Yoid: Extending the Internet Multicast Architecture

Paul Francis

ACIRI
francis@aciri.org, www.aciri.org

April 2, 2000

Contents

0.1 Abstract

If we take a broad view of the term "multicast" to mean any distribution of content to more than one machine, we find that multicast is proceeding along two distinct architectural tracks. On one track is IP multicast, which mainly targets realtime non-reliable applications, but for which hopes run high for reliable applications as well. On the other are a plethora of open or proprietary host- or server-based approaches, each typically targeting a specific application or product line.

IP multicast suffers from a number of technical problems, lacks applications, and in general is having trouble reaching critical mass, especially regarding anything resembling a global infrastructure. Server-based approaches are valuable and wide-spread, but there is no synergy in terms of multiple distinct groups working within the same architecture. As a result, progress is not as fast as it could be, and consumers are strapped with multiple application-specific infrastructures to deal with.

This paper presents an architecture, called yoid, that aims to unify both tracks under a single umbrella architecture. Yoid attempts to take the best from both tracks—reliable and asynchronous distribution from the server-based track, and dynamic auto-configuration via a simple API from the IP multicast track.

A key component of yoid is that it allows a group of endhosts (the hosts where the content-consuming application resides) to auto-configure themselves into a tunneled topology for the purpose of content distribution. Yoid can run over IP multicast, but does not require it. This allows application developers to bundle yoid into their applications, giving their applications robust and scalable configuration-free out-of-the-box multicast. This is key to the initial acceptance of yoid and for allowing it to reach critical mass.

Yoid is not limited, however, to endhost-based distribution. It can also work in infrastructure servers (boxes that receive, replicate, and forward content but that are not consumers of the content). This allows improved performance for applications that require it. It can also provide other benefits such as better security. The endhost- and server-based modes of operation taken together, along with yoid's ability to utilize local islands of IP multicast, allow yoid to support the broadest possible range of applications.

1 Changes

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

2 Introduction

2.1 Document Roadmap

Lest the reader be immediately put off by the size of this document, I offer the following brief roadmap:

If, having read the abstract, you want to go straight to a very brief technical overview of yoid, look at Subsection 2.4 ("Yoid in a Nutshell") and perhaps the subsection preceding it (2.3). For a deeper technical overview, see Subsections 3.1 and 3.2.

Otherwise, this paper pretty much proceeds in increasing detail, and you can just start at the beginning and continue until you've read enough. The first Section (2, "Introduction"), tells you why we need better distribution, what yoid is, and why yoid provides better distribution.

The second Section (3) takes you through yoid in increasing technical detail. Of particular interest might be Subsection 3.4 ("Yoid Tree Management Protocol (YTMP)"), and to some extent the following subsection, which describe how yoid tunneled topologies are dynamically configured—the only really novel thing about yoid.

The last section outlines an assortment of enhancements that may be made to the basic architecture to increase yoid's capabilities or solve various problems endemic to yoid. This section can easily be skipped by all but the most interested readers.

This document is in an early draft stage. It is still missing references, and has not gone through any peer review. Any references or comments on the document or yoid itself are greatly appreciated.

2.2 Motivation

Let's take a broad view of the term "multicast" and take it to mean every instance where content is moved from one machine to more than one other machine. For lack of a better term, and to avoid long and clumsy phrases, let's refer to this broad view multicast as simply *distribution*.

Viewed this way, the majority of what gets transmitted over the internet is distribution: mail, news, web pages (HTML) and files of all types (jpg, mp3, etc.), chat, channels, DNS records, audio-video broadcasts, and so on. While strictly 2-party exchanges are obviously important, it is not an exaggeration to say that the internet is what it is because of its distribution functionality.

In spite of this, virtually every distribution application in the internet today runs over the unicast infrastructure and derives its distribution functionality from mechanisms internal to and specific to the application itself. For instance, RFC822 mail headers have mechanisms for detecting loops among mail forwarders, NNTP has the NEWNEWS command to manage flooding of news articles, HTTP has redirection and mechanisms for handling caches, and so on. Practically speaking, though, there is no general "infrastructure" support for distribution in existence today. The exception that proves this rule is IP multicast, globally available tunneled in the form of the mbone, and privately available as either tunneled or native IP multicast.

The reason this exception proves the rule is that IP multicast has so far not lived up to early expectations, to say the least. And this in spite of the fact that it is now available on most host operating systems and in most routers (though it is usually not "turned on"). Different people have different ideas on why IP multicast has not taken off, ranging from a lack of tools for managing IP multicast installations to insufficient protocol development to a lack of IP multicast-ready applications (all examples of reasons voiced by participants in the IETF maddogs ad hoc group). Many people, myself included, have labored under the tacit assumption that if we could only fix this or that problem, and add this or that functionality, then IP multicast would reach some sort of critical mass and take off, in the same sense that HTTP/HTML at some point reached a critical mass and took off. It would then serve as the infrastructure foundation upon which various kinds of distribution applications would be built.

I no longer believe this is a realistic possibility. IP multicast, in its role as "the architectural foundation for internet distribution", suffers from one fundamental problem, one major problem, and a lot of nagging problems.

The fundamental problem is that IP multicast works only across space, not across time, whereas most distribution on the internet (almost everything mentioned above), works across both space and time. What I mean by this is that the recipients of most types of distribution content (mail, web pages, etc.) want to receive it at different times. Even content that under ideal conditions would reach all recipients immediately (i.e., "push" content like mail) can't because not all recipients are ready to receive all the time (mainly because they are not connected to the internet all the time). IP multicast, on the other hand, requires that all recipients receive the content at the same time. (I understand that there is a way around this, namely multicasting the content multiple times until all recipients have got it. But this is not a very attractive thing to have to do, and rather proves my point.)

The major problem referred to above is that IP multicast addresses are too small. Basing the global architectural foundation for distribution on a 27-bit address space is, frankly, silly. It may very well be that some of the current initiatives for IP multicast address assignment will satisfactorily solve the problem (I did say this was only a major problem, not a fundamental problem), but it really is bending over backwards unnecessarily. And of course there is IPv6, but I wouldn't want to assume that that will become ubiquitous any time soon.

The nagging problems include:

- Large IP multicast routing tables.
- Congestion control over IP multicast.
- Reliable data transport over IP multicast.
- Good access control and security mechanisms.

I want to repeat that all of the above discussion of IP multicast is in its role as "the architectural foundation for internet distribution." To be fair, I understand that its not like some person or group ever sat down and, working from a clean whiteboard, analyzed all the options and decided that IP multicast, with its 27 bit address space and space-only distribution mechanism, was the best choice for all internet distribution. Rather, we've incrementally backed ourselves into this corner via some set of historical decisions, or non-decisions, that have long since been overtaken by events.

IP multicast is ideal for applications that cannot tolerate (much) delay, can tolerate some loss, and have throughput that is relatively high but upper-bounded. This mainly includes interactive things like audio-video conferencing and internet games. A host- or server-based approach works poorly or not at all for these applications. On the other end of the spectrum, a host- or server-based approach is great for applications that can tolerate significant delay (minutes or hours) and cannot tolerate loss. This includes things like mail and file distribution. IP multicast works poorly or not at all for these applications. For stuff in between (chat, distributed whiteboard, audio-video one-way broadcast), both can suffice, and reasonable people could agree to disagree on which is better.

For this stuff in between, however, it just turns out that the path of least resistance for obtaining distribution funtionality is the server-based approach. This may be in part because of the technical problems of IP multicast, or because people don't like mucking with their router infrastructures, which today are mission-critical resources. I think probably the primary reason, though, is that for the server-based approach, the application provider needs to convince the application user to deploy only the application itself. For the IP-multicast based approach, the application provider needs to convince the application user to both deploy the application and deploy IP multicast (or, more often, convince yet somebody else to deploy IP multicast). The latter is a far more difficult proposition. Whatever the reasons, for almost any application where a host- or server-based approach will suffice, even if IP multicast would ultimately be a better approach, the server-based approach will be used.

As a result, we now live in a world where there are dozens of application-specific ad hoc and proprietary server-based systems for doing distribution. In addition to open standards like NNTP, IRC, ICP, RFC822 mail headers, and so on, many companies offer proprietary server-based systems: Pointcast for its channels, Tivoli for its software distribution, Tibco for its publish/subscribe stuff, RealNetworks for its audio-video broadcasts, and on and on. This state of affairs cries out for a standard approach that that can handle all of the above sorts of distribution and more.

2.3 Yoid: An Alternative Architecture

What is really needed is a general architecture for all internet distribution. This architecture should have mechanisms for both space-based and time-based multicast. It should be able to seamlessly incorporate multicast media (LANs, satellite) and IP multicast where they exist and where their use is appropriate. It should have an easy evolutionary path, by which I mainly mean that it shouldn't require changes to existing infrastructure, including host operating systems. It should have a fully scalable naming scheme. And it shouldn't suffer from any of the nagging problems of IP multicast.

This paper presents an architecture that holds promise as the general architecture for all internet distribution. I call it yoid, for reasons that will be clear shortly.

Simply put, yoid is a suite of protocols that *allows* all of the replication and forwarding required for distribution for a given application to be done in the endhosts that are running the application itself. In other words, yoid works in the case where the only replicators/forwarders (distributors) of content are the consumers of the content themselves. Let me be clear. Yoid does not force all distribution to be done by the content consumers—it can also be done by "servers" in the infrastructure. Nor does yoid prevent the use of IP multicast—it is used, but only where it exists and where its use is appropriate. With yoid, neither infrastructure servers nor IP multicast are *necessary* for distribution, but they can be brought to bear where economics or application requirements make them appropriate. When IP multicast is used, it will generally be confined to local networks. Islands of locally scoped IP multicast groups will be connected by global yoid.

2.4 Yoid in a Nutshell

Here is a very brief overview of the key technical components of yoid: The core of yoid is a topology management protocol, called YTMP, that allows a group of hosts to dynamically auto-configure into two topologies:

- a (tunneled) shared tree topology for efficient multicast distribution of application content, and
- a (tunneled) mesh topology, for robust broadcast distribution of various information, including control information and, where appropriate, application content.

The tunnels can be either two-party (using UDP or TCP), or N-party (using very tightly scoped IP multicast). Each host can join or leave the two topologies (jointly called the *tree-mesh*) independently, making the group itself dynamic.

Each group has one or more rendezvous hosts associated with it. Rendezvous hosts are not attached to the tree-mesh—rather, they serve as a discovery or bootstrap mechanism for allowing hosts joining a group to find other hosts already in the tree-mesh. Group members contact a rendezvous host when they first join and when they finally quit the group (where possible). The rendezvous host maintains a list of some or all group members, and informs joining hosts of a number of these. The hosts then participate in the topology management algorithm with these already attached hosts to attach to the group.

The name of a given group consists of three components:

- the name of the rendezvous hosts (if multiple, they have the same name),
- the UDP ports the rendezvous hosts are listening on (optional, if not the default port number).
- the name of the group, which is unique to the rendezvous host.

This naming scheme both 1) allows any host (with a domain name) to locally create a globally unique group name, and 2) allows for discovery of the rendezvous host. The group name can be encoded as a URL, and for all practical purposes is a URL:

```
yoid://rendezvous.host.name:port/group.name
```

This gives you a basic picture of yoid, and is all I say about it for the time being. Later (sections 3 and 3.2 and beyond) I'll introduce all the terms and acronyms and more detail (where the devil lies).

2.5 Yoid Pros and Cons (Mostly Pros)

In what follows, I present the advantages and disadvantages of yoid versus IP multicast in the context of the larger goal of internet distribution. These arguments are clearly not meant to convince, given that they are based on an overly simple description of yoid. Rather, they simply highlight the main features. It will take extensive experimentation to see if the advantages presented here really pan out.

Forwarding Table Size

I first conceived yoid solely in response to the problem of large forwarding tables in IP multicast routers. My thought was that, for many kinds of distribution, especially smallish groups with relatively low volume, there is no particular reason that IP multicast must be used—the group member hosts themselves could manage replication and forwarding just fine. This offloads the job from routers, saving them for higher volume multicast applications where their higher speeds and more optimal topologies are needed.

In the case where yoid forwarding is taking place in endhosts, there is no forwarding table size problem because the endhost has only to carry topology information for the groups it has joined. Since the host is also running an application for each group, the overhead due to maintaining the forwarding information itself can be seen as just an incremental extra cost on the application. (Each yoid host only needs to know about its neighbors in the tree-mesh and a small handful of other hosts in the group, independent of the size of the group.)

In the case where yoid forwarding is taking place in infrastructure server boxes, each server must carry only topology information for those group members among the nearby hosts it serves. Unlike the case with IP multicast, each yoid topology is created from scratch and requires no servers in the "middle" of the topology to connect servers at the "edges". The "edge" servers connect directly with other "edge" servers through tunnels. (In IP multicast the routes are dynamic, but the underlying topology, even in the case of the tunneled mbone, is not.) In other words, all yoid servers require forwarding information on a par with what "local" routers (as opposed to routers in the backbones) must carry.

Name/Address Assignment

Yoid does not have IP multicast's address assignment problems because any host with a domain name can locally create any number of globally unique yoid group names. (Hosts without a domain name will be able to use a meta-rendezvous service provided by hosts that do have a domain name, but this is getting ahead of the story.) In addition, with yoid running "above" IP multicast, address assignment in IP multicast is simplified because IP multicast will generally be limited to local areas where address assignment is much easier.

