Implementing Overcast: A High-Level Multicast Protocol

Raffi Krikorian
MIT Center for Bits and Atoms
raffik@media.mit.edu

Michael Craig, Tucker Sylvestro
MIT
{mic, tsyl1}@mit.edu

December 6, 2002

Abstract

Overcast [1] is a system providing high-level multicasting features on top of a TCP/IP network. The goal of Overcast is to efficiently and reliably transport bandwidth-intensive streams of data from a single source to potentially very many clients, beyond what a traditional, single-host (or even clustered) data server could support. This paper describes one design and implementation of Overcast, and outlines future work to make the implemented system more robust and powerful.

1 Introduction

Overcast [1] is a software system which links together a number of trusted computers spread across the Internet to provide data to potentially very many clients. It is designed to allow for better bandwidth allocation when downloading content than a centralized distribution model.

The concept of data distribution has become extremely common. The basic model is that there is one computer that wants a certain piece of data which is stored on another computer somewhere else. The first computer, called a "client" requests the data from the other computer called the "server" and the data is transferred over a network connection between the two. In practice, this data can be something as simple as a fixed-length file (ex: a web page), or as complicated as a live video feed in which data is sent to the client until an arbitrary point in the future. In the former case, the data being transferred is typically small enough in size that a single computer with a relatively low-bandwidth (e.g. 10Mb/s) network connection can serve hundreds of clients concurrently without being unreasonably slow. However, many modern applications such as streaming audio or video and multi-Gigabyte file downloads require a great deal of bandwidth to transfer from one computer to another. This means that a single computer responsible for serving such data either has very high bandwidth requirements or cannot serve many clients at once.

In the traditional client-server model for data distribution, one server allows many clients to connect to it and sends the data directly to each one. If many clients need to be supported, or if the data being served are particularly large, then there will be a burden on the server's network link. It is often the case that people must choose to sacrifice quality (e.g. use a more lossy-compression scheme on a video stream) or support less clients than they would like to due to the limits of the server's bandwidth and the high costs involved with increasing it. This is where Overcast comes in.

Overcast solves the problem of serving bandwidth-intensive content by building a tree of trusted servers with the original data sources at the root node of the tree (see figure 1). Each node in the tree is able to serve a number of clients. The root sends the data to its children, each of whom send the data on to their own children, and so on until every node has a copy of the data that needs to be served. Clients connect to the root node and are redirected to a node in the tree from which they can get the data. An Overcast tree thus transparently supports far more clients than the a single server alone could. Bandwidth usage at the root is greatly reduced since it only needs to send the data to its

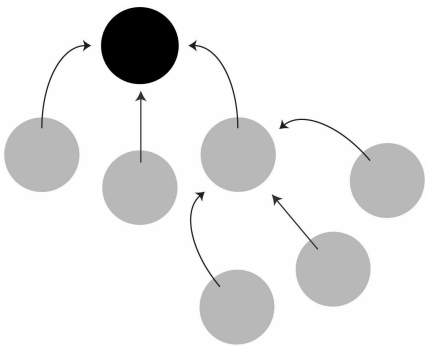

Figure 1: An Overcast network with the black circle representing the content source/root node. All the other nodes are connected in a tree with the information flowing back through the edges. Clients connect to the black/root node and are redirected to one of the grey/child nodes to download content.

child nodes rather than every client, and the total bandwidth available is theoretically the aggregate of all the nodes' bandwidth. An Overcast network of ten nodes might support 100 clients by giving ten to each node, whereas in the traditional model, the root node must serve all 100 clients by itself.

We built an implementation of Overcast in Java which supports standard HTTP clients. Our system uses the tree-building protocol described in Overcast[1] to dynamically build a data distribution network. Clients can then connect to the root, request data and be redirected to an appropriate node. For purposes of testing and demonstration, our implementation is designed to run on MIT's Project Athena [2] platform, but it should work on any environment which supports Java 1.4.

2 Background

Overcast was proposed by Jannotti et al. in "Overcast: Reliable Multicasting with an Overlay Network" [1], but details of the implementation were not discussed. Overcast mimics the well-known idea of IP Multicasting, but at a higher level: Overcast runs in the application layer on top of TCP/IP, and can ignore the low-level details of the networking protocols. Thus we call Overcast the "overlay network" while the underlying IP level is the "substrate network." For a number of reasons (commercial, political, and otherwise), IP Multicasting is not a practical solution on the Internet today. Nevertheless, Overcast extracts much of its high-level design from this idea, although its traffic only flows in one direction (from a single root node out to many clients).

