message to the former parent. This is sent over the still-established stream. After this, the member has no neighbor relationship with the former parent, and should tear-down the connection

with it. Tree-bound application frames received from the former parent after this should trigger an Invalid Neighbor Error Message.

Simultaneous with this, the member must inform its children of its new root path. This is done by sending an uninitiated Get Member Information reply to each transit child over the established stream. The Get Member Information reply contains the Root Path record. Each child must set its new root path, and likewise inform their transit children of the new root path, and so on.

Finally, also simultaneous with this, the member must inform its new parent of the reverse-path enabled Source Codes for members reachable via its children. To do this, it uses an unsolicited Get Member Information reply with Header Option Records containing Source Code header options.

Note that the member must also, over time, convey the non-reverse-path enabled Source Codes to its new parent. However, these can be conveyed at the time application frames from those sources are received and transmitted by attaching the Source Code Option in the usual way.

If at any time before receiving the positive Join reply from the prospective parent, the member receives a Root Path Trace with a Subtype of Parent/Prospective Parent, the member should abort the procedure, and restart after a random time delay, as described above. This essentially gives priority to the member that initiated the Root Path Trace to complete its parent acquisition. The Root Path Trace of course should be forwarded to the parent as normal.

8.1.3 Dependent Emergency Parent Change

This procedure is used by a member to obtain a new parent in the case where the member has no current parent through which to transmit an Intent to Join. This can occur either because the parent is unreachable (i.e. no TCP connection can be established), or because a Completed Quit has been received from the parent.

The basic procedure in this case is:

1. Using the Get Member Information query/reply, verify that the prospective parent is valid (parent-side), and has capacity.
2. Initiate a Root Path Trace message from the prospective parent.
3. Receive a no-loop indication from the Root Path Trace reply.
4. Send a join directly to the prospective parent.
5. Receive a positive reply for the Join. Save new parent and root path information.
6. Initiate traffic flow with the new parent.
7. Send an uninitiated Get Member Information reply to any children informing them of the new root path.
8. Transmit all reverse-path enabled Source Codes for sources on the child-side to the new parent.

The basic difference between the dependent emergency and dependent coordinated cases are that the emergency case uses a Root Path Trace to check for loops, while the coordinated case uses an Intent to Join. The Root Path Trace method is in a way more intuitive than the Intent to Join method, and could in fact work with the coordinated case, but it places a heavy load on the root and members near the root.

The Intent to Join, on the other hand, cannot easily be used in the emergency case because there is no parent to send the message to. In theory, we could try sending it instead to the parent's parent (and to the next one in the path if that member is unavailable), but this approach gets almost arbitrarily complex.

Therefore, we use the two different approaches for the two different cases. In what follows, we only discuss the differences between the two approaches. The common steps should be executed as described for the dependent coordinated case above.

The joining member may choose to try to join the potential parent even if it advertises no new neighbor capacity. This is because it may take some time to find a potential parent with new neighbor capacity, during which the member will not be on the tree. By using the Emergency Join Type of the Join query, the joining member can force the prospective parent to accept the Join even if it has no neighbor capacity.

The Root Path Trace sent to the prospective parent must have a Message Subtype of Parent/Prospective Parent. If the Root Path Trace indicates a loop, the member should temporarily abort the procedure, and retry after a small random time period (approximately 1/2 minute). A common source of loops is expected to be the case where multiple children of the former parent are trying to join new parents at the same time, and these joins taken together form a loop. This is why the Root Path Trace includes prospective parents—it detects potential future loops.

If the Root Path Trace succeeds (shows no loop), the member sends a Join query to the prospective parent. The Join Type is Emergency. From here, the member proceeds as described for the dependent coordinated case above.

Note that, if a member receives a Parent/Prospective Parent Root Path Trace during this procedure, it forwards it to its prospective parent, but otherwise continues the procedure as normal.

8.2 Neighbor Capacity Search

There are two general circumstances in which a member is searching for a new parent. One is the background activity done by a member that already has a parent and a couple relatives, and is simply looking for better ones (see Section 8.7). The other, covered in this section, is where a member either has no parent, or has a parent but does not have enough relatives.

In this latter case, it is important to find a parent/relative as soon as possible. (To simplify the remaining, we refer only to searching for a parent, with the tacit understanding that this includes the case of looking for a relative.) The appropriateness of the parent (for instance, how close it is) is less important. In certain scenarios, however, finding a parent with any new neighbor capacity at all may be difficult. An example might be an audio-video conference among members that are constrained in their processing ability or output bandwidth. In this scenario, most members may already be at capacity, and cannot accept any new neighbors without degrading performance for existing neighbors below acceptable levels.

To cope with this scenario, child members provide information to their parents as to whether they or one of their offspring have new neighbor capacity (using the T and S bits of the New Neighbor Capacity Record). This allows the searching member to quickly, through a series of Get Information Queries down the tree, find a member with new neighbor capacity, if there is one.

Even if there is no member with new neighbor capacity, the searching member may still be able to join if it itself has new neighbor capacity (Figure 8).

The algorithm for searching out new neighbor capacity is straight-forward. At the start, the searching member will generally have at least one entry in the parent-side member list, and therefore will know of at least one potential parent. In the case of a newcomer, this is known from the Get Member Information query made to a Rendezvous. In the case of a previously-attached member, this is known through its normal tree probing activities (Section 8.7). In the case where the parent-side member list is empty, the member must become a root (Section 8.6.1).

In any event, we assume here that the parent-side member list is not empty. The searching member sends a Get Member Information query to one of these members, in search of a prospective parent. The member selected should be the one most likely to have New Neighbor Capacity or Child New Transit or Stub Neighbor Capacity. This might be, for instance, the member that has most recently reported such capacity. Several members can be queried at once, to more quickly find a member with capacity (at the expense of sending "unnecessary" queries).

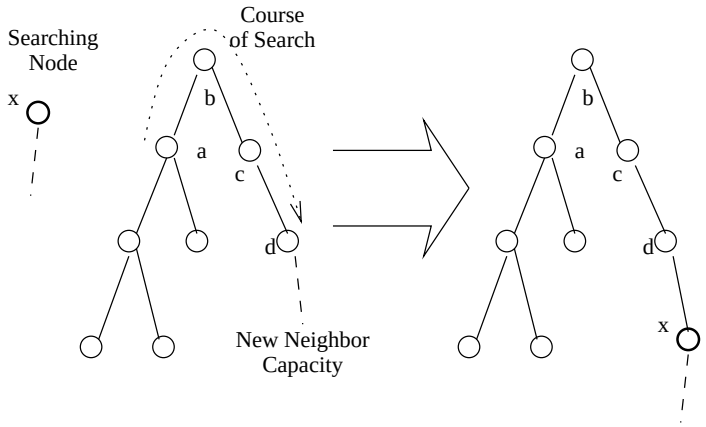
The Get Member Information query requests the potential parent's Root Path, New Neighbor Capacity, and Child New Transit Neighbor Capacity or Child New Stub Neighbor Capacity, depending on whether the member is a transit or stub.

The purpose of the Transit Neighbor List is simply to learn of additional potential parents to be added to the list.

If the Root Path contains the searching member, then it is no longer parent-side, and must be deleted from the list of potential parents. Otherwise, the members in the root path are added to the list of potential parents, if they are not there already.

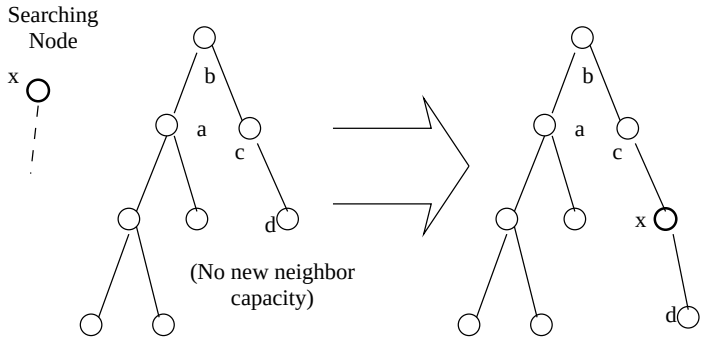
If the queried member has new neighbor capacity, it is made a prospective parent, and joined as described in Section 8.1.3. If it does not have new neighbor capacity, but lists one or more children in its Child New Transit/Stub Neighbor Capacity record, then these members are added to the list of potential parents, and one of them is queried.