Of course the cost of this is each yoid group's dependence on one or a small number of rendezvous hosts. Practically speaking, I don't think this is much of a problem. One can replicate the rendezvous host for reliability, and the rendezvous host is neither involved in the topology building algorithm per se nor is attached to the tree-mesh, so its per-group-member overhead is low. Also, there are practical advantages to having a central point of contact—accounting, distribution of security information, access control, and so on.

Evolutionary Path (or, the Chicken-and-Egg Problem)

Yoid does not have the application/infrastructure chicken-and-egg problem of IP multicast. This is the problem where nobody (hardly) wants to write applications for IP multicast because there is no infrastructure, and nobody (hardly) wants to put in the infrastructure because no users have applications that demand it. Because yoid can run purely in the hosts running the application, and in fact can be wholy bundled with each application—no OS changes—the application developer can build a distribution application with little concern about the users' network environments or OS capabilities.

This allows for a natural evolution of a distribution infrastructure. Applications start with pure endhost-based yoid. Naturally applications will push the limits of endhost forwarding performance, creating a demand for infrastructure boxes. These boxes, however, can be installed one at a time on an as-needed basis (in the same way that web caches can be added one at a time), making their deployment relatively easy. The infrastructure boxes boost performance, applications respond by demanding even more, and the ensuing demand/supply arms race results in the desired end-goal—a "distribution infrastructure box" on every wire (even if those boxes turn out to be yoid servers not IP routers).

Personally I think there will always be a good use for pure endhost-based yoid. If we assume a tree with a small fan-out (I generally assume a fan-out of 2), then even a tree consisting only of 56kbps dialup hosts can generate useful throughput. For instance, say we can get 20 kbps throughput (20kbps in and 40kbps out to two neighbors), which seems reasonable. This is enough for a reasonable quality audio presentation with graphics in real time or low-end internet radio.

But even if the end-host only mode of operation serves no other purpose than bootstrapping a distribution infrastructure, it will have been as critical to the infrastructure as a first-stage booster rocket is to a satellite.

Congestion Control

Yoid does not have the congestion control problems that IP multicast has. Because each (tunneled) link in a yoid tree-mesh is a (unicast) TCP connection or RTP stream, yoid uses the normal congestion control used over the internet today.

Having said that, yoid does have the characteristic that potentially very large numbers of hosts could be transmitting large volumes over the internet at roughly the same time. While the intent is for yoid topology tunnels to normally link hosts near each other thereby avoiding multiple tunnels passing through the same routers, this will not be perfect, especially early on before we've gained much experience. As a result, sudden internet-wide bursts of traffic could result in unexpected bad consequences.

End-to-End Reliability

Getting end-to-end reliability out of a host-based infrastructure is far easier than getting it out of IP multicast. Since IP packet loss is recovered by TCP at each hop, a yoid tree has no ack implosion problem. In fact, a key objective of most reliable transport protocols for IP multicast is to build a tree (among routers or among hosts, depending on the scheme) over which acks or naks can be sent. Yoid of course builds a tree, but just happens to use it for sending content as well as sending acks.

This is not to say that getting end-to-end reliability automatically falls out of using TCP hop by hop. Packets can still be lost: because of buffer overflow at hosts, and because of data loss during topology changes. Recovering these loses, though, is relatively straightforward. It requires an end to end sequencing protocol to identify loses, push-back flow control at each hop to avoid buffer overflow, and buffering in each host to recover topology change-induced losses locally.

The Benefits of Buffering: Across-time Distribution

Perhaps the most important advantage of the yoid approach over IP multicast is in its ability to do distribution across time, not just across space. This allows it to accommodate a large number of distribution applications that IP multicast cannot handle and was never meant to handle.

It is easy to visualize how across-time distribution works in the case of a server-based yoid infrastructure. This is no different from the numerous server-based across-time distribution infrastructures we already have (mail forwarder infrastructures, netnews infrastructures, caching infrastructures, Pointcast infrastructures, and so on).

It is much more difficult to visualize how across-time distribution works in the case of endhost-based yoid distribution, especially for hosts that are not administratively related. Such a host would, for instance, join a tree-mesh and start receiving a file. (It is easiest to think of across-time distribution in the context of file distribution, understanding that a file can be anything from a mail message to a software executable, since file distribution has relatively relaxed timing constraints.) At some time, the host is expected to forward the file to one or two other hosts (to garner the benefits of "multicast" distribution). Those other hosts may connect to the sending host long after it has received the file. Therefore, speaking generally, any host may be expected to stay on the tree-mesh long enough to pass the file on to other hosts, even if this means staying on the tree-mesh long after the file has been fully received. It could even mean leaving the tree-mesh when the host disconnects from the internet, and rejoining when the host later reconnects, solely for the purpose of passing the file on.

I believe it can be done, to some extent, but I leave further description for later (see Section 4.5). Note in any event that integrity of the file in this context is not a particular problem. I assume as a matter of course that the distributed files have signatures associated with them, and that these signatures are obtained from the (trusted, more-or-less) rendezvous host at first contact.

Discovery of Nearby Services

One of the oft-cited uses of IP multicast is to discover nearby (in the context of physical topology) hosts containing specific services. The idea here is that the IP multicast topology closely resembles the physical topology, and therefore a host that is nearby in the IP multicast topology is also nearby in the unicast topology. This is certainly true for native IP multicast (where the multicast and unicast topologies are one and the same), but is also generally true for well-managed tunneled IP multicast. The mechanism for this discovery is an expanding ring search whereby the hop count, or possibly administrative scoping, is incrementally expanded until the service is discovered.

Yoid cannot easily substitute for this use of IP multicast. Yoid topologies are dynamically formed and can only come to resemble the physical topology rather slowly. Yoid can certainly be used as a means to discover a service—by having the hosts with the service join a well-known yoid group. Systems trying to discover the service, however, cannot quickly discover which of the hosts with the service is the closest.

2.6 Back to the Architectural Foundation

I'd like to now revisit the earlier comment about IP multicast being inappropriate as the "architectural foundation" for distribution in the internet. IP multicast, in its native (non-tunneled) mode, is fast and uses the wires efficiently. This is good, and is true because it runs in boxes that can (usually) run at wire speeds, and that join multiple wires together (i.e., routers). IP multicast also has no buffering, and has a ridiculously small addressing space. This is bad, and is true because long ago it seemed (and probably was) appropriate to graft multicast functionality onto IP, which is a bufferless thing that today has not enough addresses.

Even if someday we manage to go to IPv6, IP multicast will still be bufferless, and so still inappropriate for most kinds of distribution. A yoid architecture, by which I mean one where 1) the distribution topology auto-configures and can be limited only to boxes that are consumers of the content, 2) addressing is domain-name based as described above, and 3) the links are always unicast IP tunnels, even if the nodes on either end share a wire, can have the advantages of IP multicast (fast boxes attached to and joining together multiple wires) without the limitations.

2.7 Related Work

Dissatisfaction with IP multicast's slow start seems to be growing, especially within the commercial sector. Recently the IETF chartered an ad hoc group, called maddogs, to explore ways to overcome the problems of IP multicast and to make it more suitable for commercial products and services.

In spite of this, however, many people seem still to accept the assumption that IP multicast should be the foundation for internet distribution. In support of this statement is the fact that, in addition to the substantial inter-domain multicast IP protocol work going on, there are research and working groups for reliable multicast transport.

There are a couple of exceptions. Researchers at Telcordia and the University of Maryland, based on a suggestion by Christian Huitema, have independently proposed the idea of host-based multicast. They developed a protocol for use in the ad hoc mobile network environment. The protocol is called AMRoute, and has been published as an internet draft in the Manet working group of IETF. This specific document targets IP multicast functionality only (by which I mean it is works across space only), and for mobile ad hoc networks only. The internet draft does, however, mention in passing that the approach may have broader functionality in the larger internet. As far as I know, however, no substantial work in this direction has been done.

Hui Zhang at Carnegie Mellon has also independently been working on host-based multicast (which he calls endsystem-only multicast). He and his students have worked out some basic low-level tree-formation mechanisms and have done some simulations (see Section 3.4). They recognize most of the broader issues, such as naming, and have independently conceived many of the ideas in yoid (such as the rendezvous), though they haven't done much work on these broader issues per se.

More recently, the Reliable Multicast Research Group (RMRG) of the IRTF has started a work item on what it so far calls Auto Tree Configuration. Among the goals of that effort is to create an algorithm that can build a tree strictly among the recipient hosts of a multicast group. As near as I can tell, however, the purpose of that tree is not to distribute content per se, which is still expected to be transmitted via IP multicast. Rather, the tree is for transmitting acks/naks and optionally for local retransmissions of packets dropped by IP multicast.

2.8 Simplicity (or Lack Thereof)

All other things being equal we all long for simple architectures and protocols. Yoid is not simple and I'm not going to pretend that it is. It constitutes a whole new layer in the architecture and comes with a number of new protocols. Nevertheless, I believe it can have a simplifying effect on the state of affairs in internet multicast and distribution overall. With yoid in place, we could jettison all of the inter-domain aspects of IP multicast—the routing protocols and the address assignment stuff. Many of the current ad hoc or proprietary server-based distribution technologies could also be replaced by yoid, making IT's life simpler. So while yoid may not be simple, it may very well be simpler than the alternatives.

3 Yoid Architecture

The key attribute of yoid, and indeed the only thing that makes it different from stuff already out there, is that it auto-configures tunneled shared-tree and mesh topologies among a group of hosts using ubiquitous internet protocols only (unicast IP and DNS). This is an extremely powerful tool. Virtually every form of distribution in the internet can be built upon this single capability.

3.1 Major Components

Each distribution *group* is comprised of two components:

1. One or more *rendezvous hosts*
2. *group member hosts*

Each group has a globally unique identifier (the *group ID*) consisting of the following three components:

1. Rendezvous Host Domain Name
2. Rendezvous Host UDP Port (optional)
3. Group Name

The group ID can be encoded as a URL and has the same syntax restrictions as a URL:

`yoid://rendezvous.name:port/groupName`

The group ID has this form for two primary reasons. First, it allows any host with its own domain name to independently create (globally unique) identifiers for any number of trees. In other words, identifier creation is simple and scalable. Second, it allows discovery of the rendezvous host(s).

The primary purpose of the rendezvous host (or just rendezvous) is to bootstrap group members (or just members) into the tree and mesh topologies (or just *tree-mesh*, when referring to both topologies together). The rendezvous does this by simply informing each (new) member of several current members, and optionally various other information about the tree (a signature for the application content, the appropriate buffer size, min and max throughputs, etc.). (The term "content" is used to refer to data distributed by the application using the yoid group.) Once this is done, the new member joins the tree-mesh by talking only with current members—one or more of those that it learned from the rendezvous, and others that it in turn learned from those members.

This initial joining is the only time a member must talk with the rendezvous. It may, however, talk to the rendezvous in several other cases:

- It should inform the rendezvous, if possible, when it quits the tree.
- It informs the rendezvous if it becomes the root of the tree (for the purpose of tree partition discovery and repair, though this is generally done independently of the rendezvous).
- It responds to pings from the rendezvous (which the rendezvous uses to insure that its list of members is accurate).

Every attempt is made to keep the rendezvous' activity to a minimum. The rendezvous is not attached to the tree-mesh per se (it does not receive or transmit distributed content). Almost everything needed to maintain the tree-mesh is done via members talking directly to each other. If every rendezvous is disabled (crashes or loses connectivity), the tree-mesh continues to function. New members, however, will not be able to discover members and therefore will not be able to join the group.

Because every member must first go through a rendezvous to learn of other members, the rendezvous is an appropriate place to put other important functions related to the group. Foremost among these is security. The rendezvous can convey signatures about the content being sent over the tree-mesh, can apply access control to the member, can convey authentication information and credentials for joining the group, and so on. (Understanding of course that once a rogue host does join a group, for instance because it somehow got the right credentials, there is nothing the rendezvous or other members per se can do to prevent it from passing content on to any host.) The rendezvous may also be a convenient place for accounting, billing, and so on.

3.2 Yoid Tree and Mesh Topologies

Yoid generates two topologies per group—a *shared tree* (or just *tree*) topology, and a *mesh* topology. Both figure large in a group's content distribution. The tree is the primary means of distributing content due to its relative efficiency.

But the tree is fragile—a single crashed or disconnected (or just plain overloaded) member can partition the tree, at least until the tree reconfigures itself. The mesh is not fragile. Its connectivity is rich enough that the simultaneous loss of a number of members will not partition it with high probability. The mesh serves a number of purposes, including where appropriate the distribution of content, all of which are needed to compensate for the fragility of the tree. These purposes include:

- Discovery of tree partitions.
- Distribution of content, for instance when the tree is partitioned or when very robust distribution is required.
- Detection of member unreachability.
- Notification of member unreachability.
- Verification of content reception (content checksum or signature).

The tree topology is not a subset of the mesh topology. The two topologies are created for and optimized for different purposes, and so their creation is largely independent. The tree is optimized for efficiency while the mesh is optimized for robustness.