2.1 Previous Work

The original Overcast paper [1] discusses a number of similar overlay networks. Among these are End System Multicast [3], Yoid [4], and X-bone [5]. None of these quite matches the goals of Overcast: End System Multicast provides true multicasting, at a high level, for small groups; Yoid and X-bone are more general purpose in scope, as Overcast's intended uses are rather limited in scope (at the present time, anyhow).

A number of other, similar open-source systems have been developed, such as the popular SHOUTcast. This system also differs from Overcast, however: it is designed strictly for MP3 audio streams and uses a statically-constructed tree.

3 High-Level Design

The high-level structure of an Overcast network is simple enough: a tree of trusted nodes (as seen in figure 1) is spread across the Internet, each receiving data generated by or stored at the root node. HTTP clients connect to the root node and are redirected to a node in the tree from which they receive data. The details of how this tree is built and maintained, and how clients are redirected, is described below.

3.1 Building the Trusted Tree

By using an innovative tree-building protocol, Overcast networks are grown automatically in a dynamic, cooperative manner. In addition, the tree can change its structure over time in response to node failures and changing network conditions, all without requiring human intervention. Similarly, node failures are dealt with almost instantaneously by reconnecting orphaned nodes to ancestor nodes as soon as a communication error occurs.

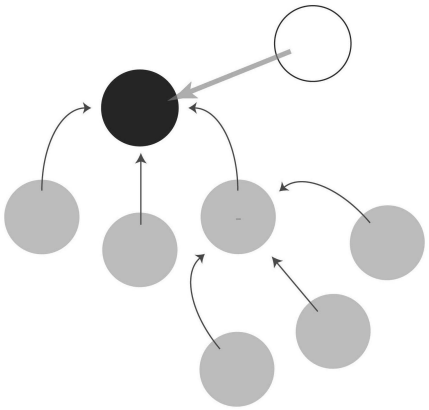

Figure 2: A new node first contacts the root node and performs a bandwidth test against it.

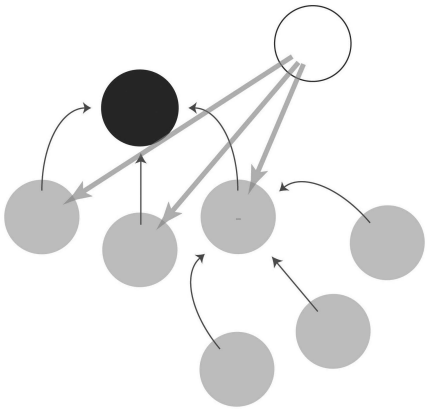

Figure 3: After performing a bandwidth test against the root node (see figure 2), the root node performs bandwidth tests against the root via all its children. It then chooses the fastest child node ...

Initially, the Overcast tree consists of a single root node. In the process of joining the network, a new node uses a bandwidth-measuring protocol to determine a "good" leaf position for itself in the tree. Here, "good" is defined in terms of maximization of bandwidth from the root to the other nodes, through the tree, and the algorithm for building the tree approximates such a "good" structure for the it.

A new node initially contacts the root node (figure 2) and measures the bandwidth between them. It then measures bandwidth from the root node *through* each of the root's existing children (figure 3). If the difference in bandwidth through any children is less than 10% of the bandwidth to the root, those children are considered better choices as potential parents for the new node. The new node picks the best (in terms of bandwidth through each) of these children, and tests bandwidth through *its* children in the same manner (figure 4), and so on down the tree. The new node thus stops at a point in the tree where moving down a level would result in more than a 10% bandwidth drop (figure 5). The reason for pushing away new nodes as far from the root as possible is that direct bandwidth from the root to its children should be maximized while preventing the unnecessary usage of bandwidth at any specific node. We only attach the new node to a non-leaf node when we know that moving it further down will result in a significant drop in bandwidth.

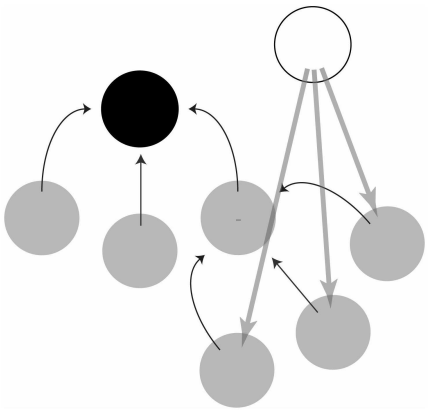

Figure 4: ... and the same procedures as dictated in figure 3 are repeated.