As long as queried members reply that they themselves have no new neighbor capacity, but one of their offspring does, the search continues depth-first. That is, the searching member queries members deeper in the tree (towards



Case 1: New Neighbor Capacity Exists

Searching Node joins node with new neighbor capacity as a leaf.



Case 2: No New Neighbor Capacity Exists

Searching Node inserts itself into the tree using its own new neighbor capacity.

Figure 8: Searching for New Neighbor Capacity

the leaves) before querying other members.

At some point, the searching member may have exhausted all known potential parents (that is, queried them all and found none with either new neighbor capacity or offspring with reported new neighbor capacity), including potential parents added to the list from the learned root paths.

At this point, if the searching member itself has new neighbor capacity, then it can join the tree by inserting itself between an existing pair of neighbors. For this, it selects the closest known potential parent, and makes an emergency switch with one of the potential parent's children (Section 8.4). (Note that there is some risk in making this switch. It is possible that internet path between the searching member and its new parent or its new child is of unacceptable quality, thus making the overall situation worse.)

If the searching member itself has no new neighbor capacity, and has no children of its own, then the searching member cannot (re)join the tree. It should wait some period of time and try again. If the searching member itself has no new neighbor capacity, but it does have children of its own, then it must send quit messages to its children. This will cause them to try to join the tree themselves. It may also make any new neighbor capacity available among the searching member's offspring available for the searching member at a later time.

Note that the member should only quit its children if it knows itself not to be the root of the tree. It will know this

because it knows of potential parents (members that are up and do not list it in their root paths). If the member knows of no potential parents, and therefore thinks that it may be the root of the tree, then of course it should not quit its children.

8.3 Relatives

A relative of a member A is another member that agrees to inform member A immediately should either its root path or its new neighbor capacity change. This allows member A to know at any instant in time, with high probability, if the relative can validly become a potential parent.

Relative is not a bi-directional relationship. The fact that member B is a relative of member A does not mean that member A is a relative of member B. (In this sense, relative is a poor choice of term, because when applied to humans it is bi-directional, however much we may sometimes wish that were not the case.) Two members can, however, both be relatives of the other.

Two members that have a relative relationship in either or both directions maintain a connection, which means that they ping each other periodically to insure the other is still up.

Once a member A decides that another member B should be a relative, member A established a connection with member B, and sends it a(nother) Get Member Information query, but this time with the Receiver Relationship field set to "Relative". If member B agrees be member A's relative, it keeps the connection up, and sends a reply. The Receiver Relationship field of the reply is set to "None". Otherwise, member B tears down the connection.

(I think, by the way, that this is an ugly mechanism, and should be changed to an explicit relative request query/reply. Currently, however, this is how the code works.)

If member B in turns wants to make member A a relative, it does the same thing, except over the existing connection.

8.4 Member Switch

If a tree is running at or nearly at capacity, then in order for members to find better parents, or indeed to find any parents at all, they may have to arrange a switch with another member. This section describes the algorithm for switching.

There are three cases of interest:

- Emergency Switch,
- Coordinated 2-way Switch,
- Coordinated 3-way Switch.

8.4.1 Emergency Switch

Like the emergency join, the emergency switch happens when the switching member has no parent. The switching member will also have already made an exhaustive attempt to find a member with new neighbor capacity before attempting the emergency switch (as described in Section 8.2). Finally, the switching member must itself have new neighbor capacity to offer to the switched member.

The emergency switch is illustrated in Figure 9. Here, Member x is the switching member, Member y is the switched member, and Member c is the parent of Member y. (In this figure, as well as in Figure 10, the higher neighbor is the parent of the lower neighbor.)

Member x first does an Emergency Join with Member c. Member c is generally forced to accept such a join, and assumes that Member x will try to find a new parent, or arrange the appropriate switch, as soon as possible. After joining Member c, Member x immediately requests an Emergency Switch of Member y. The Switch message sets the Coordinated bit in the Message Subtype to 0 (emergency), the Preliminary bit to 0 (not preliminary), and the Reservation bit is 0. There is no RTT Change Record attached to the query. Member c attaches a Join Target Record with itself inside.

Upon receiving this query, Member y must join Member and x and quit Member c using a coordinated join procedure.

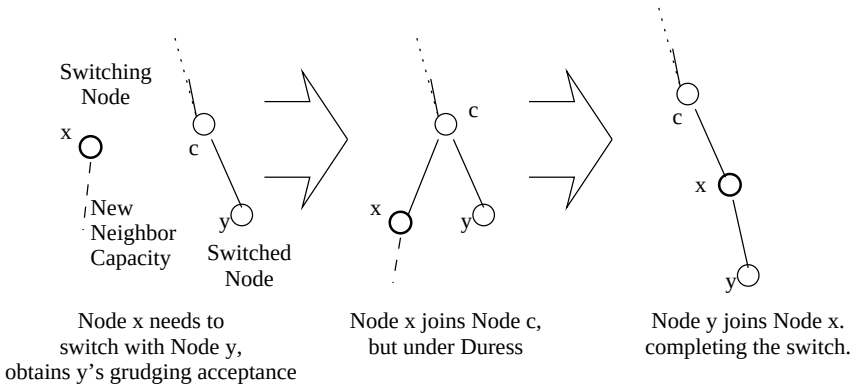


Figure 9: Emergency Switch

8.4.2 Coordinated Switches

A coordinated switch is made when the switching member has a current parent, and wants to join a better one. There are two types of coordinated switches, the 2-way switch and the 3-way switch. The 2-way switch is made when the switching member has new neighbor capacity, and a 3-way is made when it doesn't.

The 3-way switch is in fact two consecutive switches, the first providing the switching member with new neighbor capacity, and the second allowing the switching member to make the desired switch. A 2-way switch is nothing more than the second switch of the 3-way (see Figure 10). For that reason, we start with a description of the 3-way switch, following that with the part common to both types of switches.

The goal of the first half of the 3-way switch is for the switching member (Member x) to obtain new neighbor capacity for itself, which it can subsequently offer to the ultimate member it wants to switch with (Member y). It does this by shedding one of its own children (Member d). Assuming that Member x has multiple children, it must determine which to shed. It does this by ranking them in order of closest (smallest latency) to furthest, and trying to switch with them in that order.

Although Member d may have multiple potential parents of its own, Member x must offer its own parent (Member a) to Member d as a potential parent. If Member a has no new neighbor capacity, then Member x must find some other member that does (Member b) before it can proceed. Member b does not need to be close to Member x, because it will be used only as a temporary parent for the 3-way switch.

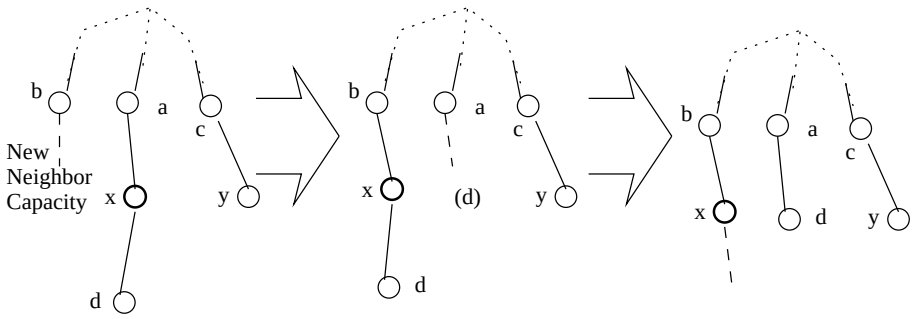
Assuming that Member x has found Member b, it then sends a Switch Message query to it's child Member d. The Subtype of the query is coordinated, preliminary, with no reservation required. The Join Target Record is attached, and contains the parent, Member a. In addition, the RTT Change record is attached, and contains the benefit that Member x expects to see after it has joined its desired potential parent Member c. In other words, it contains the difference of the Member x/Member a RTT with the expected Member x/Member c RTT.

When Member d receives this query, it must determine if the loss it will experience by joining Member a or some other member (if any) is offset by the gain experienced by Member x. In deciding whether to allow the switch, Member d should consider both the overall improvement to the topology (if any) and the negative impact it would experience itself. How exactly to weigh these two is an issue for further study.

Member d determines the expected RTT with Member a by sending one or more Get Member Information messages to Member a. It may also already know the RTT to other potential parents. It selects the potential parent with new neighbor capacity (including Member a, which may not have new neighbor capacity) that has the lowest expected RTT. If Member d determines that this new RTT is acceptable, it sends a Switch reply to Member x with a Subtype of Preliminary Accept. It attaches an RTT Change Record indicating the benefit or loss that Member d will experience by joining the selected potential parent.