The yoid stack of protocols includes a protocol for end-to-end reliable delivery (YDP, the Yoid Distribution Protocol, described later). The reliable component of YDP contains a sequence number space for identifying anything transmitted by any member. This is used to prevent looping when broadcasting over the mesh. Its use when multicasting over the tree is optional, but is used whenever content reliability is required. When used in tree multicast, it can also detect and prevent loops in the tree.

Yoid Tree Topology

As stated above, yoid uses a shared-tree for distribution. A transmitting member may be at any place in the tree (a leaf, the root, or some other branching point). A member can receive a content frame from any tree neighbor, and will forward the frame to all other tree neighbors. We use the word frame here instead of packet because members receive, replicate, and forward content in units of application frames, each of which may consist of multiple IP packets. In what follows, we use the word packet only when referring to IP packets, and we use the word frame when referring to the content units that the application and yoid control protocols deal in.

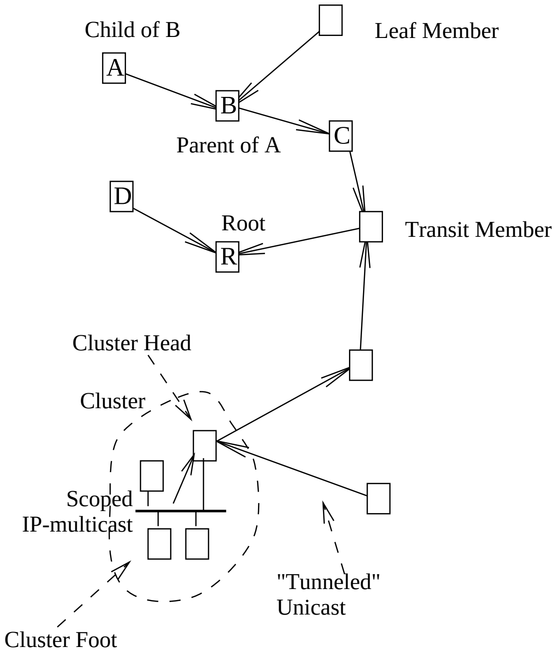

Figure 1: Yoid Tree and Terminology

Figure 1 shows a yoid tree and gives much of the terminology associated with it. Each box in the tree represents a member (the rendezvous is not shown). The solid arrows represent the “links” of the tree. The directionality implied by the arrows does not refer to the flow of frames, which can be in either direction. (Note that it is possible to constrain a tree so that only a given member or members can transmit. This information would be conveyed by the rendezvous at member join time, and enforced by all members.) Rather, they refer to the relationship between neighbor members in the tree.

A member may receive and transmit frames either via unicast IP or scoped IP multicast. In the current architecture, IP multicast is tightly scoped, typically by a hop count of 1 (i.e., to a single physical media), or in some cases a hop count of 2. More relaxed scoping using administrative scoping should be possible but hasn't been well thought out.

The relationship between two neighbor members over unicast IP is that of parent/child. Members attempt to find tree neighbors that are as close to them as possible (where closeness is determined by the latency between two members). Where multicast IP is used, a set of members are grouped as a *cluster*. One member of the cluster is elected the head, and is responsible for establishing a (unicast IP) parent neighbor, thus bridging the cluster with the rest of the tree. The other cluster members are called feet, and transmit and receive to/from the tree via the head. Cluster members are by definition close to each other, since they share a locally scoped IP multicast. The solid line arrows point from child to parent, or from foot to head.

Each tree must have a single root, which by definition is a member with no parent or head. (There may transitionally be zero or more than one root, but in steady-state there is exactly one.) All other members have exactly one parent or head.

Each member, at a given time, is a transit member or a leaf member, depending on whether it has a multiple neighbors or a single neighbor respectively. A member may intentionally limit itself to being a leaf only. While not related to the tree topology per se, we mention here that a member may be an *endhost* or a *yoid proxy server* (or just server, where the context is clear). The distinction is primarily that an endhost contains the application that is using the tree, where a server doesn't. Typically, a server would be a box that has been installed in the network infrastructure for the explicit purpose of acting as a transit member in a tree.

Figure 1 shows a sparse topology (low fan-out). This is intentional. While the tree management algorithm nominally allows any fan-out, a small fan-out appears ideal for a tree where all members are endhosts. A small fan-out maximizes the bandwidth available at each transmitting member, minimizes the amount of work each member must do, and minimizes forwarding delay at each member. A small fan-out also minimizes the amount of short-term topology change that must occur when a member quits the tree.

While testing and experimentation is needed to find optimal fan-outs for various traffic profiles, I tend to think in terms of a fan-out of two for endhost-based trees. For a cluster head member, the cluster itself counts as one "neighbor" for the purpose of determining fan-out.

For trees where the transit members are mainly servers, a larger fan-out may make more sense, especially where for policy or other reasons it is desirable to limit endhosts to being leaf members. In this case, the fan-out may be large simply because the number of servers per endhost is small.

Yoid Mesh Topology

Each member maintains a small number of neighbors solely for the purpose of insuring that there is a non-partitioned topology over which frames can be broadcast. These neighbors are called mesh neighbors. For the purpose of a robust broadcast to all members, both mesh and tree neighbors are used. In spite of this, *mesh topology* refers only to the topology consisting of mesh neighbors.

(The term *multicast* refers to delivery over the tree (with occasional transmission between non-tree neighbors). The term *broadcast* refers to delivery over the mesh-and-tree. Note that these terms are subtly different from their IP counterparts. With IP, multicast refers to transmission to a specific group of hosts (those that have joined the multicast group), whereas broadcast refers to transmission to all hosts. With yoid, on the other hand, the group of receivers for both multicast and broadcast are the same.)

To insure a non-partitioned mesh topology, each member M establishes a small number of other members—three or four—as mesh neighbors. These members are randomly selected, with the exceptions that they 1) must not include members that are tree neighbors, and 2) must not include members that have already established a mesh link to member M. The reason for this latter restriction is to prevent trivial cliques, where three or four members all use each other as mesh neighbors, thus partitioning themselves from the rest of the mesh topology. It is this restriction that makes the mesh topology robust.

Assuming that each member establishes the same number of mesh links N, each member will have on average $2N$ mesh links, N established from the member to other members, and N established from other members to them. Of course the actual number may be greater or less than $2N$ since mesh neighbors are randomly chosen. The number may be less than N is the group is small.

Efficient random selection is achieved through a frame delivery mode called "mesh anycast", whereby a discovery message takes a random walk along the mesh, randomly stopping at some member (discussed later). In particular, no attempt is made to find mesh neighbors that are nearby (latency-wise).

Other Kinds of Member Relationships

In addition to tree and mesh neighbors, members maintain communications with still other members for various purposes. For instance, members search out and find a few other members that they can adopt as parents on a moments notice should their current parent quit the tree or become unreachable. They maintain connections with these members, and stay up-to-date as to the members' appropriateness as parents.

The rendezvous needs to maintain knowledge of at least a smallish number of members (ten or so), and

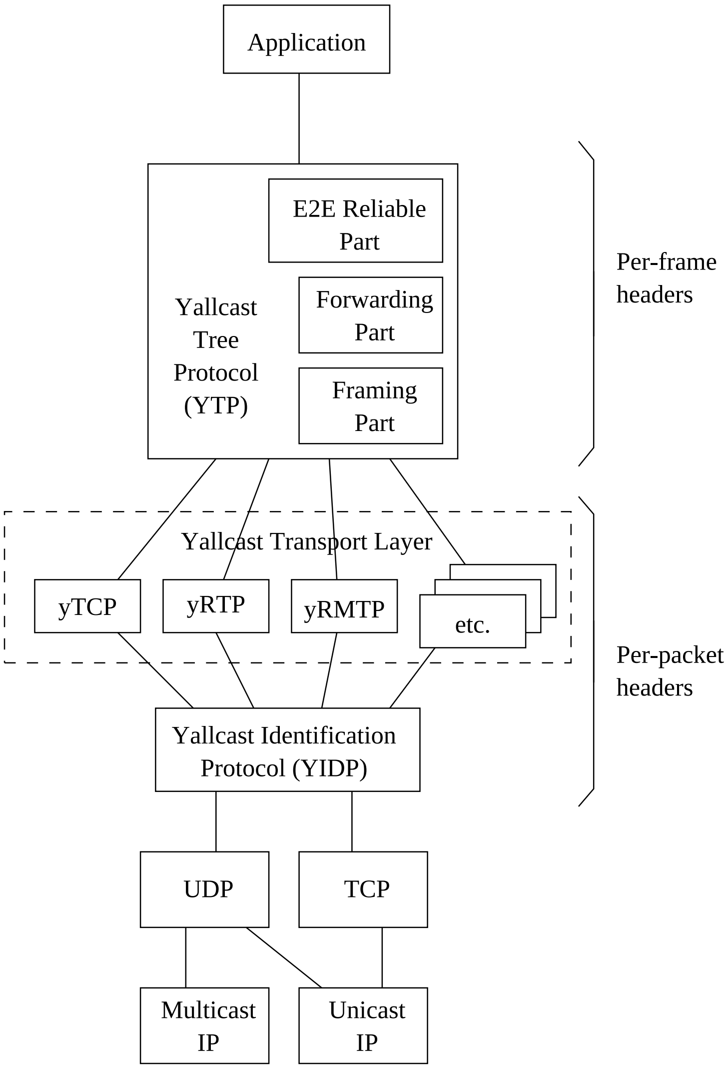

Figure 2: Yoid Content Protocol Stack

so a member my find itself in occasional communications with a rendezvous.

3.3 Content Protocols

The stack of yoid protocols for transmitting content is shown in Figure 2. Yoid protocols run directly over UPD or TCP.

Yoid Identification Protocol (YIDP)

Starting from the bottom, the yoid protocol immediately encapsulated by UDP or TCP is YIDP, the Yoid Identification Protocol. The YIDP header is attached to every packet (as opposed to every application frame as with some other yoid protocols). Its purpose, essentially, is to make up for the identification capabilities long lost from IP and TCP/UDP. Host identification is primarily based on domain name, and not at all on IP addresses or TCP/UDP port numbers.

YIDP has specific mechanisms to deal with firewalls, NAT boxes, and dialup hosts (by which I mean hosts with dynamically assigned addresses and often no domain name). All this is necessary in yoid because, unlike today's situation where most hosts (i.e. clients) typically only initiate communications, yoid requires that all hosts be able to both initiate communications with others and allow others to initiate communications with them. This includes hosts that may be behind firewalls and NAT boxes, that have dynamically assigned addresses, and that may not have a domain name.

YIDP identifies

1. the group the packet is related to,
2. the sending member, and
3. in the case of unicast IP, the receiving member, and in the case of clusters, the receiving cluster.

The group is identified by the Group ID, already given above.

Members are identified using only the member domain name and the member YIDP port. The YIDP port number is unique among all instances of members using the same member domain name in a given group. The member domain name must in turn be unique among all members in a given group. Note in particular that we do not use anything at or below UDP/TCP for member identification.

The member name is syntactically a domain name. If a member has a globally unique domain name, then it can be used as the member name. If not, one will be assigned to it, per group, by the rendezvous. The domain name does not have to be a "working" domain name, in that it does not have to be discoverable by DNS. Indeed, a member with a working domain name may wish not to use it for privacy reasons (though if it has a global and permanent IP address, then much of the privacy benefit is lost).

The YIDP port is analogous to the port numbers of TCP and UDP. It specifies the path of protocols from YIDP up to and including the application for a given member.

Because the various identifiers used by YIDP are large, they are all compressed into a single 64-bit tag. In the case of unicast IP, the tags are different in either direction, with each member assigning the tag value that its neighbor must use to send packets to it. The tag values are conveyed at the time the two neighbors establish a connection using a simple negotiation protocol.

In the case of clusters (multicast IP), the same tag is used by all cluster members, and is assigned by the cluster head. It is conveyed by the cluster head at the time it becomes the cluster head. (When a new member becomes the cluster head, it normally adopts the same tag value.)

YIDP has several features that make it compatible with NAT/PAT boxes and firewalls. First, YIDP host identification is not messed up by NAT/PAT boxes because identification is independent of IP addresses or UDP/TCP port numbers, any of which may be modified by a NAT/PAT box.

Second, because YIDP headers contain full names, a YIDP-smart NAT box or firewall can handle connections initiated from either direction, not just from behind the firewall. This is possible because the yoid header on packets arriving from outside the firewall contain the name of the target host behind the firewall. The firewall can do a DNS lookup and determine the appropriate IP address to transmit the packet to the target host.

The Yoid Transport Layer: yTCP, yRTP, yMTCP, etc.

Next above YIDP are a number of protocols used between two neighbors or within a cluster to do such things as flow control, congestion control, sequencing, lost packet recovery (retransmissions) and the like. I envision four protocols at the yoid transport layer. The equivalents of TCP and RTP (yTCP and yRTP), and multicast equivalents of TCP and RTP (yMTCP and yMRTP, to give them a name). Note that the multicast equivalents can be much simpler than the reliable multicast protocols being developed for the wider internet today, because of the strict scoping.

It is valid to run yoid over TCP rather than UDP, in which case there would not need to be any protocol running at the yoid transport layer. It is very convenient, however, to be able to run all of yoid over a single UDP port, primarily for the purpose of dealing with firewalls and other boxes that must filter on port number. For this reason, the preferred stack is yTCP (or other yoid transport) over YIDP over UDP.