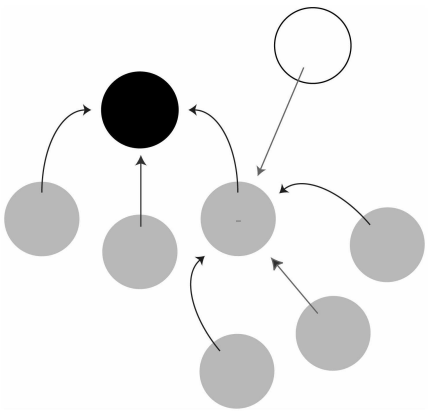

Figure 5: In this example, the children are all *not* within an acceptable speed loss, and the new node joins the network at the first level.

Consider a new node going through the algorithm described. This node tests bandwidth directly from node p, and from node p through each of its children $c_0, c_1, \ldots, c_n$. It might appear that, for our new node to make the best choice it can about whether to choose p, or some c_i, it should measure bandwidth from the *root node* through each of these choices. However, this would not help us, as p and all its children c_i share the same ancestry in the tree. Measuring the bandwidth through any links above p is unnecessary. It only makes sense for the new node to measure bandwidth through the different possibilities as a descendant of p.

Because bandwidths on various paths throughout the Internet will change over time, nodes occasionally move around in the tree to increase bandwidth between them. Periodically a node tests the bandwidths to its grandparent, its parent's siblings and its siblings and compares them to that of its parent. Of these tests, if any bandwidth is better than that of its parent by 25%, the corresponding best node will be chosen as the node's new parent[1].

[1]The reason it requires a 25% performance improvement rather than the 10% it required when it initially joined is to avoid the overhead of switching parents unless the performance has severely degraded.

3.2 Up/Down Protocol

The root node needs to keep a list of not only its children but also the entire tree below it, so that it knows where to redirect clients when they connect—it therefore needs to know when nodes join and leave the tree. This information is propagated efficiently with the "up/down protocol" in which nodes notify their parents of changes that have occurred in their subtrees, at regularly-scheduled intervals. These scheduled updates are called "keep-alives" and serve a dual purpose: by periodically checking in, a node lets its parent know that it is still functioning properly. Also, in addition to telling its parent about changes in its subtree, it receives any updates that have occurred either in global settings being propagated down from the root, or in the parent's list of ancestors.

If a child node fails to send a keep-alive by a certain time, its parent removes it from its list of children and the list of nodes in the subtree rooted at itself, then puts it in a list of dead children so that it knows to send its own parent a "death certificate" for that child during the next scheduled keep-alive. Symmetrically, whenever a node receives a keep-alive from a node it does not have in its child list, it assumes that the node has joined the tree as one of its children and adds it to its child list and subtree list. In addition, it adds a "birth certificate" for that child to its list of new children that it will send to its parent at the next scheduled keep-alive.

When a node receives a keep-alive from one of its children, it may contain a list of birth and death certificates. For each birth certificate, the parent adds that node to its internal list of the nodes in its subtree, and also adds the birth certificate to its list of new children so that it will be propagated to its parent during the next keep-alive. For each death certificate it performs the symmetric operation of removing that child from its subtree list and putting the death certificate in its list of dead children to send to its parent. However, in order to prevent sending unnecessary certificates, the node will not propagate the birth or death certificate if it did not cause its subtree to change. This would be the case if a grandchild node changed parents from one of the grandparent's children to another. Since the parent node stores its sub-

tree list as a flat array, its contents remain the same even though both children's subtree lists changed.

This simple protocol allows each node to keep track of all the nodes in its subtree using very little bandwidth. The only time tree structure information is sent between nodes is when a node has joined or left the network[2]. In addition, this protocol allows for a very simple algorithm to let nodes change parents either when they detect a better position in the tree or when their parent dies: the moving node simply stops sending keep-alives to its old parent and sends one to its new parent instead. The old parent will issue a death certificate for the node in question and the new parent will issue a birth certificate. Both will be propagated up the tree and all nodes should have the most up-to-date information in $O(\mathrm{depth} \times t)$ time[3].