If Member does not except the proposed switch, it send a Switch Reply with Subtype Reject.

Assuming that a Preliminary Accept was sent, Member x must now determine if any of the children of its desired potential parent, Member c, will accept the switch.

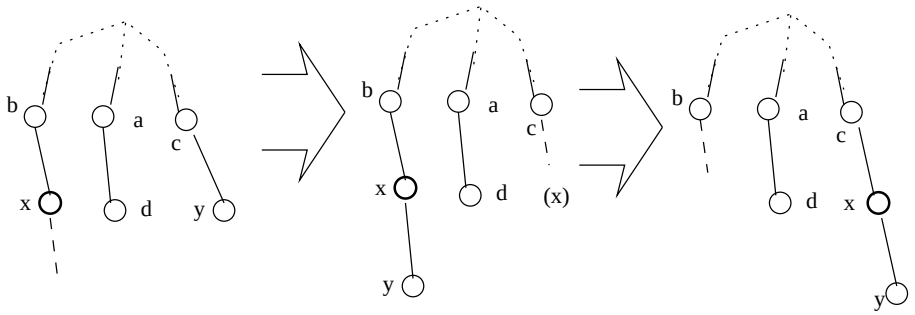


Node x wants to switch with Node y, but has no capacity. Node x requests that Node d join a, and that Node y join it. Nodes d and y happily accept.

Node x quits Node a, reserving a space for Node d. Node x joins Node b.

Node d joins Node a.

First Half of Coordinated 3-way Switch



Node x requests switch with Node y. Node y happily accepts.

Node y quits Node c, reserving a space for Node x. Node y joins Node x.

Node x joins Node c

Coordinated 2-way Switch, and
Second Half of Coordinated 3-way Switch

Figure 10: Coordinated Switch

To learn this, Member x first requests a Transit Neighbor List from Member c using the Get Member Information message. The transit neighbor list in the reply contains the estimated distances to the children. The best likely candidate for a switch is the one furthest from the potential parent. Taking the potential parent's children in order of furthest to closest, Member x sends Switch queries.

The Switch query, here sent to Member y, has a Subtype of coordinated, preliminary, with reservation required. The Join Target Record is attached, and contains Member x itself. In addition, the RTT Change record is attached. This time, however, it contains Member x's expected benefit, minus the loss expected by its child Member d. Member y is thus evaluating the overall benefit/loss of both of the switches.

Member y goes through the same process as described for Member d above, and replies with either a Preliminary Accept or a Reject. The reply, however, does not contain an RTT Change Record. Assuming it is a Preliminary Accept, Member x can now move forward with the first of the actual switches. Note that a Preliminary Accept does not guarantee that the actual switch will in fact be accepted later on.

Assuming that Member x's parent, Member a, does not itself have new neighbor capacity, Member x first does a coordinated join to the temporary parent it has selected, Member b (Section 8.1.2). The Quit with Member a requests

that it reserve the new neighbor capacity freed up by Member x for Member d.

After this, Member x sends a Switch query to Member d. This is identical to the previously send Switch query, except that the Preliminary bit of the Subtype field is set to 0. Member d then attempts to do a coordinated join with whatever potential parent it has selected. If it has selected Member a, however, and fails to make the join within something under 2 minutes (for instance because the Intent to Join failed), it must send an Extend Reservation Message to Member a.

Assuming that Member d successfully joins another member, it sends a Switch reply to Member x with Subtype Accept. The RTT Change Record may have the same value as the previous Switch reply, or may be modified to reflect any improvement in the estimate. Member d of course then also Quits Member x.

At this point, Member x now has new neighbor capacity, and can start the second half of the 3-way switch. Except where mentioned otherwise, what follows from here is also the procedure for a 2-way coordinated switch. This is shown in the lower half of Figure 10, with Member x as the switching member, and Member y as the switched member. Member d no longer has a role in the process. For the 2-way coordinated switch, it is assumed that Member x has already rank ordered the children of Member c according to the RTT between Member c and its children, and has selected Member y as the switched.

Member x next sends a Switch query to Member y. It is the same as the Switch query sent from Member x to Member y described above, except that it is not preliminary. The RTT Change is also calculated in the same way, though it may have a different value from before based on new information.

Upon receiving the query, Member y tries to do a coordinated join with the prospective parent it has chosen (either Member x itself or some other member). Assuming that it succeeds, when it Quits Member c, it reserves new neighbor capacity for Member x. Member y then sends a Switch reply with Subtype Accept to Member x. The reply has no attached record.

At this point, Member x does a coordinate Join to Member c. As with Member d's coordinated Join described above, Member x must send an Extend Reservation Message to Member c if it will not be able to accomplish the Join within 2 minutes.

Note that at any time in the process, an attempted Join may fail, or a member may decide after all that it cannot accept a switch request. This means that part, but not all, of the switch will have occured. There are no specific mechanisms for attempting to undue an aborted coordinated switch. Understanding that the overall topology may be worse off than before the switch was attempted, we simply allow the usual mechanism of searching for better parents to continue its course and hopefully improve things.

8.5 Join Group

To join an existing group, a newcomer must know the Group ID, including the Group Name, Group Subname, and the Group Port. These are learned via some mechanism outside the scope of this specification.

A newcomer may choose to first determine if there is a local cluster to join, or first try to find any parent in the tree, or do both simultaneously. Either way, the two activities are independent, except for the fact that, if the newcomer finds a local cluster and it is not elected as head of that cluster, then it does not need to find another parent. The remainder of this section assumes a member that is not a foot.

The first thing the newcomer does is transmit a Get Member Information Query to one of the rendezvous for the group. The rendezvous is found by doing a DNS lookup on the Group Name. The Get Member Information Query is sent unicast over a (TCP) stream, using the Group Port as the destination port number. The query requests a Transit Neighbor List and Group Information, as described in Section 7.1.

A "Phase 0" implementation can simply join one of the listed members at this point, and skip the remaining.

Upon receiving the Get Member Information Reply from a rendezvous, the newcomer has a list of potential parents in the tree. The goal of the newcomer is to find a parent as quickly as possible (versus finding the best possible parent). It does this by following the algorithm of Section 8.2.

If the newcomer finds a potential parent with new neighbor capacity of the appropriate type (stub or transit), it immediately transmits a Join Query to the member. This is sent over the same stream that was setup to transmit the Get Member Information messages.

The Member Type of the Join Query is set to that of the newcomer (stub or transit). The Join Type is set to Coordinated. If the Join Reply is positive, then the newcomer appends the parent to the received root path to create its own root path.

If the Join Reply is negative, or if for any other reason the join failed (the potential parent became unreachable during the join process, the potential parent had no new neighbor capacity, etc.), the newcomer continues the search as described in Section 8.2.

8.6 Root

A root member is defined as a member that does not have a parent, and has exhausted all reasonable possibility of finding a parent. Many members will, for short periods of time, have no parent, for instance because their parents crash. Such members are usually able to find new parents quickly. They are referred to as *orphans*. When a member first loses its parent, it considers itself an orphan, and tries hard to find a new parent. If an orphan cannot find a parent after a concerted attempt, it considers itself to be the root.

A root periodically searches for another root, but otherwise does not try to find a parent per se.

The following two sections describe the procedures taken by an orphan and a root.

8.6.1 Orphan

When a transit member first joins a tree (as a newcomer) it discovers one or more other members in the tree (if it is not the first member to join). All of these members are initially parent-side members. Through the normal process of searching for a better parent, each member potentially learns of other parent-side members. All of these members are potential parents, and are kept in a list of parent-side members.

Starting with this list, the member searches for a potential parent that has new neighbor capacity, as described in Section 8.2. If the search succeeds, and the member joins a parent, then it is no longer an orphan. If the search ultimately fails, it is either because there is no spare capacity in the tree, or because the member is the root. The former is the case if the member knows of parent-side members.

If the member does not know of any parent-side members (all known potential parents turned out to no longer be parent-side members), then the member assumes that it is the root of the tree. It may in fact not be the true root of the tree, because there may be parent-side members that it does not know about. It, however, has no way of knowing this, and therefore must assume that it is a root.