Yoid Distribution Protocol (YDP)

Next in line is the Yoid Distribution Protocol (YDP). This protocol handles everything needed to move an application frame end to end over the tree-mesh with near-zero probability of loss. Reflecting the broad functionality of yoid, YDP is itself rich in functionality. It handles framing, determines the type of forwarding (multicast, broadcast, etc.), has a (hop by hop) pushback flow control mechanism, and has a hop count. YDP can also identify the final destination(s) of the frame and the original source of a frame. In essence, YDP is to a yoid topology what IP is to a router/host topology.

In addition to all this, YDP has a sequence number space that can sequence and uniquely identify every frame originated by any source. This sequence number is used not only to insure end-to-end reliability and ordering of frame delivery, but also to prevent looping of frames and duplicate delivery of frames to members acting as transits. This allows frames to be transmitted over the tree, the mesh, or a combination of both (for instance, in response to a temporary partition of the tree).

YDP Source Identification

Source identification in YDP is optional, but typically used (I mean here the true originating source, not the previous hop sender identified by YIDP.) The source is identified by its domain name and YIDP port number. The source identification is carried in an option to the YDP header. The fixed part of the header carries a 16-bit source identification tag. When a member transmits a source identification option, it also transmits a tag to associate with that option. When its neighbor acknowledges receipt of the option, subsequent frames from that source carry only the tag.

YDP Forwarding Modes

YDP has five forwarding modes. They are: multicast (over the tree, primarily), broadcast (over the mesh), two types of anycast (over the tree and over the mesh), and unicast. The two types of anycast are different from IP's anycast. Unlike anycast IP, which targets a specific group of hosts (those with the anycast address), and typically selects the nearest, YDP anycast simply causes a frame to (randomly) walk the tree or mesh until it (randomly) decides to stop (or the hop count expires). Anycast is used by the Yoid Tree Management Protocol (YTMP) for discoverying arbitrary members in the topology, allowing members to build up a knowledge base about other members.

Unicast allows a frame to be routed over the tree to a specified target member. At first glance this might seem silly, since any member could just as easily transmit the frame directly to the target member using IP. The purpose of unicast, however, is for the case where many members are sending frames to a single member. Doing it over the tree has two advantages. First, the target member does not need to maintain a separate connection to every sending member. Second, the pushback flow control over the tree prevents the target member from being congested (should every member, for instance, decide to transmit at once).

For unicast to work, routing information must be installed in the members. Any member can cause routing information pointing to itself to be installed by multicasting a frame with itself identified as the source, and with a flag indicating that the reverse path back to the source should be remembered. When the topology changes, each member updates its new neighbor as to which "sources" are behind it. The intent here is that routing information is maintained for only a few members, or else there would be considerable traffic exchange at each topology change.

YDP Pushback Flow Control

Every member, acting as a transit, has a certain amount of buffer space for a given tree for storing frames in transit. This amount can be described in terms of space (e.g. 500 kbytes), or time (e.g. one minute), or a combination (e.g. 500 kbytes or one minute, whichever is greater). The amount, however, is the same for all members in the tree, and is one of the parameters given a member by the rendezvous when it joins.

If the buffer fills, the member must either drop frames for outgoing neighbors or pushback incoming neighbors. To determine which, each tree also has a minimum transmission rate associated with it. When the buffer fills, incoming neighbors are pushed back to this rate. If the buffer remains full (because an outgoing neighbor is even slower than this minimum rate), then frames for that neighbor are dropped.

The current pushback mechanism is a single bit transmitted to the neighbor being pushed back. When a member receives this bit set, it stops transmitting to its neighbor. When the bit is not set, it transmits. When frames are being received from several neighbors, the neighbor(s) chosen for pushback are those with the highest per source transmission. In other words, if a member is receiving from two neighbors at roughly the same rate, but one neighbor is transmitting frames from fewer sources, that neighbor will be pushed back first.

Note that this is by no means the only pushback mechanism or policy. For instance, the pushback could specify a transmission rate rather than simple on/off. Or, the pushback could specify certain sources (those sending the most). The best pushback mechanism is an issue for further study.

YDP Sequence Numbering

As mentioned above, every YDP header contains a sequence number. This number sequences the bytes transmitted by any source. Right now I'm assuming the sequence number is 64 bits in length. The sequence number is initialized by each source at a default low value (1024 in the current implementation), and continues until it reaches the max all-ones. (In the astonishingly unlikely event that this max is reached, the member must quit the group and rejoin with a different YIDP port number.)

The YDP sequence number is of course used to insure end to end reliability (in-order delivery of all frames). But it does more than that. It is used to prevent looping during broadcast. It is used to prevent frame loss during topology changes. It can even be used to help determine which members become neighbors in the tree.

More generally though, it allows any multicast frame to be transmitted to a non-tree neighbor without concern for looping. This means that non-tree neighbors can be used to augment multicast delivery when some tree neighbor is temporarily unable to forward multicast content. It allows yoid to respond very quickly to short-lived lapses in the tree without having to reconfigure the tree.

Exactly how this gets done depends on the nature of the content—its volume, how much gets buffered by each member, how much delay can be tolerated, and any constraints on the order in which the content can be received.

Without getting into details, I envision some kind of sequence number negotiation protocol between both tree and mesh neighbors whereby neighbors (or potential neighbors) tell each other what they have and what they would like to receive from the neighbor. (I don't get into details here in part because the full protocol hasn't been designed yet.)

As an example, consider a file distribution application whereby the entire file is buffered by each member, and there are no delay constraints to speak of (that is, while the file should be received as quickly as is reasonably possible, nothing breaks if it takes a few minutes more or less). What's more, the entire file must be received by each member, but the order in which the bytes are received is not important. In other words, the file is not used by an application until it is received in its entirety.

Here, when two members are considering becoming neighbors on the tree, they would exchange information on what sequence numbers each has already received. They would also determine which of the two was closer in the tree to the source of the file. Based on this, they would determine if they should become

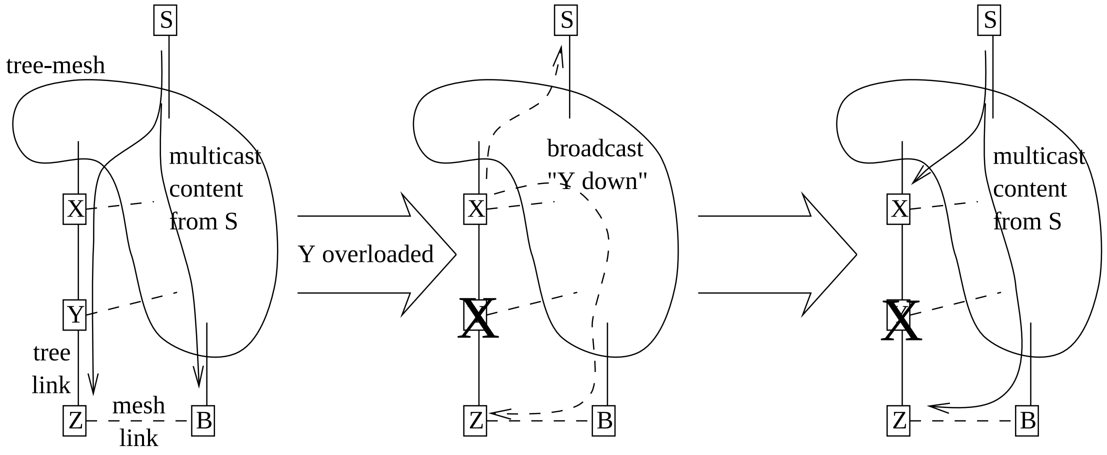

Figure 3: Fast Recovery From Overloaded Node

neighbors, and if so, which should send to the other and from what point in the file (see Section 3.5).

As another example, consider a voice conferencing application. Here, content is real-time so there is virtually zero buffering. A lost packet here or there may be tolerable, but multi-second loss of transmission (for instance, in order to detect that a neighbor is down and then respond through reconfiguration of the tree) is not tolerable.

By way of explanation, assume that member X at a given moment is transmitting content from source S to member Y, and that Y in turn is transmitting to member Z (see Figure 3). Assume that member Y becomes temporarily overloaded and stops receiving and transmitting. Z may not be able to detect this quickly, but X can because it receives no acknowledgement in the expected time (accepting here that this could be simply because of an unusually long network delay). X can broadcast a message indicating that Y is unreachable. This will quickly reach Z, which can in turn request from some non-tree neighbor that content for source S be transmitted to it.

Because this could happen to several members in different parts of the tree at the same time, the sequence numbers are needed to quell duplicates. If Z again hears from Y, it would tell the non-tree neighbor to stop transmission. If it does not hear from Y after a brief timeout, it could find a new tree neighbor (ideally the non-tree neighbor it is already receiving from, if appropriate).

There are other variations on this theme, but the basic idea is that members can use the sequence numbers and various negotiation or notification protocols to manage transmission over the tree and allow transmission over non-tree links. The best set of mechanisms will probably depend on the needs of the application and will require considerable work and experimentation to tease out.

3.4 Yoid Tree Management Protocol (YTMP)

For my money, the most interesting and in most ways the hardest thing about yoid is how the shared tree gets built and maintained. This is done by the Yoid Topology Management Protocol (YTMP). YTMP runs in every member and in the rendezvous, though since the rendezvous is not part of the tree per se, the tasks it performs are different from that of the members. YTMP is responsible for configuring both the tree and the mesh, but tree configuration is the much harder problem and is what I talk about here.

Before talking about YTMP specifically, I would like to broaden the context and talk about general issues related to dynamic shared tree configuration for any tunneled topology.

Two Basic Approaches to Tree Management

There are two basic approaches you can take for configuration of a tunneled tree topology. I call them the tree-first and the mesh-first approaches. In the mesh-first approach, members maintain a connected mesh topology among themselves. A root member is chosen (for instance, the member doing the most sending), and a (reverse-path) routing algorithm is run over the mesh relative to the root to build the tree.

This approach is similar to how routing (which is essentially tree building) is normally done. A mesh topology (for instance, the physical topology of routers and links, or the tunneled mbone topology) is explicitly created. A routing algorithm is run over this, and a tree falls out. In other words, the mesh topology is known and intentional. The resulting tree topology is unknown, but if a good mesh is chosen, the tree likewise will be a good one.

Both the AMRoute and Carnegie Mellon methods are mesh-first. In the case of AMRoute, the mesh is created through an expanding-ring search utilizing the underlying broadcast capability of the mobile network. Because the underlying broadcast is sensitive to topological closeness, the resulting mesh is naturally a relatively efficient one. In the case of Carnegie Mellon, the mesh itself has a broadcast mode (similar to yoid), and each member periodically broadcasts its presence over the mesh. Members measure the distance between themselves and other members, and select the closest members as mesh neighbors. The quality of the mesh improves over time.

To generate the tree from the mesh, AMRoute uses a periodic broadcast of a message from the root to select out the tree. In the case of Carnegie Mellon, a DVMRP-like algorithm is used.

In the tree-first case, on the other hand, the tree is built directly in that members explicitly select their parent in the tree from among the members they know about. Something that looks like a routing algorithm is run over the tree after the fact, but for the purpose of loop detection (that is, verifying that the tree is indeed a tree), not for the purpose of creating the tree per se. There is no intervening mesh topology. This is the approach taken by yoid.

Considering that yoid does employ a mesh, this might seem confusing. In fact, the mesh used in yoid is created separately from the tree—the tree is not a subset of the mesh in any way. Nor is there any attempt to make the mesh efficient in the sense that neighbors in the mesh should be near each other. Rather, the goal is to make the mesh "well connected", and therefore robust.

The reason yoid uses the tree-first approach rather than the mesh-first approach is that the tree-first approach gives us more direct control over, well, the tree. This control is valuable for various aspects YTMP, such as maintaining strict control over fanout, selecting a parent neighbor that has buffered the appropriate content, or responding to failed members with a minimum of impact to the tree (see Section 3.5).

Direct control over the tree may also be useful for policy. Indeed, if you look at BGP, much of its functionality is in support of being able to manipulate policy—that is, which nodes traffic is received from.

While it may be possible to manipulate the tree topology to a significant extent via the selection of mesh neighbors and metrics between mesh neighbors, it seems much more straight-forward to be able to manipulate the tree directly.

Having said that, I must confess that the mesh-first approach never occurred to me. This in spite of the fact that the mesh-first approach seems more obvious in-so-far-as it is more similar to other kinds of routing. In other words, I have not thought deeply about what kinds of control one can get out of a mesh-first approach, nor what other advantages might accrue. For instance, it may turn out that the mesh-first approach produces more efficient trees. In addition, not having to think about how to manipulate the tree per se may make the mesh-first approach simpler to work with.

