Re-Transport Media
A Peer-to-Peer Streaming Solution

Javier Castro, Drew Houston, Sam Prentice, Will Stockwell

May 12, 2005

Abstract

In this paper we propose Re-Transport Media (RTM), a peer-to-peer streaming architecture which amortizes the bandwidth costs of serving internet media by cooperatively distributing the streaming load among connected clients. Conventional media streaming architectures typically impose significant bandwidth load on the server's Internet uplink, requiring the content provider to serve the stream to all end-users. In recent work, systems use an application-layer multicasting protocol and place clients in layered clusters to implement efficient content-distribution. RTM improves upon existing systems by reducing control traffic to constant time, using a centralized tracker to maintain an organized client tree structure. We create a functional peer-to-peer radio implementation of the RTM design as a proof-of-concept which uses the Shoutcast streaming protocol. The implications of RTM are widespread: RTM eases the bandwidth burden on current content providers, enables providers to broadcast higher-fidelity media streams, and empowers ordinary users with modest bandwidth to become content providers.

1 Introduction

Traditional media streaming systems, such as Internet radio and webcasts, rely on a single, centralized server to provide content, transmitting a separate instance of the stream data to each connected client. This unicast structure causes the content provider's bandwidth and CPU requirements to grow linearly with the number of clients.

In this paper we propose Re-Transport Media (RTM), a scalable peer-to-peer[1] ("p2p") system which amortizes this significant bandwidth cost across many network links by organizing peers in a hierarchical topology. In our proposed scheme, the content provider resides at the top of the hierarchy and serves content to a handful of first-tier peers who in turn re-transport the stream to second-tier peers and so on. The aggregate bandwidth utilization across all links for a given number of peers increases only minimally due to additional overhead required to manage the hierarchy, yet the potential number of total clients expands far beyond unicasting since listeners help to bear the broadcasting burden.

Recent systems have improved upon the traditional unicast system by implementing application-layer multicasting in an attempt to minimize stress on the underlying network links. While these systems focus on getting to most out of the underlying links, RTM seeks to bound the stress on peer links and minimize control traffic by using a centralized architecture. The system design is simple, yet robust, tolerating peer and network failures and maintaining an exceptional level of usability.

The underlying framework of RTM can be used to broadcast any form of stream data. In this project we implement a p2p radio system utilizing the Shoutcast[1] streaming protocol for audio stream data. RTM eliminates the bandwidth overhead assumed by Internet radio providers that serve an audio stream to all end-users. Rather than serving all clients, the content provider serves only a select few seed clients that then propagate the radio stream through the p2p network.

Peer-to-peer radio is of interest not only to large content providers serving a multitude of clients, but also to anyone with limited computing resources interested in broadcasting a stream. The exorbitant bandwidth costs of traditional streaming networks impose a significant barrier to entry, limiting stream broadcasting to only those with sufficient bandwidth and serving power. By distributing the broadcasting load among listeners, RTM enables anyone with modest computing resources to broadcast an audio stream. Furthermore, stream providers with limited bandwidth who preferred to serve a lower-fidelity audio stream in order to reach a wider audience may use RTM to get the best of both worlds, utilizing bandwidth savings to serve higher-quality streams to listeners.

[1] Technically, the peer-to-peer moniker may be misleading as clients are arranged in a tree. However, the system borrows many concepts from peer-to-peer systems, especially in the sense that first, listeners send to and receive from other listeners, and clients function as servers and vice versa.

The key design challenge faced by RTM is maintaining a dynamic client tree amidst network failures and peer disconnections. Client data buffering and protocol conventions allow the system to reorganize the tree such that these problems can be avoided gracefully. Nevertheless, inherent latency problems are an unavoidable effect of RTM and significant latency must be minimized by limiting the depth of the peer tree.

The paper is structured as follows: In Section 2 we give a brief overview of related work. Section 3 describes the system design and details of the server-side and client-side applications. In Section 4 we discuss our testing methodology to ensure proper function and stability. Testing results are presented in Section 5 and Section 6 gives a performance analysis of the system. The paper is concluded in Section 7.

2 Related Work

Streaming of live media through peer-to-peer networks is not a new idea in and of itself. CoopNet[2, 3, 4] describes a system similar to RTM. Rather than evenly distributing a stream from a server to several root nodes, CoopNet has a centralized server that serves as many clients as it can before forwarding new clients to previously existing clients.

Network-layer multicasting is not feasible for content distribution to end-users because this technology is not universally accessible. For this reason, Narada[5] and NICE[6] use application-layer multicast protocols for packet delivery from one source to many end-hosts. Narada attempts to optimize the overlay structure based on end-to-end measurements, while NICE builds on this by imposing a hierarchical structure on end-host clusters. The NICE protocol seeks to minimize control traffic in the creation and maintenance of a clustered tree structure for streaming data in O(logN) time, where N is the number of clients in the data stream. RTM performs tree maintenance in O(1) time by centralizing this task in a tracker, which organizes and reconfigures the position of clients within the tree. Furthermore, the amount of bandwidth required by nodes in each tree layer in RTM is bounded by a set branching factor. However, nodes that span multiple tree layers in the NICE protocol are essentially unbounded in bandwidth, as nodes on higher layers stream to clusters on all subsequent layers.

While not as effective as latency measurements in other systems, a simple bit-distance metric is used in RTM to cluster nodes of geographic proximity under common tree branches. This helps to minimize sending duplicate packets over a given link, thus decreasing the stress on links in the network.