8.6.2 Root

Once an orphan decides that it is a root, all it can do is periodically send Root Announcement Messages to a rendezvous, if it is up, or broadcast them to the mesh otherwise (phase 4 only). On the assumption that other roots are doing the same thing, they will find each other with high probability. The only time they won't find each other is when both 1) the rendezvous are unreachable (or, different rendezvous are reachable only by different roots), and 2) the mesh is partitioned.

If after a small number of periodic Root Announcement messages to the rendezvous the root receives no reply, it must consider the rendezvous to be down. Thereafter, it periodically sends the Root Announcement message to both the rendezvous and the mesh. Any time it is receiving replies from the rendezvous, it may stop sending to the mesh. The periods for the two Root Announcement messages, as well as the number of attempts before which the root starts sending to the mesh, are issues for study.

At any time the root may learn of a potential parent, either from a Root Announcement message, or some other message such as a Get Member Information message. When this happens, the root again considers itself to be an orphan, and operates as described in the previous section (8.6.1).

8.7 Background Parent Search

If a member is not a root, then it is a child of some member. If it has no prospective parent (is not actively changing parents), then it is said to be in steady state. Even in steady state, each child (hereafter called a member) is continuously searching for a better (closer) parent. (Note that this is not true for feet, which assume that the head is the best possible parent.)

The basic process goes as follows:

1. Periodically find a random parent-side member by sending a Get Member Information query tree-anycast to the parent. The period used increases over time, as closer parents get harder and harder to find.
2. Add or refresh the found potential parent in the parent-side member list. (If a potential parent is to be deleted from the list, which to delete is an issue for study.)
3. Determine the distance to the new potential parent by sending, unicast, one or more Get Member Information queries, and timing the replies.
4. If the new potential parent is found to be significantly closer (in latency) than the current parent, make it a prospective parent and try to acquire it according to the procedure of Section 8.1.2.

In what follows, the above basic steps are described in more detail.

The Get Member Information message tree-anycast to the parent should request statistics information. This will give some general knowledge about the discovered member, such as how long it has been a member of the tree (which is probably a good predictor of how long it will continue to be a member of the tree).

We desire the contents of the parent-side member list to have two attributes. First, the members listed should be up and still members with high probability. This is important for being able to quickly find a new parent. Second, the members should be relatively close.

To some extent these two goals must conflict. Assuming that different members will have different membership times and different reachabilities, it is important to know of at least a few members that will be up with high probability even though they are far away. Once this is satisfied, however, it is useful to also keep a number of nearby members in the list, even if their future status is less certain. Some care must therefore be taken in maintaining the list.

An open issue is that of how to be confident of the measured latency to the discovered member. There are two problems here. The first is that of filtering out the noise from the measurements to get an accurate reading. Significant amounts of research have been done in this area, so we should be able to borrow something decent. The second is knowing what the effect of joining the new member would be, since it means that the new member would have to forward that much more traffic.

A particularly potent example is that of a PC sitting at the end of a modem receiving a video stream. Assuming the PC has no children, the delay to that member might be reasonably small when the measurement is being made, but would grow significantly as soon as the member has to transmit the video stream back over the modem connection. (One could argue that such a member shouldn't be a transit member in the first place, but that is another matter.)

Gathering the right statistics, such as a member's Maximum Available Bandwidth, may help with this sort of problem. The Get Member Information messages used to make the measurements should also request root path and new neighbor capacity information.

If the newly discovered potential parent is found to be close enough to join, and it has new neighbor capacity, then it is considered a prospective parent, and the process for joining is started (Section 8.1.2).

If the potential parent doesn't have new neighbor capacity, then the member should determine if a coordinated switch, as described in Section 8.4, is appropriate. Before doing this, it may also check the distance to the children of the desired parent, on the assumption that those children will be relatively close to the desired parent, and therefore relatively close to the member.

The frequency with which new potential parents should be discovered (using the anycast Get Member Information message) should decrease over time. This is because, as the member finds better and better parents, it will become progressively harder to find still better parents. It will also become less important to find still better parents, as each successive improvement will on average be less than the last.

Members should probably implement an algorithm that increases the time between searches with each successive failure to find a better potential parent, and decreases the time each time a better parent is found. This way, a member will aggressively search out better parents at first, when much is to be gained by doing so, and slow its activity over time.

Finally, members should place a ceiling on the amount of traffic they are willing to spend on this background optimization activity. In particular, a member should not devote more than something like 10% of the total traffic of the tree (including both application and control frames) to trying to improve it.

8.8 Quit Group

When a member is to quit a group, it should, wherever possible, arrange for effected members to react appropriately in advance. This is most important for children of the quitting member, since they must find new parents, and should do so with a minimum of disruption to the flow of frames. The quitting member, however, should also inform its parent, and any systems for which it has unexpired pairwise knowledge (rendezvous and other transit members).

Upon deciding to quit, the member should first, if at all possible, send a Quit query to each of its children, with a subtype of Anticipated. After it receives Quit messages from all of its children, it may send a Quit message of Completed to its parent, and Destroy Pairwise Knowledge replies to all systems for which it has unexpired pairwise knowledge. Until it receives Quit messages from all of its children, however, it should continue to forward frames.

If after some reasonable period of time some children have still not quit, the member may sent Completed Quit messages to its remaining children and stop forwarding frames.

8.9 Excessive Hop Count Expired

A member can generally detect that it is in a loop by having recently generated multiple hop count expired messages to different members in a relatively short time period. Such a member can test for a loop by initiating a Root Path Trace message from itself. This Root Path Trace message must have the Subtype set to Parent Only. This is because the trace is checking for an active application frame loop, which won't occur with a prospective parent.

If there is a loop, the initiating member will detect it, either because it receives the Root Path Trace query it initiated, or because it receives a Root Path Trace reply with a loop in the root path.

If the member so detects a loop, it should immediately drop its parent, and attach to a new parent as described in Section 8.1.

8.10 Neighbor Reachability Detection

In order to maintain an intact tree, members must of course detect, in a timely manner, when their neighbors have become unreachable. What constitutes timely depends on the application. In particular, different detection algorithms are used depending on whether detection can take place at the time an application frame is transmitted, or whether detection must take place before an application frame is delivered.

The former case includes non-realtime applications, where the maximum acceptable application frame latency is substantially greater than the time it takes to establish a new neighbor. It also includes realtime applications where some seconds of frame loss is acceptable. The latter case includes other realtime applications. These two cases are described separately in the following two sections.

8.10.1 Application-Frame Time Detection

Neighbor reachability detection is generally easier and more efficient if tree repair can take place at the time that application frames are transmitted. The basic strategy is simple. When a frame arrives at a member to be delivered and is transmitted to a neighbor, the acknowledgement of the frame serves as the indication that the neighbor is still up. Likewise, lack of an acknowledgement is an indicator that the neighbor is down.

If the neighbor is discovered to be down, the action taken by the member depends on whether the neighbor is a parent or a child. If it is a parent, the member obtains a new parent as described in Section 8.1, and then transmits the frame to the new parent.

If it is a child, the member must inform the children of the child that their parent has become unreachable. In general, a member is not expected to know the children of its children. Therefore, it broadcasts a Member Down Message (Section 5.11) over the mesh. The Member ID Record contains the Member ID of the (apparently down) child.

If the child was a head (as indicated by the Head bit of the Flags field of the YTMP fixed header), the member should reserve the new neighbor capacity freed up by the loss of the child for the soon-to-be-elected head for the same cluster. It should be held for roughly 4 minutes, to allow adequate time for the new head to be elected. The reservation is held in the name of the former head. The replacement head will identify the former head in its Join message.

Upon receiving the Member Down message, each member checks the Member ID Record to determine if the member is its parent (Section 9.2.1 describes what to do in the case of a foot). If it is, the member tries to contact its parent by sending a Get Member Information Message over the existing connection. If it is determined that the parent is down, the member obtains a new parent as described in Section 8.1. If it is determined that the parent is up, the member does nothing.

Note that we intentionally do not attach the original application frame to the broadcast message. While in some cases doing so would save some overhead, it is generally awkward to do it, for a couple of reasons. First, if the source is identified in the frame using the Source Code, a YDP Frame Source Option would have to be attached to inform members of the source. Second, it generally complicates the implementation to mix application and control information in the same frame.

8.10.2 Pre-Application-Frame Time Detection

The only way to detect that a neighbor has become unreachable before application frames are sent is to periodically ping the neighbor (using Get Member Information queries). This is a classic approach, and has the basic trade-off that shorter detection latencies require more frequent pings.