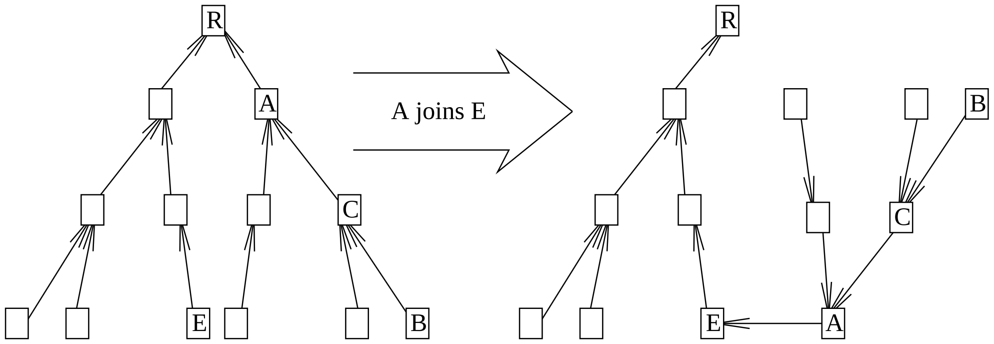

Figure 4: Example of a Long Diameter

While I hope that others can explore the pros and cons, I nevertheless find the tree-first approach a natural way to go about building a tunneled tree, and intend to continue with this approach.

The Tree-first Approach

To reiterate, in the (yoid) tree-first approach, each member that cannot join a cluster or is the head of a cluster is responsible for either finding a parent in the tree, or deciding that no other member can be a parent, and so becoming the root of the tree. The root of the tree is circularly defined as that member that does not have and cannot find a parent. A member is not responsible for finding children—children will find it, though it is of course free to reject members requesting to be children.

One effect of this is that each member acts relatively independently. This is a good thing, as it keeps the protocol simple (less multi-member coordination). Another effect of this is that when a member joins a new parent, it drags all of its offspring with it. That is to say, its children remain its children, its children's children remain their children, and so on. This is good in that none of the offspring have to change neighbors because of their ancestor's change. It is not necessarily good in that it can result in lop-sided trees with rather longer-than-necessary diameters, as Figure 4 illustrates.

This may not be much of a problem for latency-insensitive applications, like file distribution or one-way radio transmission. For latency-sensitive applications, like for instance an internet game, it may be a critical consideration. As a result, I am finding coordination between members slowly creeping into the design, and imagine this will continue as the protocol is fleshed out. (Indeed, if an application had a hard upper bound on maximum end-to-end latency, one can imagine the need for a centralized topology manager to determine the best topology and dictate to members who their parent in the tree should be.)

I said before that in the tree-first approach, members choose their parents, and that the "routing algorithm" runs after the fact to do loop detection. While this is true, members also go out of their way to screen potential parents whose selection as a parent would result in a loop. Note that we don't do loop prevention—only loop avoidance. We can, however, always detect loops after the fact.

The primary means of screening potential parents is through the *root path*. The root path of any given member M is the set of members in the path from M to the root of the tree. This is completely analogous to the AS path in BGP. Whenever a member obtains a parent, its root path becomes that of the parent with itself appended. Every member always immediately transmits any changes in its root path to its children.

This is what allows loops to be detected (after the fact). If a member receives a root path from its parent that contains itself, then there is a loop and the member must find another parent. The root path also allows members to avoid selecting parents that would result in a loop. Simply put, if a member M sees

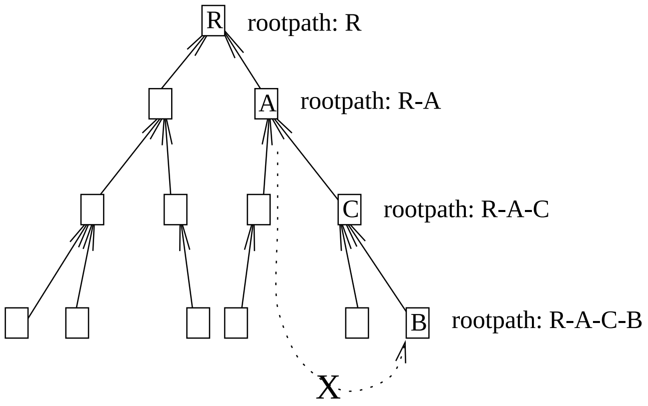

Figure 5: Screening Potential Parents Using the Root Path

itself in the root path of another member P, then M cannot select P as a parent (see Figure 5).

This is fine as far as it goes, but the tricky part is in dealing with simultaneous topology changes. Two or more simultaneous or near-simultaneous changes can easily result in loops. For instance, in Figure 5, if C had changed from some other place in the tree and just joined A, and B had not yet learned of its new root path, A would not know not to join B. A loop would result and be detected when C advertised its presumed root path R-A-C to B, and B advertised R-A-C-B to A (or simply noticed that its new child A was in its even newer root path R-A-C-B).

To avoid these loops, yoid employs two mechanisms, depending on whether the member joining a *prospective parent* already has a parent or not. A prospective parent of member M is one whose root path has already been checked by M, and that M is actively seeking to join. Note, however, that a member that has no children does not need to employ these mechanisms or indeed check the root path of other members. A child-less member cannot form a loop no matter where it attaches.

Coordinated Loop Avoidance Algorithm

If a member already has a parent when it is getting a new one, it uses the so-called *coordinated loop avoidance algorithm.* The main reason a member would get a new parent when it already has one is to improve performance—most typically, because the prospective parent is closer to the member than the current parent.

To understand how it works, first lets examine more carefully what kinds of simultaneous changes can produce a loop. Looking at the left side of Figure 6, assume that C wants to join B (that is, wants to become the child of B). Assume further that the topology has settled down from any past changes—all of the root paths are up to date and reflect the topology shown.

Under these circumstances, the only simultaneous changes that would cause a loop without being preventable in advance by the examination of potential parents' root paths is B or D trying to join C or any of C's offspring (E and F). Specifically, Figure 6 shows the case where D is simultaneously joining F. Allowed to happen, a loop forms and, not incidentally, the tree is partitioned.

The reason is easily apparent, and can be generalized by the following statement: Any member X in C's new root path cannot move to a position in the topology that places C in X's new root path—that is, C

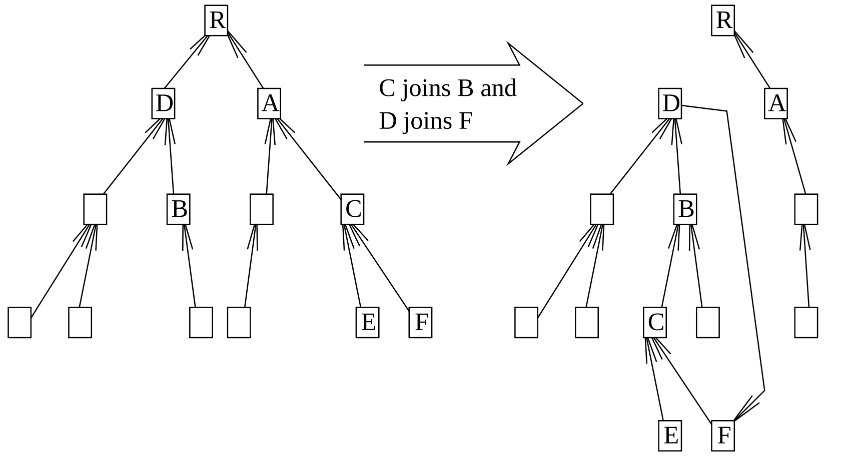

Figure 6: Loop Formed by Two Incompatible Changes

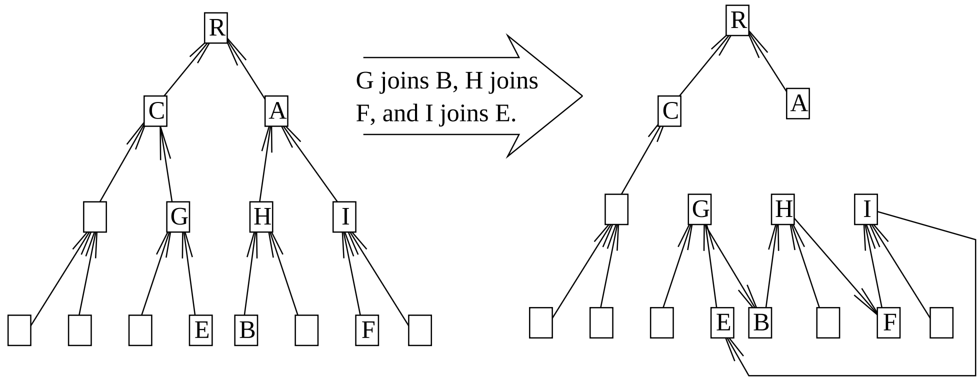

Figure 7: Loop Formed by Three Incompatible Changes

or one of its offspring. Or, more general still: Two members X and Y cannot change parents such that X is in Y's new root path and Y is in X's new root path. Or, in the context of topology rather than root paths: Two members X and Y cannot change parents such that X's new parent is an offspring of Y, and Y's new parent is an offspring of X.

This problem is not limited to pairs of members. A loop also forms among three members X, Y, and Z if simultaneously X becomes an offspring of Y, Y an offspring of Z, and Z an offspring of X. This is shown in Figure 7, where any two of the three changes do not result in a loop, but all three together do.

The coordinated loop avoidance algorithm has a coordination mechanism designed to block incompatible changes with a minimum of overhead and synchronization. I'll describe it by way of example, referring again to Figure 7. When G goes to join B, it sends a so-called *intent to join* message along the tree from G to B. That is, the message traverses C, R, A, and H before reaching B. The intent to join contains the full path it needs to traverse. The path is used both as a source route for forwarding the message, and as a mechanism for blocking incompatible joins, as follows.

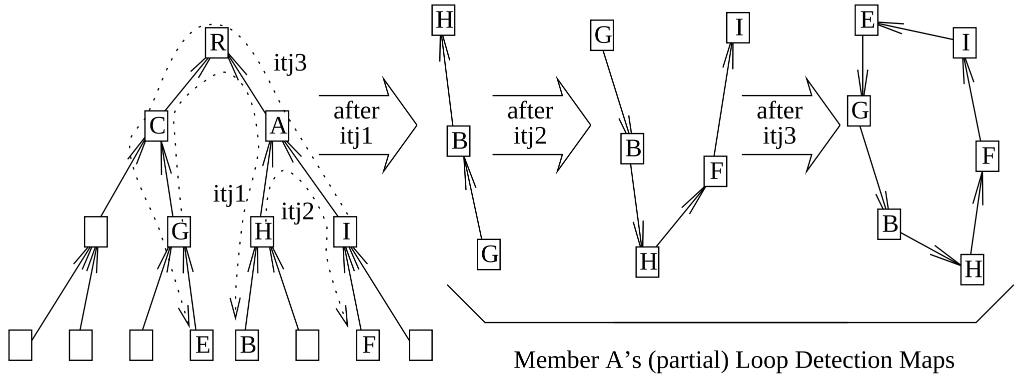

Figure 8: ITJ Messages and Member A's Loop Detection Maps

Each member receiving the message stores its contents, and builds a *loop detection map*—a map of of what the effected portion of the tree would look like after the described join occurs. Figure 8 shows the maps that would be produced by member A after receiving, in succession, G's intent to join (itj1), H's intent to join (itj2), and I's intent to join (itj3). (The maps shown do not include everything A would normally store, such as members R and I. Rather, they show only what is pertinent to the example.) As shown, A can determine that a loop would form if I were to join E, assuming that G had joined B and H had joined F.

Members forward any intent to join that:

1. is forwardable (the next member indicated in the path is indeed a neighbor in the tree),
2. would be forwardable were the changes in the loop detection map brought about, and
3. does not result in a loop in the loop detection map.

All other intent to joins are stamped as rejected and replicated, with one copy going back the way it came over the reverse path to flush it from the loop detection maps, and the other going directly to the originator. (To facilitate the flushing, the originator in turn passes it along the path in the forward direction. A member receiving a rejected intent to join for which it has no stored entry simply discards it.)

An intent to join that reaches its intended target (the prospective parent) without incident is stamped as accepted and sent directly to the originator. If the originator fails ultimately to join the prospective parent, it resends the intent to join back along the path, stamped reject, to flush its corresponding map entries. An intent to join's map entries can also safely be flushed anytime any portion of its path changes. In any event, all unflushed intent to join map entries are flushed after a relatively short timeout. The exact time is a trade-off between preventing loops and blocking otherwise valid tree changes.

Note that, contrary to what might be presumed from the above description, it is emphatically not the case that any possible loop can be detected by at least one member. There exist looping combinations for which no single member would detect the loop. This is the reason for the second of the three criteria for forwarding an intent to join. I believe (though I'm not sure) that the addition of this criterion is sufficient for preventing loops, but it can also block otherwise valid intent to joins.

This is almost certainly worth the tradeoff because the coordinated loop avoidance algorithm only happens when members already have parents and are just looking for a more optimal situation. When a parent change is blocked by the second criterion, the member need only backoff and try again later.

Emergency Loop Avoidance Algorithm

If a member does not have a parent when it is joining a prospective parent, then it uses the so-called *emergency loop avoidance algorithm* to check for loops. In this algorithm, the joining member has the prospective parent initiate what is called a *root path trace*. A root path trace is a message that is forwarded by each receiving member to its parent if it has one, or its prospective parent if it doesn't have a parent. The root path trace message records the path as it travels.

When a member receives it, it returns it to the original joining member if:

1. it finds itself in the recorded path, thus indicating a loop,
2. it find the joining member in its own root path, thus indicating that if the root path trace were forwarded it would eventually reach the original joining member, thus indicating a loop, or
3. it cannot forward the message because it has no parent or prospective parent (typically the root).

Otherwise it forwards the message to its parent/prospective parent.