One complication can occur with the node movement protocol described above. If an ancestor receives the death certificate for the node *after* it receives the corresponding birth certificate, it will think the node has actually died. The solution is for the parent node to record the number of times that that child has switched parents in the birth or death certificate it sends out for it, and receiving nodes ignore any death certificates that have a lower number than the most recently-received birth certificate for that node.

3.3 Serving Clients

When a client wishes to request a data source provided by an Overcast network, it makes a standard HTTP request to the root node with a URL identifying the data it wants. The root then uses an algorithm (see section 4.7) which selects the node it should connect to for downloading and sends the client an HTTP redirect to that node. The client follows the redirect to that node and downloads the data from it.

Overcast uses HTTP because it is a well-established, convenient protocol not only for file transfer but for certain types of "live" or continuous data streams. MP3 audio streams, for example, traditionally use HTTP. While future applications may wish to use video streams or other media that do not use HTTP, many such protocols can be simply adapted into our Overcast system. Only the code responsible for interacting with user clients needs to be changed; since data sources in our Overcast implementation already support different "types" (similar to mime-types), selecting appropriate client-handler code should not be difficult.

3.4 Using Non-Volatile Storage

One of the biggest differences between Overcast and other multicast-like distribution systems is that the tree nodes are assumed to have large amounts of non-volatile storage, allowing them to cache large amounts of data. One of the reasons for this design choice is that we assume the nodes are all running on trusted, dedicated servers rather than personal computers on the Internet. In fact, Overcast was originally developed to be run as a private network for streaming video between different offices of a single company. The use of this large cache is important for some implementation details, which are discussed further on.

3.5 Node-to-Node Communication

There are two types of connections between a parent node and a child node: one "organizational" connection, and possibly several "raw data" connections. The organizational connection is dedicated to the up/down protocol described in section 3.2. It is used at sparse intervals and for small amounts of data. The "raw data" connections are used to transfer the actual data an Overcast node is delivering to its child nodes[4]. Distinctions between these types of connections are made in the interests of a modular code-base and to simplify implementation as will be discussed in section 4

[2] Keep-alive messages are sent between nodes at regular intervals; however, keep-alives that don't contain any birth or death certificates and are only 22 bytes long in our implementation.

[3] Where depth is the max. of how far down the tree the moving node was either before or after the move, and t is the time between keep-alives

[4] Although these connections are *not* used for its clients. Clients connect on a different port and use a different protocol as described in section 3.3

4 Implementation Choices and Design

We decided on building our implementation in Java for several reasons. First, since our target platform, Project Athena, spans a variety of hardware architectures and operating systems, we wanted to make our implementation as easily portable as possible. With Java this goal was accomplished very painlessly; we did not have to make any code changes, or even recompilations, to run the system on each of the numerous different environments supported by Athena. Second, Java allowed for considerably rapid development of networking code: debugging is very easy, garbage collection is built-in, exception handling is straightforward and overt (even forced), and a large, well-documented and robust class library is available. Finally, although we did not implement it, a nice GUI for monitoring and visualizing the Overcast network would have been fairly straightforward to build with Java. This might be an area of future work.

It should be noted that using Java does not lock ourselves, nor any potential third-party implementor, from creating an Overcast node that can inter-operate with our code-base. We took care not to use any Java-specific serialization techniques for networking. For node-to-node communication we developed a very simple byte-specific protocol that any third-party implementation can interact with trivially.

Aside from the use of Java, there are a number of implementation details that are either not mentioned in the original Overcast paper, or discussed there only briefly. Some of the important details, and our choices regarding them, are explained below.

4.1 Substrate Network

The "substrate network" (see section 2) we use is the Internet, and we followed the lead of the original Overcast authors by using TCP/IP for all connections. TCP/IP is well-established and suitable for all our needs; we saw no reason to build another protocol on top of either IP or UDP. Although UDP may have given us a minor speed advantage, there are better reasons to use TCP/IP instead. In particular, congestion and flow control are built into TCP/IP; we did not want to be bad network citizens by flooding the network with our UDP packets, and these are features we did not have time to implement ourselves.

4.2 Asynchronous I/O vs. Threads

As is typical for a software system which must concurrently support multiple network connections, there were two basic code models to choose from in building our Overcast implementation: handling all connections in a single thread using asynchronous I/O calls and a system call such as `select()`, or handling each network connection in its own thread, with blocking I/O calls.