Acknowledgements to application frames (or, if datagram, rate or congestion control feedback) can serve as an indication that the neighbor is up. During the times that no application frames are being sent, pings must be sent at some frequency. Different applications will have different needs for ping frequency.

One reasonable approach, however, would be to ping less and less frequently the longer that applications frames are not sent.

9 Cluster Algorithms

This section describes the various algorithms for operation in a cluster, including:

- how to forward frames over a cluster,
- how a member determines if it is attached to a cluster,
- whether it must become the head member for the cluster,
- how to manage parent selection within the context of a cluster,
- how to manage mesh links within the context of a cluster.

9.1 Cluster Discovery

Every member, whether or not it already has a (non-cluster) parent, continuously monitors to see if a cluster has formed (or disappeared) on each of its shared media. If a cluster exists, the member further participates in an algorithm to determine its status (head or foot) in the cluster.

There are two kinds of clusters, *static* and *dynamic*. Dynamic clusters are formed over a single shared-media (the cluster membership do not span any routers), and the head of the cluster is dynamically selected. Static clusters have an administratively assigned head, which may be off the shared media, but should not be far away (ideally no more than a single router hop).

Only two members are required to form a cluster. If there are three or more members, IP multicast is used for forwarding frames. Otherwise, IP unicast is used. Either way, IP multicast is used to run the configuration algorithm.

Specifically, members:

- join (and therefore listen to) the IP multicast groups (for each seperate interface) for the clusters.
- periodically transmit a Cluster Announce Message over the IP groups.

Each member joins an IP multicast group for each host interface. The IP group addresses for the groups, as well as the UDP port numbers, are pseudo-randomly selected from a range of values. The following table defines the ranges:

Note that these group addresses come from the global scope addresses assigned to IPv4. This is intentional. The scope of cluster discovery is limited by the IP time-to-live field. Therefore, the administrative scoping of these groups is, strictly speaking, unecessary. All other things being equal, it would still be preferable to use the administratively scoped group addresses (link-local or organization-local). There are two problems with this.

First, the range of available addresses is smaller for the administrative scopes, particularly organization-local, which contains only roughly 16,000 addresses. Second, there is likely to be more other groups using these addresses, in the local environment, than the global ones. Both of these problems increase the probability of collisions (multiple groups using the same group address and port).

Note also that multicast protocols other than IPv4 may be used, for instance on networks that run a proprietary multicast protocol. For these cases, the group address selection has not yet been specified.

The algorithm for selecting the value for each range is given in Section 9.11.

In the algorithm for configuring the cluster, there are two major states: electing, and elected (expressed by the Election State bit in the Flags field of the Cluster Announce Record). As long as a head has already been selected and is not known to be unreachable, the cluster is in elected state. Otherwise, it is in electing state, by the end of which a head will have been elected.

In the context of cluster operation, a member can be either a head, a foot, or a *candidate*. It is a candidate if it is participating in an election, and a head or a foot otherwise. Being a candidate is a transient state.

The member operation as a foot, head, and candidate are discussed below. After that are brief sections describing a member's joining and quitting operations.

9.2 Foot

This is steady-state for a foot. In this state, the member knows that it is a foot, knows who its head is, and knows the information needed for transmitting and receiving frames over the cluster.

In this state, the member has two basic jobs, determining if the head is still reachable, and informing the head as to whether 1 or more than 1 foot exists. The former job is part of normal neighbor reachability detection, as described in Section 8.10, though it works slightly differently in the case of clusters.

The latter job is for the purpose of informing the head as to the number of feet: 0, 1, or more than 1. This is so that the head knows whether to stop transmitting application frames to the cluster, or whether to use TCP or some multicast transport protocol.

9.2.1 Head Reachability

For the purpose of determining head reachability, there are two sub-states: pinging and not-pinging. In the pinging state, the foot (and all other feet) is attempting to determine if the head is still reachable. The pinging state can be entered four ways:

1. expiration of a periodic timer,
2. receipt of a Member Down Message with the head listed as the down member,
3. a member's initial attachment to a cluster,

4. receipt of a Cluster Announce Message with Member Status foot and Election State elected, from another foot (this is called a *head ping*).

The first two ways of entering the pinging state correspond to the Pre-Application-Frame Time Detection and Application-Frame Time Detection methods for determining reachability of a neighbor respectively (Section 8.10). The third way is part of the initialization process for a new group member. When a member enters the pinging state in one of these three ways, it first immediately transmits a head ping message. The fourth way is in response to another foot member having already entered the pinging state.

Whether the pinging state is entered because a member transmits a head ping or receives a head ping, the member sets a short head-reply expiration timer. The value of the timer is randomly chosen to be between roughly 3 and 6 seconds.

If after the timer is set another head ping is heard, the head-reply expiration timer is canceled and restarted (thus suppressing the member's own ping in deference to another member's ping). If any frame is received from the head, the head-reply expiration timer is canceled, and the member goes back into the not-pinging state.

In general, anytime a foot receives any frame from the head, it goes to the elected state, and sets a electing-hold-down-timer for 30 seconds (resetting it first if it was already set). Until this timer expires, it must not enter the electing state.

If no frame is received from the head after at least four unanswered head pings are heard over a period spanning at least 24 seconds, the head is assumed unreachable, and the member becomes a candidate.

9.2.2 Foot Announce

In order for a head to decide whether to stop transmitting application frames to the cluster, or whether to use TCP or some multicast transport protocol, it must know whether there are zero (stop transmitting), one (use TCP) or more than one foot (use a multicast transport protocol).

The head should also know if there are transit members, stub members, or both, among the feet. This is for the purpose of forwarding anycast and broadcast frames.

In order for the head to know this, the feet must periodically advertise their existence. They do this at a relatively low frequency, roughly once every half hour.

Specifically, each foot sets a long-foot-count timer to a random value between 20 and 40 minutes. Each foot member cancels and restarts the timer if, before the timer expires, the foot hears Announce Messages from at least two different other feet, both with the same or a superset of listed Multicast Transport Protocols, and, if the foot is a transit member, at least one of which is also a transit member. When the timer expires, it transmits an Announce Message and restarts the long-foot-count timer.

If a foot receives a Cluster Announce Message from another foot with the Quit Status set to 1, the foot sets a short-foot-count timer, at a random value between 5 and 10 seconds. It also sets this timer any time a new member becomes the head.

If before the short-foot-count timer expires the foot hears Announce Messages from the set of feet described above, it cancels the short-foot-count timer but does not restart it. This timer is operated independently of the long-foot-count timer. The purpose of the short-foot-count timer is for the head to learn quickly when the number of feet of each type has shrunk, particularly to zero.

Transit feet should also remember the number of responding feet, whether they are stub or transit, and the multicast transport protocols common to all cluster members, in case they are elected head.

9.3 Become Head

A member becomes a head when it has either "won" an election (dynamic clusters) or is administratively assigned to be a head (static clusters).

9.3.1 Selecting Cluster Values

When a member becomes a head, it must determine the value of the IP TTL and MTP Type fields, and the value of the ID Code field of YIDP.

If the values of these fields were already established before the member became a head (that is, there was a previous head), they remain the same values. This way, the current instantiation of the multicast transport protocol does not need to be restarted. Otherwise, the values are selected as follows.

If the cluster is dynamic, then the IP TTL must be 1. If static, then the IP TTL will have been configured.

The MTP Type is selected among those that have been advertised by all known feet. If there are multiple such protocols, then the selection is made according to some locally defined policy.

The selection of the ID Code field is described in the YIDP protocol specification.

9.3.2 Selecting a Parent

The head must of course find a parent of its own. If there was a previous head, then the newly elected head will attempt to join the same parent. How this is done depends on the circumstances under which the new head is replacing the old.

If the old head has simply become unreachable, then the new head tries to join the same parent using the Dependent Emergency joining procedure (Section 8.1.3). If this fails, the new head goes through the normal process of searching for and joining an appropriate parent.

If on the other hand the old head is gracefully quitting, then a Replacement Join is used. This procedure is described in Section 9.7.

Failing this, it goes through the normal process of searching for and joining an appropriate parent.

9.3.3 Obtaining Other Children

If the old head had (non-cluster) children of its own, the the new head should, to the extent possible, accept these children. It may not be able to accept all or even any of them, depending on its own neighbor capacity.

The new head, immediately upon establishing its own parent, should send uninitiated Get Member Information replies to the children. It should only do this, however, if it is a transit member and has capacity for children. These messages should contain the root path and new neighbor capacity of the new head. While they may have already joined other parents (for instance in the case of the old parent becoming unreachable), this will at least inform them of an alternative choice.