Finally, if the original joining member receives the root path trace from a child, it knows that there would be a loop were the member to join its prospective parent. If a loop is detected via one of the above methods, the joining member either tries another prospective parent, or backs off and tries the same one a little later.

Emergency loop avoidance is more general than coordinated loop avoidance in that emergency loop avoidance can be used whether or not the changing member has a parent. In other words, coordinated loop avoidance is strictly speaking unnecessary. The reason I have both mechanisms instead of just the one is that coordinated loop avoidance places a lighter load on members "high up" (near the root) in the tree.

The root path trace of emergency loop avoidance typically goes all the way to the root, meaning that the root gets involved in almost every (emergency) parent change in the tree. The intent to join of coordinated loop avoidance, on the other hand, goes only as far up the tree as necessary. Of course, the intent to join may traverse the root. But more normally, as members find closer parents, the intent to join should traverse only members further from the root.

3.5 Parent Discovery and Selection in YTMP

The above description of loop avoidance assumed two things:

1. that the member obtaining a parent had a list of members from which to choose the parent, and
2. that the member knew which member from that list would make a good parent.

This parent discovery and selection is an important component of YTMP, though somewhat independent in the sense that different parent discovery and selection techniques can work with the YTMP tree management algorithms. This section outlines some of the issues and approaches to parent discovery and selection.

A key attributes of YTMP is that it scales to very large groups (thousands of members). Parent discovery and selection must likewise be scalable, which means that, except for small trees, members can't be expected to know of all other members, much less the distance to those members. Second, there are a number of factors, some of them orthogonal or even contradictory, that a member must consider in picking a parent.

I don't have any silver bullet approach to parent discovery and selection. Rather I have in mind a number of heuristics. This is an area where we need a lot of experimentation, both to find out how important good parent selection is, and to find good ways to do it.

When Parent Discovery and Selection is not Necessary

Just to be clear, I want to make sure the reader understands that all of this applies only to the situation where a member cannot join a (local IP multicast) cluster. The first thing a newly joining member (newcomer) always does is to check to see if it can join a cluster. It can even do this without contacting the rendezvous, if there are no other reasons, such as security or accounting, to contact the rendezvous.

All clusters for any given tree have a deterministically selectable IP multicast address created by hashing the Group ID. If a cluster has been established, which would be the case where other members already exist within the tight scoping, the newcomer simply joins the cluster (under most but not all circumstances, see below). If a cluster has not been established, the newcomer must then find a parent in the tree at large. In any event, however, the newcomer continues to listen on the cluster IP multicast address for other newcomers. (There is an algorithm, not described in this document, for how members discover other members on the cluster and elect a cluster head.)

Parent Discovery

Each member must maintain a list of potential parents. A potential parent at a minimum must not have the member in its root path, and must have capacity for accepting new children. The member does not in general maintain a connection with each potential parent, so information in the list is always to some extent out of date. The member does, however, try to maintain a connection with the most promising two or three *stand-by* potential parents and keep up to date with their status so that it can obtain a new parent quickly should it need to.

There are several means by which members can learn about potential parents. Certainly a newcomer learns of some when it contacts the rendezvous. In fact, for small groups, the rendezvous can keep a full list of the group membership and pass this onto newcomers. A member can also learn of potential parents from information contained in various control messages, such as other members' root paths or intent to join paths.

Otherwise, there are two primary means of discovering potential parents—one relaxed and one rushed. The rushed method is used when a member does not have enough stand-by potential parents (or worse, has no stand-by and no parent). This would be the case where the former stand-by potential parents either left the tree, or obtained a full complement of children.

What the member does is query the former stand-by, or any other potential parent in its list, to learn of its children. It then queries one or more of the children to see if they are appropriate stand-bys. If not (they too have full complements of children), it in turn queries their children, working its way leafward depth first. In this way a valid stand-by is found relatively quickly.

The relaxed method is used when a member has a parent and enough stand-bys, and is simply looking to keep its list of potential parents more-or-less up to date. To do this scalably, each member transmits a member discovery message to its parent using YDP tree-anycast. Frames sent via tree-anycast are pseudo-randomly sent neighbor to neighbor along the tree, and pseudo-randomly stop at some member. This member answers the message, thus getting discovered. It is then added to the list (or refreshed if it is already there), and if the list if full, the oldest entry may be taken out.

The pseudo-random neighbor selection we use favors going leafwards over going rootwards, making discovery of nodes closer in the tree more frequent, though by no means exclusive. By initially sending the discovery message to its parent, the member insures with high probability that the discovered member won't be an offspring (and therefore invalid as a parent).

Partition Discovery

As stated earlier, when a member cannot find a valid parent at all, it assumes it is the root of the tree. Of course it is possible that multiple members may fail to find a parent and all consider themselves the root of the tree. This results in a partitioned tree—one partition for each root.

When a member becomes a root, it both

1. periodically broadcasts an "I am the root" message over the mesh, and
2. periodically informs the rendezvous that it is the root.

If multiple members are doing this, then unless the mesh is also partitioned, and the rendezvous are down, the members will discover each other. At that point, they add each other to their potential parent lists and continue to follow the parent discovery algorithm, with the result that a single root will emerge.

Parent Selection

Having built up a list of potential parents, each member must determine which would be the best parent and stand-bys. The basic approach is for each member, over time, to query entries in their list and determine the appropriateness of each entry. How aggressively this is done depends on the volume of content and importance of a good-quality tree. The order in which it is done may be based on domain name commonality.

I assume too that any information learned about another member that is likely to remain stable over time (such as the distance to the member or the member's transmission capacity) can be stored locally in a database and used with multiple groups. This would be particularly useful in an enterprise environment where a member may encounter the same other members over and over.

All other things being equal, members should pick parents that are nearby, by which I mainly mean a small latency. If each member choses, among the other members that may validly be a parent, a nearby one, then the overall cost of the tree is kept low. This improves performance for the members while reducing load on the internet.

(Having said this, I acknowledge that there are types of applications, particularly those that have a hard upper-bound on end-to-end latency, that may require some kind of tree-wide optimization. This sort of tree is an issue for further study.)

There may also be other factors to consider in choosing a parent than just distance. Foremost among these is whether the parent has or is actively receiving content needed by the member. By way of explanation, content can be divided into three types, what I call *buckets*, *spigots*, and *kegs*. A bucket is a finite object, such as a file, whereby it doesn't matter to the application the order in which the contents of the object are received. The application does nothing until the entire object is received.

A spigot is a content "stream" with no specified beginning or end. A member joining the tree receives the contents of the spigot at whatever point it is being transmitted. The contents are received in the order they are sent (with the caveat that some applications only require best effort delivery).

A keg, like a bucket, is a finite object. Like a spigot, however, the object should be received in order from beginning to end. An example of a keg is a realNetworks file, whereby some portion of the file from the beginning is buffered, after which the file "plays" from the beginning while more is received.

Streams are not much of an issue when considering which parent to join. Every member will be receiving and transmitting from roughly the same place in the stream. With some care, we can make this happen with buckets as well. Because the application doesn't care in what order the content arrives, each member can arrange to transmit from roughly the same point in the bucket (file) it is actively receiving. This allows members to move anywhere in the tree and be able to pick up roughly where they left off (see Figure /refbucketEx1).

Kegs are more difficult. Because each member must receive from the beginning, members that have received more of the keg cannot select as parent members that have received less. It can easily be the

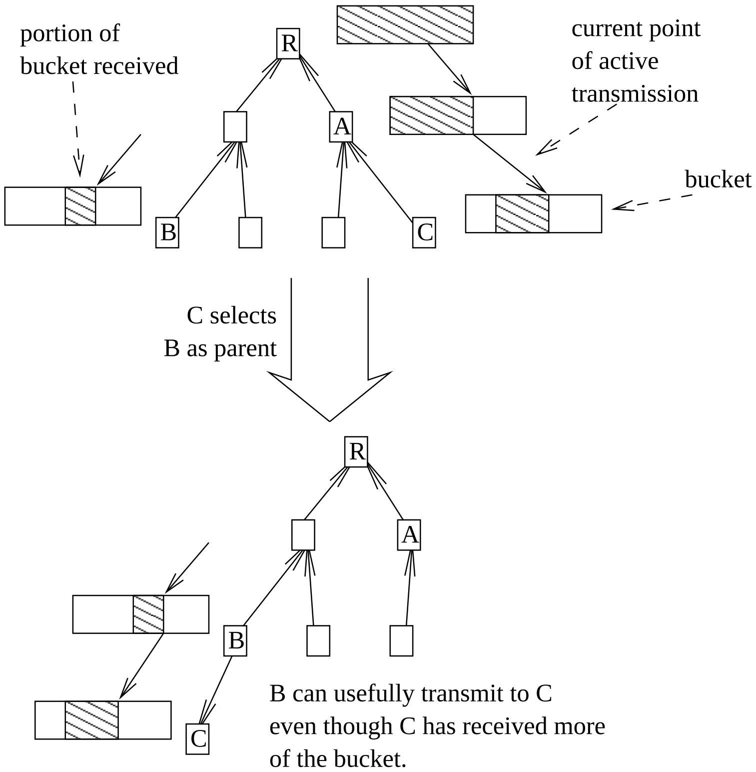

Figure 9: Example of Bucket Distribution Topology Flexibility

case that there are no such members with spare capacity for new children in a tree.

To see this, consider a tree where each member may have no more than two children. Consider a parent member P and its children C1 and C2 that high up in the tree (and therefore far along in the keg relative to lower members). Assume further that all other members in the tree that are as far or further along as C1 and C2 have no spare capacity for new children.

Now assume that P quits the tree. This leaves one "vacancy" in the tree because P's former parent now has capacity for one child. Assume that C1 attaches to P's former parent. C2 has no free member to join.

There is no real easy solution to this. It seems that C2 must find a parent/child pair of members such that the parent has received as much or more, and the child has received as much or less. C2 must displace the child of this pair by joining the parent and having the parent reject the child. The child, then, must do the same thing as C2, leading to a kind of domino effect as a series of members, each holding less of the keg, displace other members. (None of the details of how this would really go have been worked out.)

(Note that buckets can be distributed one bucket per group or multiple buckets per group. A good example of the latter would be netnews, with one yoid group for one netnews group, and with each message being a single bucket. Here even the buckets could be transmitted out of order. This could be called a bucket brigade. A group with multiple kegs is called, or at least it was back in my undergraduate days, a kegger.)

3.6 Security Architecture Overview

This section gives only a scant overview of security in yoid. To be honest, I haven't given this area a great deal of thought, and in any event am not an expert in security. Clearly this area needs more work.

For yoid security, we are interested in the following:

1. Preventing unauthorized members from joining a group.
2. Preventing unauthorized members from reading content.
3. Preventing any member from modifying content forwarded over the tree-mesh.

The first two items are probably somewhat easier than in IP multicast because the rendezvous can act as a point of access control and key distribution. For instance, the rendezvous could create a shared key for the group, and distribute these to joining members when they first contact the rendezvous. Members would not allow any host without the shared key to attach to them. Or, for something stronger but less scalable, members could always contact the rendezvous before allowing another member to become its child.

The third item is somewhat harder than in IP multicast. Assuming a single group key for content encryption, an authorized member attached to the tree-mesh can read and modify any sender's content before passing it on. To protect against this, it is necessary to use asymmetrical keys, with each sender holding a private key for itself, and all members holding a shared private key for the group (as well as the public keys for all senders and the group). The rendezvous would also have its own private key, which could be used to distribute senders' keys via the tree-mesh.

Content would be encrypted with the group's shared symmetric key, and the signature of the content would be encrypted with the sender's private key.

For more casual applications, such as a public chat group, where anybody can be a member, and where members cannot really be expected to use encryption, we need a way to prevent easy modification of content. Observing that a rogue member may by and large only modify content for those members "behind it" relative to the sender, one approach may be to have members transmit signatures of received content to their mesh neighbors. This would at least allow detection of modified content, though it might be quite hard to locate the rogue member.

3.7 API

I envision an API consisting of the following basic calls:

- StartRendezvous and StopRendezvous
- JoinTree and QuitTree
- SendToTree and ReceiveFromTree

Compared to the IP multicast API, the StartRendezvous and StopRendezvous calls are new. In addition, each of the calls will have significantly more parameters associated with them compared to the IP multicast API. This reflects the broader capability of yoid. For instance, one will be able to set a number of parameters related to the tree in the StartRendezvous call, including:

- bucket, spigot, or keg transmission styles
- maximum transmission rate

- minimum receive rate (below which frames will be discarded)
- transit buffer size
- maximum time in buffer (after which frames will be discarded)
- yoid transport protocol (yTCP, yRTP, etc.)
- average or maximum fanouts
- maximum number of sending members
- maximum number of members

and so on.

The StartRendezvous and JoinRendezvous calls take a Group ID (rendezvous domain name, rendezvous UDP port, tree name) as their primary calling parameters, and return a yoid socket number. This in turn can be used with other calls.

The StartRendezvous call does not necessarily require that the host on which the call is made be the rendezvous host named by the rendezvous domain name. Any host with a working domain name may offer a rendezvous service to other hosts. This allows hosts with no working domain name, or that may only be connected for a short time, to still create a tree.

Note that such a rendezvous service could work without requiring a StartRendezvous at all. Rather, it could treat the first join it receives for a given tree name as a command to start a rendezvous for that tree, though in this case it would require some alternative means of determining the tree parameters.