Performance would call for using an asynchronous I/O-based model. Using threads involves a fair amount of locking and context switching, as well as additional memory overhead. Surprisingly, asynchronous I/O (provided by the `java.nio` package) is a relatively new feature in the Java API. The classic approach to Java network applications is to use the thread-based code model. For this reason, and because we were more comfortable and experienced with threads in Java, we chose the threading model over the asynchronous I/O one. Despite the performance hit this might incur, it is far from disastrous: as discussed in the original Overcast paper, it is not expected that a single tree node will ever serve more than about 20 clients, and the number of possible children for a single tree node will generally be a handful. Thus, the number of threads for one node should never be unreasonably high.

We believe that using threads and blocking I/O for our Overcast implementation led to a faster development time than using asynchronous I/O would have allowed. In addition, now that the implementation of the node-to-node communication protocols have been developed, it would not be very difficult to rewrite the code using asynchronous I/O. This thought, along with some ideas about slightly different code models, are discussed in the section on future work below.

4.3 Request/Response Pairs

Communication on the organizational connections is done in terms of synchronous request/response pairs in an RPC-type mechanism. That is, only one request

is "on the wire" at a time—a child makes a request to its parent, and the parent sends a response to the child; the child makes no other requests until this response has been accepted and processed. There are only three types of request/response pairs: keep-alive messages, node join requests, and data source requests. The latter two are done upon initial joining of a child node to the tree, and might be subsumed in a more generalized keep-alive response, but separating them leads to somewhat cleaner code. Furthermore, the same request/response code mechanism is used in a few other places where it is very convenient: as an example, when a child connects to its parent on a "raw data" socket to stream a data source, the socket actually starts as an extra organizational connection in the code, and both nodes involved agree to turn it into the "raw data" socket. This is a very code-technical point, but it illustrates the usefulness of the request/response mechanism we adopted.

We had the choice of either leaving the organizational connections always open, or creating them on the fly every time a message needs to be sent. The latter choice would not be unreasonable since after initial node joining messages, only the keep-alives are sent regularly, and this is done at infrequent intervals (a global constant in the code defines it as once every 30 seconds). To reduce some CPU overhead and to simplify the code, we chose instead to simply leave the connections open. As mentioned, the original Overcast paper makes the point that parents never initiate contact with children, because some children may be behind firewalls. We interpreted this to mean that organizational connections are, in fact, opened and closed for every message sent. As this is a rather minor detail, we are not concerned that we chose the opposite approach. We did, however, maintain the constraint that only children ever initiate requests to their parents.

4.4 HTTP Clients

We followed Overcast's approach of serving only standard HTTP clients. By settling strictly on HTTP, our implementation is simplified and a wide variety of existing client software (web browsers and MP3 players in particular) can connect to the network unchanged.

4.5 Naming of Data Sources

An Overcast network may provide several sources of data, e.g. a number of different large files, as well as a number of separate audio streams. Since only standard HTTP clients are used, it makes sense that data sources be named by URLs. In each URL, the hostname is that of the root node, and the file path corresponds to a data source which all tree nodes know about.

4.6 Data Synchronization

There is a potential problem with data synchronization for nodes that become rearranged in the trusted tree that we needed to solve. Consider, as an example, a user client retrieving an MP3 stream from a tree node: if this tree node reconnects to a new parent, it will still receive the same MP3 stream, but there are no immediate guarantees about the newly begun position of the MP3 stream at the node. The same problem exists for file data.

Our solution is to break up raw data into incrementally numbered packets. Thus, when a tree node disconnects from its parent, it knows the number of the last raw data packet it received. When it reconnects, it requests that its parent begin sending at the packet just after the last one it has received. In the case of file data, all past packets are cached, so requests for older packets are not a problem. In the case of audio or video streams, a significant number of older packets should be cached at each tree node to avoid the same problem. This cache likely needs to be quite large to allow tree nodes to move around under the re-building protocol without a problem, although if a node requests very old data from an audio or video stream, it is probably safe to assume that this node is heavily lagged and should be removed. If a node switches parents and requests newer data than is available, it will simply wait. If the difference in packets is unreasonably large, we might assume there is a similar lagging problem; the exact method for handling this has yet to be decided, but it should not be a common problem.