If the old head had become unreachable, and this was discovered via some mechanism other than the receipt of a broadcast Member Down Message, the new head should send Member Down Messages, with the old head named as the down member, unicast, datagram, to the children. These should be sent whether or not the new head has sent the above-mentioned Get Member Information replies.

9.3.4 Head Announce

The head responds to all Cluster Announce Messages heard from feet by immediately transmitting a Cluster Announce Message, with the exception that it should never transmit an identical Cluster Announce Message within 10 seconds of a previously transmitted message. Anytime the contents of a head's Cluster Announce Message changes, however, the head immediately transmits another.

If the head H1 hears a Cluster Announce Message from another member H2 claiming to be a head, and the head H1 is dynamically elected, then the head H1 immediately becomes a foot and initiates a head ping. If H1 is administratively assigned and H2 is dynamically elected, then H1 remains a head.

If both are administratively assigned, then local policy determines which becomes a head. Absent any local policy, the member with the alpha-numerically higher Member name becomes the head.

9.4 Head Election

The goal of a head election is to elect one and only one head for the cluster. An attempt is made to elect the most qualified member, defined as follows:

- A transit member is assumed to be more qualified than a stub member.
- Among transit members, the member with the greatest Raw Neighbor Capacity (as advertised in the Cluster Announce Message) is the most qualified.
- For members with equal Raw Neighbor Capacity, the most qualified is defined as that with the longest Member Age (also as advertised in the Cluster Announce Message).
- Where Member Ages are identical, the most qualified is arbitrarily defined as that with the alpha-numerically highest Member name, and in the case of identical member names, the highest port number.

A foot becomes a candidate whenever:

1. it determines that the head has become unreachable,
2. it receives a Cluster Announce Message from another candidate, and it is not in the electing hold-down period, or
3. it receives a Cluster Announce Message from the head with the Quit or Anticipated Quit Status bits set.

Upon becoming a candidate, the foot must not change its advertised Member Age, even if the actual member age increments while it is a candidate. It must also not change its advertised Raw Neighbor Capacity, even if its actual raw neighbor capacity is changed while it is a candidate. This is to prevent the member's qualifications from changing during the course of an election.

If the foot becomes a candidate via the first means, then it immediately transmits a Cluster Announce Message. Independent of how it became a candidate, it sets its short-electing-timer to a random value between roughly 3 to 6 seconds.

If a candidate hears a Cluster Announce Message from a head, it returns to being a foot (and, as already mentioned, (re)starts its electing-hold-down-timer). If a candidate hears a Cluster Announce Message from another candidate, where that other candidate has better qualifications, the candidate restarts its short-electing-timer (without transmitting any message).

If a candidate's short-electing-timer expires, it transmits a Cluster Announce Message and restarts its short-electing-timer. (Note that in this algorithm a candidate may sometimes transmit a Cluster Announce Message even if there is a better qualified candidate correctly transmitting Cluster Announce Messages. This is because of the random skew in the short-electing-timer.)

If a candidate transmits at least 4 Cluster Announce Messages over at least a 24 second period, without receiving any Cluster Announce Messages from better qualified candidates, the candidate becomes the head. It immediately transmits a sequence of four Cluster Announce Messages, spaced randomly between 3 and 6 seconds apart.

If a candidate receives at least 4 Cluster Announce Messages from another candidate C1 over at least a 24 second period, without receiving any Cluster Announce Messages from still better qualified candidate C2, then it expects candidate C1 to become the head. It sets an expecting-head-timer for a random time between 40 and 80 seconds. During this time, it stays a candidate (that is, periodically transmitting Cluster Announce Messages).

If, after the expecting-head-timer expires, no member has declared itself to be the head, the expected head C1 is considered to be disqualified from the election, and the candidate continues the election, but ignores any received Cluster Announce Messages from the disqualified candidate.

Note that this algorithm requires fully symmetric connectivity between participants. In other words, for all members A and B, if member A can receive the messages of member B, then member B must also be able to receive the messages of member A.

9.5 Joining a Cluster

When a member first starts participating in the cluster algorithm, it assumes that it is a foot. It immediately initiates head reachability detection on that cluster (see Section 9.2). If there is another head established on that cluster, this will cause that head to transmit a Cluster Announce Message, thus allowing the new member to quickly learn the relevent cluster information (IP TTL, ID Code, and MTP Type), and start receiving frames.

Otherwise, when the member finds that there is no reachable head, it will become a candidate, and from there either become the head, or lose the election to another member that joined the cluster at roughly the same time.

If a member is the sole member of a cluster, it considers itself clusterless, but continues to listen to the cluster for other members' Cluster Announce Messages. It also periodically transmits a Cluster Announce Message over the cluster, at a period of roughly 1 hour, just in case another member on the cluster considers itself to be clusterless as well.

9.6 Foot Quit

When a foot quits a cluster, it transmits a Cluster Announce Messages to the cluster, with the Quit Status bit set. This triggers Cluster Announce Messages from other feet, allowing the head to determine if the number of feet has shrunk to one or zero.

It also triggers a Cluster Announce Message from the head. If the quitting foot held pairwise knowledge, then the Cluster Announce Message from the head should no longer include the pairwise knowledge in its list. If it does, the quitting foot periodically transmits its Cluster Announce Message until it sees that it has been removed from the head's pairwise knowledge list.

9.7 Head Quit

When a head quits a cluster, it should engineer a smooth transition from itself to the new head. The basic steps for this are as follows:

1. Announce its anticipated quit to the cluster, causing the election of a new head.
2. The new head becomes the child of the quitting head's parent.
3. The new head takes over the job of forwarding application frames between the cluster and the rest of the tree.
4. The parent quits the old head. If the quitting head has other (non-cluster) children, it starts receiving application frames from the cluster (as a foot), and forwards these on to its children.
5. The new head sends uninitiated Get Member Information replies to the children (of the quitting head).
6. The quitting head sends Anticipate Quit messages to the children.
7. When all the children have finally quit the quitting head, it can itself leave the tree.

Note that the whole process would be simpler if the quitting head first quit its own children, and then quit the cluster/tree. It is, however, better to allow the children to join the new head, rather than find some other parent, for two reasons. First, given that it is in the same internet "location" as the quitting head, it will be an equally good parent (to the extent that it has capacity). Second, if the tree is running at or nearly at capacity, this will make it easier to allow the children to find any parent at all.

The above steps are now described in more detail. First, the quitting head transmits a Cluster Announce Messages to the cluster, with the Anticipated Quit Status bit set. This triggers an election.

When a Cluster Announce Message with the Head Status set from another cluster member is received, indicating that the election has successfully completed, the quitting head sends a Quit Message to its parent, listing the newly elected head as its replacement. As soon as this Quit Message is acknowledged, the quitting head transmits a Cluster Announce Message with the Quit Status bit set.

This will trigger the newly elected head to Join the parent using the Replacement form of the Join. This in turn causes the parent to replace the quitting head with the new head as its child. In so doing, it sends a Quit Message

to the quitting head. Until this final Quit Message is received, however, the quitting head should continue to forward application frames between the cluster and the parent.

If the quitting head has other (non-cluster) children, then, immediately upon receiving the Quit Message from its (now former) parent, it should become a foot, and continue receiving frames from the cluster. It behaves as a foot in all respects (including sending the appropriate Cluster Announce Messages) except one. That is, it continues to forward application frames to its children.

It must, however, immediately send Anticipated Quit messages to its children, and go through the process of a parent quitting its children as described in Section 8.8. Once all of its children have quit it, the quitting head (now a foot) finally fully quits the cluster, though this time as a foot as described at the beginning of this section.

9.8 Echoing YTMP Messages to the Cluster

Certain YTMP messages sent and received by cluster members should be echoed to the cluster. In other words, they should be transmitted, as is, over the cluster IP group, using stream transmission. These messages include:

- All Join Message replies with one of the Accept Subtypes received by the head.
- All Quit Messages received by the head.
- All Get Member Information replies received in response to anycast queries transmitted by the head and directed to parent-side members (as described in Section 8.7).
- All Pairwise Knowledge replies received from other members with a Subtype of Create Mesh.

The first three are transmitted by the head, and contain information useful to a future head. These should be received by all feet, including stub members (which may be elected head in the absense of any transit cluster members). The first two are for the purpose of learning the non-cluster children of the head, which is used as described in Sections 9.3. The third allows a newly elected head to know of parent-side members This is useful for quickly responding to the loss of its own parent, and for finding a better parent.