The API could be made available on a host as part of the application (a compiled library), as a daemon running on the host, or as part of the kernel.

Additional Calls

In addition to the above basic calls, a pair of calls are needed to cause unicast routing information back to the calling member to be installed: installRouting, flushRouting.

For the purpose of diagnostics, it may be useful to have a call that returns the tree or mesh neighbors or root path of the local node (or perhaps even of remote nodes): getNeighbors, getRootPath.

There will be other calls as well, especially as yoid grows in functionality and flexibility.

4 Odds and Ends

This section presents a selection of possible enhancements to yoid. Some solve specific problems while others add new capabilities.

4.1 Neighbor Aliveness

One problem endemic to yoid that you won't find to nearly the same degree in IP multicast is that of detecting neighbor aliveness. In IP multicast (in either its native or tunneled modes), each router has a relatively small number of neighbors, independent of the number of multicast groups. The neighbors are relatively nearby, often on the same wire. Furthermore, routers are "predictable" boxes—they can be expected to answer a ping in a certain amount of time with high probability. As a result, each router can afford to ping its neighbors frequently (say one ping per second), and can quickly deduce that a router is no longer alive with a high probability of correctness.

Not so with yoid. Here, any given member has several neighbors for every tree it has joined. It is easy to imagine a typical host joining hundreds of trees, each with only sporadic transmissions. For instance, one tree for each stock of interest, one for each mailing list of interest, one for each news source of interest, and so on. One ping per second, say, across several hundred neighbors amounts to a lot of overhead.

The problem is made worse by the fact that it is harder to determine aliveness for a typical desktop PC than for a router. A typical PC may suddenly become overloaded (for instance because a new application is being loaded), and may disappear for seconds at a time. Mistaking a busy neighbor for a down neighbor is costly. As a result, conservatively speaking, it can take on the order of half a minute to determine that a down neighbor is in fact down. While this is going on, the tree is partitioned.

We're looking for two things here. First, a way to do neighbor aliveness without generating huge volumes of traffic (relative to the amount of application data over the tree). Second, we want a way of dealing with the fact that neighbor down detection can take a long time.

Before getting into possible solutions, I want to first point out that at least for one class of application—namely file distribution—aliveness detection (or lack thereof) isn't as much of a problem, for two reasons. First, transmission to/from each neighbor is continuous—either the file has been fully received/sent or the neighbor should be actively sending/receiving it. Lack of activity works as a signal that the neighbor is unavailable. Explicit pinging is therefore not necessary and so there is no extra overhead.

Second, order-minute lapses in transmission are, for many applications, not a serious problem (provided they are relatively infrequent). An example would be download of a large software package. The download takes many minutes in any event, so an occasional lapse in transmission is certainly not going to break the application and may not even be noticed. Even streaming audio or video, provided that enough of the stream is locally buffered, can survive lapses in transmission.

In what follows I outline several approaches, that can be used alone or in combination, for attacking the following two scenarios. In both scenarios, application traffic is sporadic (not a steady predictable stream) and possibly infrequent. In the first scenario, occasional order-minute lapses are acceptable to the application, so we're primarily interested in reducing ping overhead. An example here would be mail distribution for a typical mailing list. In the second scenario, lapses are not acceptable—application data must be received very shortly after it is transmitted or the application breaks. An example here would be stock quotes for an active trader or certain internet games (understanding here that most games probably involve a pretty steady stream of traffic). I call the first scenario time-relaxed, and the second time-critical.

(The reader might be thinking that nobody in their right mind would use a network consisting of users' desktop PCs for distribution of time-sensitive stock information. I don't disagree, but to that I would repeat that one can always engineer a yoid infrastructure to look, topologically, like an IP or server infrastructure, using dedicated, reliable, and well-placed servers for yoid distribution. In other words, throwing money at the problem is a possible valid approach.)

Broadcast

The simplest approach is to simply distribute all application data using mesh broadcast. Because mesh broadcast can still work if one or more mesh neighbors go down, one can get away with pinging mesh neighbors less frequently than tree neighbors. Required ping frequency varies depending on the number of mesh neighbors, the probability that they will disconnect without notification, and the desired robustness of the application. If the volume of application data is similar or less than the ping volume, then simply doing mesh broadcast is a win. This approach is good for the time-critical scenario as well as the time-relaxed scenario.

Use the Mesh for Member Down Notification

Another approach is to use mesh broadcast not to deliver content per se, but rather to quickly inform members when a node is detected as non-responding. This is described in Section 3.3 under the heading "YDP Sequence Numbering". The process of detecting a node is down, broadcasting a notification about it, and having the recipients "turn on" alternative senders obviously takes time, but on the order of a few seconds, not half a minute. This means it should be adequate for many time-critical applications.

Aliveness Buddies

Both use-the-mesh approaches still generate per-tree ping traffic, which while less than a non-mesh approach still may amount to significant amounts of volume. An approach that is less sensitive to the number of trees a host may be attached to is the aliveness buddy approach. Here, each host selects a small number of other hosts (three or four), not necessarily among its current tree neighbors, as aliveness buddies. These hosts should be as nearby as possible.

Aliveness buddies ping each other continuously and can determine relatively quickly when each other has gone down. They also tell each other who all their current neighbors (tree and mesh) are. When a host detects that its aliveness buddy has gone away, it tells all of the buddy's neighbors. A host with a very large number of neighbors (because it is attached to a very large number of trees) could have more aliveness buddies, and could tell each of a portion of its neighbors.

This approach is not as useful for the time-critical scenarios, because a host, even if it pings frequently, must still wait a significant amount of time to insure that its buddy is really down.

4.2 Configuration with Yoid Proxy Servers

Section 3.4 only described YTMP endhost tree management. In other words, it assumed that each endhost knows through local means (i.e., the JoinTree API command) that it wants to join a given group, and that the endhost would serve as a transit member.

For various reasons, there are certainly cases whereby an endhost should not be a transmit member, but instead should use a nearby *yoid proxy server* instead (see Section 3.2). This is a host that joins the group on behalf of the endhost, acts as a transit member of the tree-mesh, and forwards all content between the endhost and the group. The reasons include performance (the endhost may simply not have the bandwidth or horsepower to act as a transit), reliability (the application may require highly reliable transit members), and security.

Configuring with a proxy server is not necessarily as straight-forward as one might imagine. The main sticking points are:

1. how does the endhost know when it needs to use a proxy server, and

2. how does the endhost know which, of possibly several proxy servers, to choose from?

Concerning the first item, it may be possible in many but certainly not all cases for the rendezvous to know that a proxy should be used (for instance, because the application starting the rendezvous knows, or because the rendezvous has some policy database driving it). In these cases the rendezvous can tell the endhost to use a proxy at first contact. It may also be possible for the endhost itself to monitor its own performance and itself determine when a proxy is needed, but this is non trivial.

Concerning the second item, there may be different proxies to use in different situations. At a minimum, it is easy to imagine one set of proxies for high-bandwidth realtime groups, and another set of proxies for large file distribution groups, as these two have markedly different performance criteria. There may be other reasons for choosing a specific proxy over others, for instance security reasons.

As for proxy discovery, perhaps a good approach would be to have a well-known group name, such as

`yoid://well-known-yoid.local.domain.name/proxy-discovery`.

This group could distribute information about proxies in the local domain, including some kind of policy information. Endhosts could stay attached to this group, and over time discover the nearest proxy servers of each type and the policy for when to use them.

Once an endhost determines that it should use a proxy, the protocol for coordinating with the proxy should be straight-forward.

4.3 Web Caching (and Hierarchically Nested Groups)

One cannot talk about host-based multicast without talking about web caching infrastructures which are, after all, a form of host-based multicast. This leads naturally to the question of yoid's applicability to web caching.

There are several aspects of web caching I would like to consider:

1. Finding the nearest web cache (cache discovery).
2. Finding a web cache (among many) that is most likely to already hold the desired content (cache routing).
3. Pre-loading web caches with content that is most likely to be desired (cache pre-loading).

The first is rather straightforward with yoid—simply create a well-known group of web caches, and have web clients contact the group and over time converge on the nearest web caches. (Note that the client would not join per se. Rather, it would contact and "explore" the group, same as a true joining member does, but not actually join.) But there are also other ways to do nearby web cache discovery, so I'm not sure that this by itself is very interesting.

The second and third aspects are important areas where yoid can perhaps make a valuable contribution. To do so, however, some additional mechanisms are required for yoid. Namely, it must be possible to 1) organize a yoid "group" into a nested hierarchy of subgroups, and 2) be able to route unicast frames through the hierarchy to a specific subgroup (or subsubgroup, subsubsubgroup, and so on). I call a group with these two characteristics a *nested group*.

Nested Groups

In what follows, I talk about nested groups outside the context of web caching. After that I'll bring web caching back into it and show how nested groups can be used for cache routing and cache pre-loading.

Figure 10 illustrates the major components of a yoid nested group. The name of the group in the figure is `yoid://rn1.com/gn1`. The figure illustrates three levels of nesting. Each oval represents a

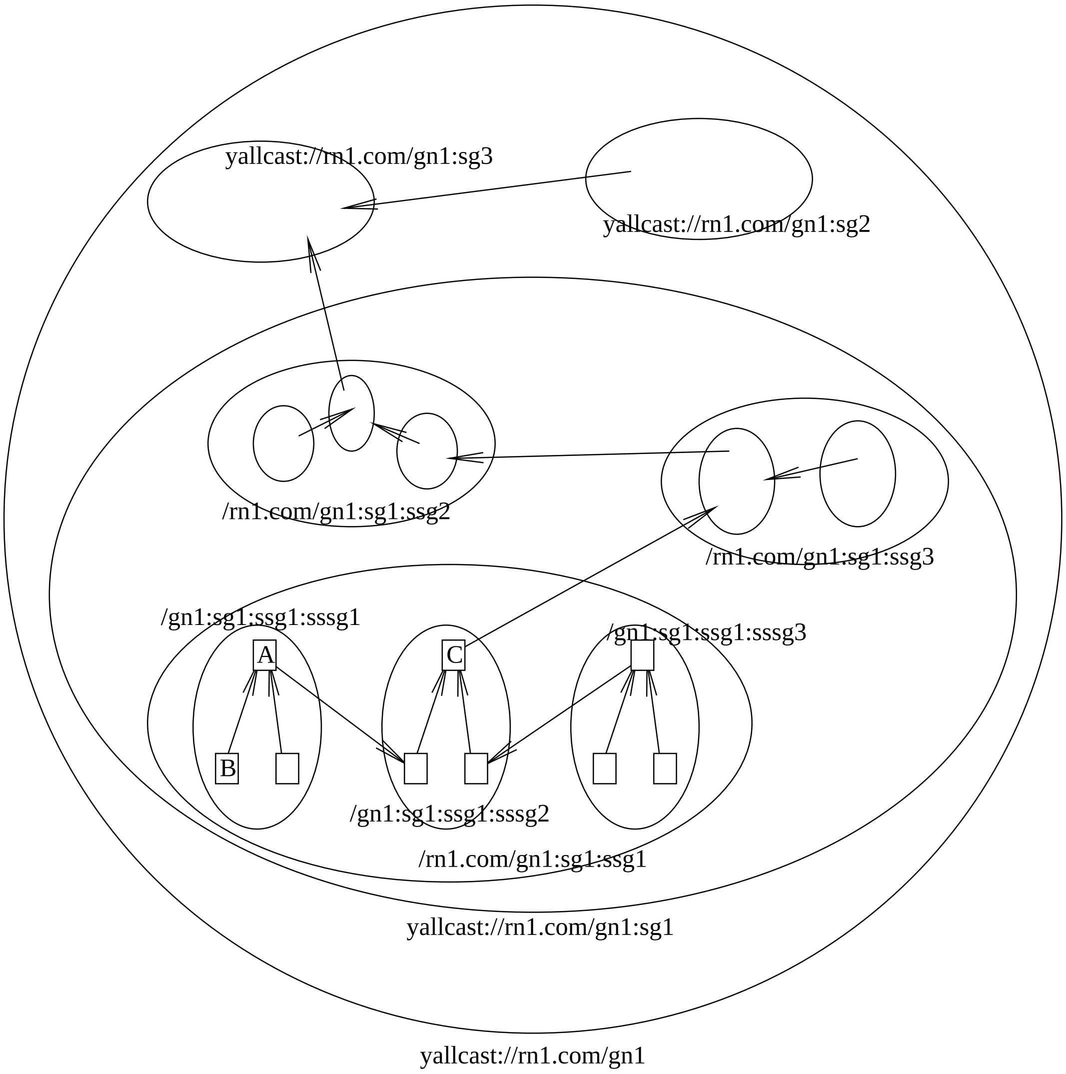

Figure 10: Yoid Nested Group

nested subgroup. Each subgroup is internally connected, by which I mean that a frame multicast to the subgroup only needs to traverse members of the subgroup.

I have chosen a naming convention whereby the group id of each subgroup is a superset of that of the subgroup it is nested in. I have arbitrarily selected ':' as a delimiter. It is not necessarily the case that a delimiter is needed, as long as within the protocol there is a way to indicate which subgroup given named subgroup is nested in. For that matter, it is not necessarily the case that a subgroup id be a superset of that of its parent subgroup, again as long as the protocol can make the identification. It is probably a good idea, though, for the sake of the humans that have to read the strings, to have these kinds of conventions. (To avoid too much clutter in the picture, I have truncated the beginning of the group id in the lower subgroups.)