4.7 Assignment of Tree Nodes to User Clients

Deciding which tree nodes to assign to user clients may have considerable impact on the performance of an Overcast network. Running bandwidth tests from various tree nodes to the user client as a means of deciding the best tree node for the client would require multiple HTTP redirections and involve complicated code. While algorithms have been created to solve this problem, we chose a simple approach that works very well in the average case: since the root node keeps track of all its children, it simply distributes user clients randomly among all the tree nodes. Since this code is rather abstracted and does not depend on the rest of the system, it could be replaced in the future by a better algorithm if one proved to be necessary.

4.8 Client-less Root Node

We decided to make the root node explicitly not stream any data to user clients. The reasoning for this decision is that we don't want to unnecessarily tax the bandwidth of the root node; we want it to distribute data to the rest of the tree as quickly as possible. In fact, we considered having only leaves of the trusted tree ever actually serve user clients, but this might be wasteful of other tree nodes' available bandwidth. Furthermore, since nodes can move around during tree re-building, it is not clear how long a leaf node will remain a leaf.

Additionally, because the root node is a single point of failure, we want to take as few chances as possible with taxing the physical resources of the machine it runs on. The root node is already responsible for accepting connections from user clients and redirecting them to other tree nodes; we don't want to create extra work for it if there are a significant number of other tree nodes to handle clients.

4.9 Slow Tree Nodes

The policy for dealing with a tree node whose bandwidth becomes too slow is minimal. For file-based data sources, no *functionality* is lost by this decision because every node caches the entire file. For "live" streams, which are essentially infinite in duration, naturally only a portion of the data can be cached, so slow tree nodes that cannot be supported by their parents (because the cache has grown too old) must be dropped. We do not expect this to occur very often in practice for the following reasons. The choice of tree node placement on the substrate network should generally be good enough that bandwidth will not drop severely unless there is some sort of serious, unavoidable problem. Similarly, as we have assumed very large amounts of disk space at each tree node, caches for stream data should also be very large and support perhaps significantly lagged connections. HTTP clients which are too slow to receive their streams must be dropped—again this will likely be rare.

4.10 Third-Party Software

We made use of a few third-party software packages to help build our implementation. The obvious reason for this was to avoid writing mundane code that was not directly related to the Overcast system so that we could to concentrate more of our effort on the interesting parts of the implementation. `getopt` allows for simplified command line argument parsing. `log4j` provides easy, extensible logging.

5 Results

The goal of our Overcast implementation is essentially to make maximum use of available bandwidth while minimizing network load. In other words, we hoped for our system to transparently support very many clients.

Part of achieving this goal may lie in careful placement of trusted nodes on the TCP/IP network. Additionally, it would be ideal for the root to redirect each client to its closest tree node; however, as discussed, we did not yet include this as part of our implementation.

We were able to test the functionality of our Overcast implementation using computers on and nearby to MIT's Project Athena network (located in Cambridge, MA) and using a small number of computers on a slower connection in Washington, D.C. We would have liked to test a trusted tree spanning a much wider portion of the Internet, and consisting of

an equally diverse selection of clients, but we unfortunately did not have access to any such resources.

5.1 Stability Testing

Additionally, we tested the stability of the system by building a tree and connecting a small number of clients to it. In killing off a tree node in the ancestry of a client's connection, no noticeable drop in speed was detected at the client. This may be partially explained by the fact that all machines in the test were on the same 100 Mbit LAN; however, we were pleased with the results nonetheless.

5.2 Startup Time

A final test involved building a tree on five machines, of varying hardware and software configurations, and testing startup time on a 100 Mbit LAN. With one node on each machine, and with the root serving a single 10MB file, startup time was less than 10 seconds, averaged over five trials. This includes time for the tree to be built, and for the 10MB file to be distributed to all tree nodes.

5.3 Specific Applications

For our all tests, we used various file data sources. We regret not having an MP3 data source available at the time of testing. This would have been an exciting test of the system, and it is described below as possible future work.

6 Future Work

There are a number of improvements and extensions to be made to our Overcast implementation. Some of these are purely features of explicit usability, while others are more performance-related.

6.1 Asynchronous I/O

In some regards, we regret not using asynchronous I/O for our implementation. Though functionality would not have changed much, using such a model for our code would almost certainly have resulted in better CPU- and memory-related performance. The Overcast system should ideally be bandwidth-limited at each tree node, but we were unable to test code on machines with very fast network links, and we fear that for such machines bandwidth might not be the bottleneck. Switching away from a threaded code model would almost certainly help in this regard.