The last is used by the head member to maintain its cluster pairwise knowledge list, and by feet to trigger pairwise knowledge creating and deletion. These messages should be received by all cluster members except stub members, which do not maintain pairwise knowledge. The use of these messages is described in Section 10.2.

9.9 Forwarding Frames over a Cluster

This section describes how to forward frames of the various delivery modes (multicast, unicast, broadcast, and the two anycasts) to and from a cluster.

There are two ways a cluster member can view the feet: as a single member (the cluster child), or as multiple members (the feet). Both views can be taken at different times, depending on the delivery mode of the frame. In the multicast, broadcast, and anycast modes, the Dest Code field of the frame header indicates whether the frame is to go to transit members only, or both transit and stub. If the cluster child contains a single transit foot, it is considered to be transit, even though the individual feet cannot themselves have child neighbors. Otherwise it is considered to be stub, even though strictly speaking the head does forward frames.

In what follows, transmitting "to a cluster" or receiving "from a cluster" is taken to mean transmitting to the IP multicast group of the cluster or receiving from the multicast group of the cluster. Frames transmitted to the cluster always contain the source address of the transmitting member. Frames received from the cluster with a member's own IP address as the source address are ignored.

9.9.1 Multicast Delivery Mode

A multicast frame received from the cluster is never transmitted to the cluster. A frame originating at a foot is transmitted to the cluster. The head receives it and forwards it to its other neighbors (even if the head is, strictly speaking, a stub member).

A head, receiving a multicast frame from anywhere other than the cluster, forwards it to the cluster unless the Dest Code is transit only, and the cluster is stub.

9.9.2 Unicast Delivery Mode

In the case of datagram, a unicast frame is always delivered IP unicast directly to the destination, regardless of the destination's cluster status.

In the case of stream, the reverse path to the destination must be known. When a cluster member transmits a multicast frame with a reverse-path enabled YDP Frame Source Option, all other transit cluster members, and in the case of a stub cluster, all cluster members, must record it as being attached to the cluster. In the case of feet, however, this is done only in preparation for potentially becoming a head later on.

Unicast frames originating at a foot are transmitted to the cluster, whether or not the destination is on the cluster. If the destination is on the cluster, it will receive the frame from the cluster. Other feet ignore all unicast frames received from the cluster and not destined for them.

When the head receives a unicast frame from the cluster, and the destination is on the cluster, the frame is ignored. If the destination is not on the cluster, the head forwards the frame according to the reverse path. When the head receives a unicast frame from other than the cluster, and the destination is on the cluster, the frame is forwarded to the cluster.

9.9.3 Broadcast Delivery Mode

All cluster members consider the cluster to be a neighbor for the purpose of forwarding broadcast frames. If a cluster is stub, a broadcast frame with a Dest Code of transit only, originating at a foot, is forwarded to the cluster, but is ignored by all members except the head. The parent, if it is a stub, treats the frame as though it originated at itself, and forwards it to its parent.

A head, upon receiving a transit-only broadcast frame from a non-cluster source, does not transmit it to the cluster if the cluster is stub.

9.9.4 Tree and Mesh Anycast Delivery Modes

For the purpose of selecting the next hop for an anycast frame (that is, randomly choosing among the possible next hops), a head considers the cluster child to be a single neighbor. A foot considers the head to be a single neighbor, but does not consider the (remainder of) the cluster child to be a member.

If a foot selects the head as the next hop, then the frame is transmitted directly to the head (using IP unicast). If the frame is stream, then a TCP connection must be established first.

If a head selects the cluster child as the next hop, then it must determine which of the individual feet should receive the frame. To do this, the parent simply selects randomly among the feet that it happens to know about, based on various frames recently transmitted by feet (recognizing that this is not true random distribution among the feet).

If the frame is mesh anycast, and the "hop count" value in the upper 8 bits of the Dest Code field is 2 or greater (after it has already been decremented), and the member selected by the by the head is a foot without a mesh link, then the head can simply decrement the hop count by 1, and transmit it back to itself (or, equivalently, decrement the hop count by 2, and reselect the next hop). The reason for this is that such a foot would have no option but to decrement the hop count, and transmit the frame back to the head.

9.10 Multiple Clusters

If the host where a member resides has multiple interfaces, the member runs the cluster configuration algorithm separately over all of the interfaces. The member may continue to run the cluster configuration algorithm over any interface for which no other cluster members exist. As long as one or more clusters have one or more other members, however, the member must select a single interface over which to participate in the cluster.

Type	First	Last	Size
Port	32768	65535	15 bits
IPv4 Group Address	232.0.0.0	235.255.255.255	26 bits

Table 1: Ranges of Port and Group Address Values for Cluster Configuration

While it is free to listen to the IP groups of the clusters where it is not participating, it may not tranmit to those groups.

(Note that while, strictly speaking, a member could be a head for multiple clusters, or a head for some clusters and a foot on another, allowing this opens the possibility of a tree loop among the heads and feet of clusters. The simplest way to eliminate this possibility is to limit cluster participation to a single cluster only.)

9.11 Generating Pseudo-random Port and Address Values

This section describes how to generate the pseudo-random port and group address values from the ranges given in Table 1. The low and high values of each range are given by the columns First and Last in Table 1. Size contains the number of values in the range, followed by the number of bits required to express the size.

The method for pseudo-randomly selecting the value is to generate a pseudo-random number of the appropriate size (number of bits N), and then add that number to the first value in the range.

The Group ID is used as the key for generating the pseudo-random number. This way, all members generate the same values for a given group. The Group ID consists of the Group Name (the Rendezvous Host DNS name), the Group Subname, and the Group UDP port number. Specifically, the key is the string of bytes consisting of the Group Name, the Group Subname, and the port number concatinated together.

The Name and Subname must be all lower case (ASCII). Neither the Name or the Subname have a trailing zero byte—the first byte of the Subname immediately follows the last byte of the Name. The low order byte of the key is the leftmost byte of the Name. Likewise, the first byte of the Subname in the key is the leftmost byte of the Subname. The first byte of the port number in the key is the low order byte of the port number.

For instance, the key for a Group ID with Name “FOO.com”, Subname “Bar”, and Port 1234 is (in hex):

```
low order                    high order
66 6f 6f 2e 63 6f 6d 62 61 72 04 d2
f  o  o  .  c  o  m  b  a  r  [1234]
```

The MD4 message-digest algorithm is used to generate a 128-bit digest from the key. This digest is then “folded” into the desired range of values as follows:

1. Initialize a variable, called output, of the appropriate size (N bits), to all zeros.
2. Generate a 128-bit digest from the key using MD4.
3. Using the low-order N bits of the digest, do an exclusive OR of the digest with the output, putting the result in output.
4. Shift right the digest by N bits, so that the low-order N bits disappear, and the high-order bits are replaced by 0’s.
5. If the digest is non-zero, repeat from step 1.
6. If the digest is zero, quit. Output contains the pseudo-random number to be added to the first value of the range.

10 Mesh Maintenance Algorithms

The primary purpose of the mesh topology is to provide a means of frame dissemination that is highly reliable—that is, reaches all group members with high probability. This requires a mesh topology that is connected with very high

probability, which in turn requires that all members maintain several mesh links with other members.

The methods for maintaining the mesh topology differ depending on whether a member is a cluster cluster member (including the cluster head) or not. The first section below describes the case where the member is not a cluster member. The subsequent section describes the case where the member is a cluster member.

10.1 Non-Cluster Member Mesh Maintenance

For the purpose of maintaining a mesh topology that is connected (not partitioned) with extremely high probability, each transit member, including the root, must maintain a list, called the mesh member list, of randomly selected members in the group. Broadcast or mesh anycast frames are sent to members from this list as well as to neighbors.

The means of finding members for the mesh list are similar to that of finding members for the parent-side list in that an anycast message is used to discover another member. The differences are that:

1. the recipients of the anycast are not limited to parent-side members,
2. list entries are pairwise. That is, if member A has member B in its list, then member B also has member A in its list, and
3. the Pairwise Knowledge Message is used rather than the Get Member Information Message.

The number of members in the list is a trade-off between insuring a connected mesh and minimizing the overhead in keeping the list up-to-date. Off-hand, 4 or 5 entries is probably sufficient, but this should be studied.