I have chosen to draw this picture as a series of balloons inside balloons, rather than as a branching tree.

This is intentional, and is meant to emphasize the point that a member of a subgroup is also a member of all the subgroups above it. What this means practically is that if a frame is multicast to a certain subgroup S, it will reach all members of all subgroups nested within S, however deep.

(This will all look, by the way, tediously familiar to anybody versed in standard hierarchical routing algorithms. Such a person will also know that there are a number of ways to structure, name, and route through hierarchies. I do not explore alternatives here, have not yet thought deeply about them, and as it is do a lot of hand waving with what I do say here. This is all an area for further study.)

To build the nested structure shown in Figure 10, each member simply joins its lowest subgroup as it would any (non-nested) group, with the following exceptions:

1. The root of a subgroup joins the next higher level subgroup.
2. A few members of a subgroup must establish mesh neighbors in other subgroups at the same level.
3. There is no rendezvous for subgroups. Rather, members of a given subgroup are discovered using one of two alternative methods described below.

For example, member B shown in the figure would join subgroup `rn1.com/gn1:sg1:ssg1:sssg1` in the normal way, by which I mean it would only select members in the subgroup as a parent. One member of `rn1.com/gn1:sg1:ssg1:sssg1` finds itself to be the root (A in the figure), and so joins `rn1.com/gn1:sg1:ssg1`. This means that it chooses its parent from the set of all members of all subgroups of `rn1.com/gn1:sg1:ssg1`. The root of `rn1.com/gn1:sg1:ssg1` (member C) selects a parent from `rn1.com/gn1:sg1`, and so on.

Required, both by applications and by YTMP itself, is a way to efficiently discover a member of any subgroup. The normal method of discovering a member of a regular (non-nested) group is via the rendezvous. In the case of a nested group, the rendezvous knows of a few (or more) members of the group, but it doesn't necessarily know of members from each subgroup. This means that there must be a way to route from any member of the group to any subgroup.

Though endless variations are possible, the basic method for doing this would be to require each member to know of one or two members of every sibling subgroup at each level. In other words, A would know of one or two members each from `rn1.com/gn1:sg1:ssg1:sssg2` and `rn1.com/gn1:sg1:ssg1:sssg3`, one or two each from `rn1.com/gn1:sg1:ssg2` and `rn1.com/gn1:sg1:ssg3`, and one or two each from `rn1.com/gn1:sg2` and `rn1.com/gn1:sg3`.

A simple way to propagate this information would be for every member to broadcast a control message about itself to all of its mesh neighbors. They would store the contained subgroup information if either they didn't already have it, or if it were from a more nearby member, where nearby here could be measured in terms of hops over the tree or hops over the mesh (versus say latency, which might be better but more costly to measure). If they stored the information, they would also then forward it to their mesh neighbors.

An example of how this would allow discovery is as follows. Say a member X (not shown in the figure) in `rn1.com/gn1:sg3:ssg2` wanted to discover a member in `rn1.com/gn1:sg1:ssg1:sssg3`. (Note that it isn't necessary for all members to be at the same depth in the hierarchy.) By virtue of its place in the hierarchy, X would know of a couple arbitrarily selected members in `rn1.com/gn1:sg1` (a sibling subgroup at the highest level). Say one of those members were in `rn1.com/gn1:sg1:ssg2:sssg1`. This member would in turn know of a member of sibling subgroup `rn1.com/gn1:sg1:ssg1`, say one in `rn1.com/gn1:sg1:ssg1:sssg1`. This one, finally, would know of one in `rn1.com/gn1:sg1:ssg1:sssg3`. Thus, with three queries (more if one of the members was unavailable, possibly less if we were lucky), an appropriate member can be discovered.

Note that this method would give each member the required information about other members, but not a connection to each said member. As a result, discovery could be relatively slow—a connection would first have to be established at each hop of discovery along the way. An alternative would be to actually pass routing information through the tree-mesh, so that a frame could be unicast forwarded hop-by-hop through the tree-mesh to any addressed point in the hierarchy. In other words, employ a pretty much standard hierarchical routing algorithm to support fast discovery.

To be clear, I am not, I repeat not, suggesting an alternative to unicast IP (or IPv6). (Though I suppose there might be applicability to VPNs.) I am simply exploring possible fast methods of discovery in a nested group, the importance of which will be clear when I talk about web caching.

To complete this sketch, I need to talk about one more thing—how a member decides what subgroup it should join (or create if it is the first to join). The short explanation is that the application tells it. For the application to do this, however, it very likely needs information about what subgroups already exist (the web caching application certainly does).

Thus, the API between the application and yoid requires a mechanism whereby the application can query yoid and ask what the set of child subgroups are of any named subgroup at any level. Yoid in turn needs a protocol whereby any member can make this query any other member. So, for instance, the application in the member at `rn1.com/gn1:sg3:ssg2` could ask over the API about all subgroups of `rn1.com/gn1:sg1:ssg1`. The yoid at `rn1.com/gn1:sg3:ssg2` would discover a member in `rn1.com/gn1:sg1:ssg1`, query it for all child subgroups (which it would already know by virtue of the information needed for discovery), get the reply, and return that to its local application.

Web Cache Routing and Pre-loading

Now I can describe how nested groups can be used in support of web caching.

First, assume a coordinated web cache infrastructure—a set of machines deployed around the global internet solely for the purpose of web caching. These caches would all join the same nested group. The sub-grouping scheme would be known in advance by the caches. Specifically, each cache would know the naming convention, including how many subgroups each subgroup has (i.e., the subgroup fan-out), and the maximum depth of the hierarchy.

At initial join time, each cache would randomly select a bottom level subgroup and join it. It may be the only member of that subgroup—this is not a problem. There will be many "holes" in the hierarchy—potential deep subgroups with no members at all. This is also not a problem. Top-level subgroups will be well-populated, getting thinner as you go down the hierarchy (how thin how quickly depends on the fan-out and number of caches).

Web clients can also "join" the group, but only for the purpose of discovering nearby web caches (if this is necessary at all—as I said, other mechanisms also exist for this).

When a web client wants to get a page, it sends the URL to its cache (proxy) as usual. The get is then routed through the hierarchy as follows: At each level, starting at the top, the cache hashes the URL into the space of populated subgroups at that level, pseudo-randomly but deterministically selecting one. The get is forwarded to a member in that subgroup, which in turns hashes and selects from the subgroups at the next lower level.

This continues until either 1) there is a cache hit, 2) the lowest level subgroup is reached, or 3) some other criteria, such as the number of caches traversed, is met. If there is no hit, the get is forwarded to the web server, the page is retrieved, cached, forwarded back to the original proxy, and finally back to the client.

Where the page gets cached at this point is a good question. It could be cached nowhere, at the target cache only (the cache where it was ultimately routed and fetched), at the original client proxy cache only, at the target and proxy caches, or at all the caches traversed in routing the get. The reason we wouldn't necessarily want to cache it everywhere, and even might not want to cache it anywhere, is that by and large it is preferable to use cache pre-loading rather than have to deal with potential stale caches.

Cache pre-loading would work like this. Caches keep track of how many gets they get for each URL (even if they don't actually cache the page). When a cache notices that it is getting a lot of gets for a given URL, it invokes cache pre-loading for that URL. This means that it 1) maintains an update copy of the page, and 2) multicasts the page to some low-level subgroup.

Ideally the web server for the page would get involved and update the cache whenever the page changed. Barring this, the cache could just be real aggressive about keeping it fresh. Whenever the page changes, an update of the page would be multicast to the subgroup. This would have the effect of spreading the document over more caches, but without the stale cache problem (or, at least, much less of one).

If after invoking cache pre-loading at a low level individual caches are still getting a lot of queries for that URL, pre-loading can be expanded by multicasting the page and any subsequent changes to the next higher level subgroup, and so on.

Note that the addition or deletion of individual cache servers can modify the set of populated subgroups. This in turns changes where some URLs get hashed—URLs that formerly hashed into a given low-level subgroup may now instead hash into the new subgroup created by the new cache server. This is not a particular problem in this case, for two reasons.

First, caching is from the word go a hit-and-miss thing. A URL being routed to a new cache server looks, from the client's perspective, no different from a cache miss, and from the replaced cache server's perspective, no different from a lack of new queries.

Second, assuming a reasonable number of cache servers to begin with (some tens if not hundreds), the addition or deletion of a single cache server only changes the set of populated subgroups at the lowest levels (with high probability). Higher level subgroups, where popular pages get pre-loaded, will rarely change.

4.4 Meta-Rendezvous Service (Another Nested Groups Application)

The rendezvous node is in the critical path of a working yoid group. This creates several problems. First, many applications that want to create a group will have no domain name of their own (because they are on dial-up hosts), and so cannot start a rendezvous. Second, there may be long-term groups (such as a public chat group) whereby the membership changes over time, with no single member always available to be the rendezvous. Third, even if there is such a host, it may crash, lose connectivity, etc.

This all points to the need for a globally available robust meta-rendezvous service. Such a service could be at a well-known domain name, for instance `rendezvous-service.yoid.net`. Group ids would look like:

```
yoid://rendezvous-service.yoid.net/category/topic/groupName
```

or

```
yoid://rendezvous-service.yoid.net/user-handle/groupName
```

Not surprisingly, the service could be based on a nested group, similar to the web cache application. In this case, there would be dedicated rendezvous servers populated around the globe, and they would pseudo-randomly establish subgroups according to a pre-established scheme. Rendezvous servers for a group would be discovered by repetitive hashing of the group id into the population of subgroups at increasingly lower levels.

However, instead of stopping at the lowest possible level (as was the case with the web cache application), it would stop at the lowest-level subgroup that had more than some minimum number of rendezvous servers (say 3). This is for robustness. All of the rendezvous servers in the subgroup would act as a rendezvous for the group. Each of the rendezvous would multicast its list of group members to the others.

Applications using the service would explore the rendezvous service group to find the nearest servers. When an application wanted to create a group, it would choose a group name (and other info such as a description of the group) and submit it to a nearby rendezvous server. This and subsequent servers would route it towards the target subgroup. If a group with the same group id already existed, this would be reported back to the application.

Otherwise, the selected rendezvous server or servers would establish themselves as the rendezvous for the new group. Other joining members, even starting from different rendezvous servers, would be routed to the same subgroup and find the appropriate rendezvous servers.

If the group turned out to be very popular, the number of rendezvous servers could be expanded in the same way web cache pre-loading is expanded. That is, the database for the rendezvous server would be multicast to all rendezvous servers in the next higher level subgroup, allowing them to share the load between them.

This expansion would also happen if some of the rendezvous servers in the lowest-level group crashed, in order to maintain robust operation.

4.5 Distributing Across a Lot of Time

As already stated (many times), yoid distributes across time as well as across space. Distributing across time, however, can require certain new coordination mechanisms—particularly for endhost-based distribution.

To get the benefits of "multicast", each consumer of content must be expected to transmit on average, as much as it receives. For a group with spigot content, at any given time, some members will be transmitting more than they receive, some less, and some the same, depending on whether they have more then one, zero, or one tree neighbor respectively. For instance, on a perfectly balanced tree with a fanout of 2, half the members will transmit to two neighbors, and half to none. On average though, transmission equals reception.

The difficulty arises with bucket and keg content (file distribution). Here it will typically not be the case that a member transmits as much as or more than it receives *during the time it is actively receiving.* For popular content, where there are multiple simultaneous receivers, typically a member will be part way into its reception before it is asked to start transmitting to another member. For less popular content, it will often be the case that a member receives all the content without being asked to transmit it at all.

As a result, if we want to get the benefits of yoid, members of bucket or keg content must be expected to stay attached to the tree for some period of time after they themselves have received the full content. This presents a number of questions. Is a member always expected to stay attached after it has received the content? If not, what are the criteria for deciding to stay attached? How long is a member expected to stay attached? If a member disconnects from the internet before it has transmitted as much as it received, is it expected to rejoin the group later until it does transmit?

There are no easy answers here, just various engineering and policy trade-offs. For instance, since there is overhead involved in keeping a group even when no content is being transmitted, clearly it is a lose to keep a group for very infrequently accessed content. The cost of keeping the tree at some point exceeds the cost of always transmitting the content from the original source.

Even if the cost of keeping the tree is less than the cost of transmitting the content, endhosts may certainly object to having to stay attached to a lot of groups. For instance, imagine an endhost that receives 1000 yoid buckets per day, with each bucket retrieved on average one per hour. Such an endhost would be expected to store content for and stay attached to about 40 groups at any given time.

An alternative to this would be to require each endhost to stay attached to some fraction of groups it joins, but to transmit proportionately more for each group. For instance, the endhost could be expected to stay attached to 10% of the groups it joins, but until it has transmitted 10 times as much as it received for each group.

I don't want to belabor this any more—you get the idea. Obviously a lot of research is needed here.

It may work out that doing bucket or keg yoid distribution over the public internet is just not tenable, and that some sort of cache server infrastructure is more appropriate. Doing endhost-based bucket or keg yoid distribution in an enterprise setting strikes me as more palatable, given the administrative commonality across endhosts and the control of those endhosts by IT.