We should note, however, that given the infrastructure we have built, including the design of the communication protocol between nodes, it would not be a great amount of work to rebuild the code using an asynchronous I/O-based model. Using threads helped us to establish a working communication protocol as the code was developed. Because we were more comfortable using threads in Java, this would have been more difficult if we had begun our code using asynchronous I/O.

There are additional code models that might fuse the best of each of these approaches. We might retain an overall thread-based model, but set up one thread to handle all "raw data" traffic; asynchronous I/O would be used only within that particular thread. The connections handled by this thread would include those for transferring raw data between tree nodes, and those for sending data to the user clients. Since very little state needs to be associated with each of these connections, it would not be too difficult to adapt them to use asynchronous I/O—the node-to-node organizational connections would be more difficult to adapt, since they involve a good deal of response/reply state. This model, then, might even be ideal, since the busiest connections are placed in a single thread, and the lower-trafficked, but more state-dependent connections are in their own threads. In the future we might transition to this model before or even instead of a fully asynchronous I/O-based model.

6.2 Careful Placement of Tree Nodes to User Clients

An important improvement for the performance of Overcast is in the decision of the tree node to which each user client should be redirected. Our current policy for this decision, as discussed earlier, is fairly simple. We place a new user client at the tree node with the least current clients, thereby attempting to balance the number of clients at each node. The original Overcast paper only briefly mentions a bet-

ter policy: the root node contains information on the structure of the Internet so that the user client's IP address can be matched to a close[5] node in the tree. This seems to us to be the best policy since it would very likely determine good choices for user clients, and would not require any change to the client software.

6.3 Redundant Root Nodes

One particular fragile aspect of the Overcast system lies in the root node: if it goes down, the whole trusted tree fails (this is at least true for clients which have not yet connected). One possible solution, mentioned in the original Overcast paper, is to have any number of redundant root nodes: multiple "root" nodes are linked together in a chain. At one end of the chain is the true root node, and at the other end is the node which has children in the trusted tree. If any particular "root" node goes down, it is simply removed from this chain. Because these "root" nodes are linked as a chain, they are in a hierarchical order and the entire Overcast network remains a tree. Thus, the information stored at the tree nodes for the up/down protocol remains replicated correctly if one of the "roots" goes down.

Implementation of redundant root nodes would also require a round-robin DNS mapping to several IP addresses, and/or IP takeover when a machine fails.

6.4 "Live" or Continuous Data Streams

An interesting application of Overcast is to stream a "live," continuously-generated data source through the trusted tree to user clients. This would not be very difficult to implement given the infrastructure we have developed, but it would involve a fair amount of work and testing. For example, a more involved caching policy is required than for files: only a certain number of packets of the stream can be cached (since it may run for a very long time), the cache must be spread across several files so that the oldest portion of the cache can be removed to make room for the newest portion, and so on. Also, the cache would need to extend far enough into the past so that during tree re-building, there would be very little chance of a node switching parents and needing packets of the stream older than what that parent has stored.

[5]Here, "close" means topologically close, in terms of the structure of the Internet.

7 Conclusions

Overcast proves to be an interesting mechanism for large distributed data access, whether it be for static or live data sources, in order to lessen bandwidth requirements on servers throughout the Internet. We, the authors of this implementation, have built an implementation with Java that, with a few modifications and more extensive testing, is ready for deployment and usage on a grander scale.

References

[1] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, James W. O'Toole, Jr., *Cisco Systems*. "Overcast: Reliable Multicasting with an Overlay Network." From *4th Symposium on Operating System Design & Implementation*, October 2000.

[2] Jerome H. Saltzer, editor. *Project Athena Technical Plan*. "M.I.T. Project Athena, 1987." Reprinted in *Project Athena: The First Five Years, Volume 1*. Digital Equipment Corporation, Hudson, Massachusetts, 1988.

[3] Yang hu Chu, Sanjay G. Rao, and Hui Zhang. "A case for end system multicast." In *Proc. ACM SIGMETRICS Conference (SIGMETRICS '00), June 2000.*

[4] Paul Francis. "Yoid: Your Own Internet Distribution." Technical report, ACIRI, April 2000. `www.aciri.org/yoid`

[5] J. Touch and S. Hotz. "The X-Bone (white paper)." Technical report, SIS, May 1997. `www.isi.edu/x-bone`