The reason list entries should be pairwise is because the mesh should be a connected topology with high probability. If the entries are not pairwise, then there is a reasonable probability that a given member is unknown to all other members, even if it knows about a number of other members, The mesh would therefore not be connected. If entries are pairwise, a given member can insure not only that it knows about other members, but that other members know about it.

The algorithm for keeping pairwise entries is a lazy one. The basic "goal" of each member is to keep a certain number of entries in its mesh list. The entries in the list each have a timeout associated with them. When an entry is timed out, or deleted through the reception of a destroy Link Mesh message, the member should sooner or later send another Link Mesh create query to establish another entry.

A list can easily exceed its target number of entries, because it cannot control the number of link mesh queries it receives. In general, this is ok. If, however, the list gets way too large, the member can choose to simply ignore received queries. When the sender of the query does not receive a reply after some period of time, it will generate another anycast query, which will probably be delivered to a different member.

10.2 Cluster Member Mesh Maintenance

For a cluster member, the simplest way to maintain the mesh topology would be for it to operate exactly as a non-cluster member does—that is, maintain its own mesh links independently of the other cluster members. This is problematic in the context of a cluster, for scaling reasons. Imagine, for instance, a cluster with 20 members, each of which maintains 4 mesh links (that is, 4 pairwise knowledge entries). Any given broadcast frame would be received or transmitted by all 20 members 4 times, for a total of 80 frames, and their acknowledgements, crossing the same shared media (all with the same content).

The next simplest thing to do would be to have only the head maintain pairwise knowledge. The head could additionally inform the feet of the pairwise knowledge. The problem here, of course, is that if the head crashes, broadcast messages, such as a Member Down message sent in response to the crashed head, will not reach the feet.

Pairwise knowledge is therefore distributed among transit cluster members, but in such a way that the total amount of pairwise knowledge does not exceed that normally held by a single non-cluster member. The algorithm for this goes as follows.

Every transit cluster member maintains a list of pairwise knowledge *that they themselves setup*, either by initiating or responding to a Pairwise Knowledge query. This is the usual list already described in Section 10, and is here called the member pairwise knowledge list.

In addition, every transit cluster member maintains a list of cluster pairwise knowledge entries, here called the cluster pairwise knowledge list. This is pairwise knowledge cumulatively held by all members in the cluster, but only that established by a cluster member (that is, the cluster member initiated the pairwise knowledge by transmitting the first Pairwise Knowledge query). The entries in this list identify both ends of the pairwise knowledge, as well as the other information normally associated with a pairwise knowledge entry (for instance, the expiration time).

Cluster members obtain the cluster pairwise knowledge list entries from two sources:

1. Individual members echo, to the cluster, all unicast Pairwise Knowledge replies the received that resulted in adding or refreshing entries in the member pairwise knowledge list.
2. The head lists all entries in its Cluster Announce messages.

The former method is the initial means of learning the entries. The latter allows new cluster members to quickly learn the list.

The cluster pairwise knowledge list must have at least three and no more than four entries. If the list has only three entries, and one or more transit cluster member hold none of them, then they make an attempt to establish the fourth. If, on the other hand, the list has four entries, and one or more cluster members holds two or more of them, then those members will make and attempt to destroy one of their entries. In this way, there will generally be three or four entries, and they will be distributed across as many cluster members as possible.

Specifically, whenever the head transmits a Cluster Announce Message (which contains a list of the pairwise knowledge entries), each transit cluster member starts/continues or cancels a timer according to the following table:

Number of Member Entries	**Number of Cluster List Entries**			
	< 3	3	4	> 4
0	Start Short, Create	Start Short Create	Cancel	Start Long, Destroy
1	Start Long Create	Cancel	Cancel	Start Long, Destroy
> 1	Start Long Create	Cancel	Start Short, Destroy	Start Short, Destroy

Start Short means to start a timer (or allow an already started timer to continue) for a randomly set time from 0 to 5 minutes. Start Long means to start a timer (or allow an already started timer to continue) for a randomly set time from 5 to 10 minutes. Create means to create a pairwise knowledge entry if the timer expires, as described in Section 10. Destroy means to destroy a pairwise knowledge entry if the timer expires.

The effect of the timer is that typically one member (though occasionally more than one) will accomplish the task (create or destroy) and echo the result to the channel, causing the others to cancel their timers. Through having long and short timers, the members with fewer (greater) entries are preferencially chosen to create (destroy) entries.

The echo of course also serves to inform the head of the added or deleted pairwise knowledge. This causes the head to update its list and transmit another Cluster Announce message. Anytime a foot receives a Cluster Announce message from the head that either lists pairwise knowledge that it does not hold, or does not list pairwise knowledge that it does hold, it should repeat the appropriate Pairwise Knowledge Message.

All transit cluster members are free to accept requests from other members for pairwise knowledge at their discretion.

11 Open Issues

11.1 Full Tree

If a tree is full (all members have no capacity for additional stub members), such that a new stub member cannot join, we should probably have some kind of message that can be sent to the rendezvous indicating as much. This could in turn be forwarded to a human, who might be able to do something about it.

11.2 Dependence on Rendezvous

One of the significant advantages of IP multicast (at least in its DVMRP form) over YTMP is that the former is more robust, specifically because it does not depend on any special purpose members such as rendezvous.

One way to avoid this weakness in YTMP is to use IP multicast itself in lieu of rendezvous for tree discovery. Exactly how this would work is not clear, but it is clear that one wouldn't want one IP multicast group per yoid group. This would defeat one of the purposes of having yoid—that is, to overcome the "per-group" scaling difficulties of IP multicast routers.

Instead, one IP multicast group could be shared among many yoid groups. The IP multicast group would only be used as a backup, in case the rendezvous for any given group were unavailable. The basic idea is that members would join the IP multicast group when they joined yoid group. Newcomers would do an expanding ring search on the IP multicast group to find existing group members.

For this to work it would have to be self-configuring like the rest of YTMP. This means that the IP multicast address used would have to be self-selected, perhaps based on a hash of the group name. Detection of IP multicast capability at the host should be automatic.

11.3 Discovery of Infrastructure Members

Need to discuss the case where a member should try to discover and attach to an infrastructure member rather than that of another application member.

11.4 Security

Clearly security functions need to be added as soon as possible. I can imagine a wide range of security functionality, from a simple access list at the rendezvous, to full knowledge of the security information of all members by all other members. It seems likely, in any event, that the rendezvous will be the focal point for all security functions.

11.5 Group With Identical Group IDs

It is of course possible for two groups with the same Group ID to be created at different times. Given, however, that a group can continue to exist without the rendezvous, it is possible for a new group to be created while a former group with the same Group ID still exists. Once this happens, without safeguards put in, the old and new groups are very likely to merge (because of the root partition discovery functionality).

Whether this is a feature or a bug depends on the situation, but generally speaking it would have to be considered a bug. For now, we do nothing about it. When security features are added, this will certainly become detectable and fixable (for instance, because each group will have an encryption key associated with it).

11.6 Determining Spare Capacity

Not enough is said in this spec about how a member goes about determining spare capacity for itself.

Towards that end, each member must have a good idea of what its basic raw capacity is (outgoing and incoming BW). It should probably base this on the levels of performance it is able to achieve with minimal packet loss.

It also must know the flow required for the given group, and the distribution of senders (to know whether a new neighbor is likely to contribute to incoming flow as well as outgoing). By taking the difference between capacity and group flow, the member should be able to make a reasonable estimate of what its spare capacity is (normalized by the group flow).

11.7 Excess "Background" Activity

Background activity is that activity used to improve the tree, or maintain its correctness. Generally speaking, members should limit the number of frames sent in support of background activity to a fraction of the number of applications

frames (5% - 10%). This means that trees with very little traffic will remain highly non-optimal. This is ok, though, because the cost of optimizing such a tree would generally be more than the savings.

11.8 Infrastructure-based Members

This specification only describes the algorithms for configuring groups where the members and the applications are co-resident in the same host. It is also desirable to have the option of *infrastructure-based members*. These are members that are installed in the infrastructure for the purpose as acting as transits for groups for which no application is co-resident.

For this to work, member members must 1) know that such infrastructure members exist, 2) know that they can be used for a given group, and 3) must join the infrastructure member, and get the infrastructure member to join the group. The first can be accomplished through the use of well-known DNS names (something like, yoid.in.member.domain). The second will require additional information about the group be set by the application creating the group, and transmitted by the rendezvous to newcomers. The third will require an addition message type.

None of this appears particularly hard, it just has to be specified.