Microsoft Research designed SplitStream[2], a streaming architecture which differs from the standard tree approach used by RTM and CoopNet. The main goal of SplitStream is to distribute not only the bandwidth load, but also the forwarding load that is placed on a server in the tree-based architecture into the clients of the system. This goal is accomplished by separating the shared media into pieces called stripes, and then distributing the stripes in separate multicast trees. While this approach to distribution could prove beneficial to systems designed to provide static content, it is not ideal for streaming live media content which is consumed in real time. First, there is typically not enough live media present at a given time to effectively divide the stream into stripes for distribution. Second, stripes present additional complexity since all sequential packets in a live audio stream must arrive and be arranged without any missing members before the stream is consumed to offer reasonable usability.

A third media streaming system, PeerCast, is proposed by Deshpande[5]. This system suggests tree maintenance policies which, despite the authors' efforts, degrade with the number of clients. Additionally, the UDP/RTP protocol used by PeerCast is not sufficient for the live streaming of audio data.

Zebra[7], a system designed by 4 MIT engineers, focuses on streaming live video. Zebra borrows ideas from SplitStream and achieves acceptable usability due to the redundant nature of sequential video images, in that human vision is tolerant to minor stream disruptions. It is unlikely that this same strategy would be effective for audio data, since humans are more sensitive to aural disturbances. RTM seeks to overcome these limitations and provide a robust p2p audio streaming infrastructure.

3 System Design

Typical Internet radio stations suffer from both severe limitations on the number of simultaneous users and unfavorable trade-offs in audio quality to allow a greater number of connections.

The primary goal of RTM is to eliminate these weaknesses by moving the majority of the cost of data transmission from the streaming server to the listeners, enabling a higher number of total clients that can be served by low-bandwidth stream sources. We place a hard upper bound on the number of clients served by the original source and on the number of clients to which a parent re-transmits stream data. RTM must also be able to transparently layer on top of virtually any existing unicast streaming media applications, and therefore eliminate the need to alter existing client or

server programs (e.g. Shoutcast server and Winamp client) software. RTM should do a reasonable job of placing clients that share network locality near one another in the tree to optimize bandwidth usage and minimize overall latency from the source of the stream.

RTM itself consists of two software components – in our nomenclature, a Client and a Tracker – which work in conjunction with a popular media player (Winamp) and streaming server (Shoutcast). The RTM system acts as an intermediary: it appears to the streaming server as a single client and appears as a Shoutcast server to the end user's media player.

The RTM network is a tree of Clients managed by a Tracker. Figure 1 depicts RTM's tree structure, in comparison with the traditional streaming structure. The Client at the root of the tree is configured to receive the stream from the originating Shoutcast server (the "Source"), instead of receiving audio data from other Clients. Each RTM Client also acts as a Shoutcast server to the local media player, leaving the local media player oblivious to RTM's existence.

3.1 Client

The RTM Client runs on each node in the tree and serves three functions. First, it is responsible for receiving from and forwarding audio data to other Clients. Second, it maintains a receive window of audio data which acts as a buffer to maintain skip-free operation in the face of typical data transfer anomalies, such as packet loss and variances in latency between Clients. Finally, it provides an interface to Winamp that simulates a Shoutcast server.

3.1.1 Inter-Client Communication

Clients and Trackers communicate using an RPC mechanism over a persistent TCP connection. This relieves concern of retransmitting messages believed to have been lost; TCP does this for them.

Clients are slightly out of sync due to connection latency and especially due to disruptions in service or relocations within the tree. To curb this problem, each Client buffers both upcoming audio data as well as data already sent to the media player (that would otherwise be discarded.) This allows newly connecting Clients to specify an absolute stream offset and thus to pick up exactly where they left off, when possible.

3.1.2 Media Player Interface

The Client also acts as a Shoutcast server, accepting connections from localhost on an arbitrary port. The listener's Winamp client then connects to this virtual Shoutcast server and receives the stream as if it were connecting directly to the Source. Small protocol-dependent modifications to the metadata within the stream (e.g. client IDs, session cookies) may be necessary to provide a coherent interface to the media player.

3.1.3 Cooperative Buffering

For resiliency, our Client software engages in both backward and forward buffering of data. The forward buffer holds data in anticipation of parent failures so that the Client maintains data which it can serve to the local Winamp instance while waiting to be reassigned. When these Clients relocate, their new parents use their backward buffer to help the Clients quickly refill their buffer (hence 'cooperative buffering'). The cooperative buffering technique enables the Clients to assist in ensuring that each Client has a complete, uninterrupted copy of the data stream.

Reassignment only causes audible disruption if the Client's buffer is exhausted. In our implementation, both buffers are 1 MB, holding about a minute of audio. This provides Clients with more than enough time to be reassigned within the tree.

When a Client joins an RTM system, its parent provides it with a global offset specifying the number of bytes which have been transmitted by the stream source since the system was first initiated. All Clients continually maintain knowledge of the current offset values stored in their buffers. When a Client is reassigned due to parent failure, it can inform its new parent of precisely the stream data it requires using this global offset, thereby ensuring a seamless transition from old parent to new. The new parent then bursts data from its backward buffer (hence the 'cooperative' qualification since this data is of no use to itself) to the Client until it is caught up to the latest data the parent has to offer.

3.2 Tracker

The RTM Tracker is responsible for managing the tree of nodes. The Tracker should be run on the same host as the Stream Server. When a Client joins the tree, the Tracker finds a suitable location for it using the algorithm described below. If a Client leaves the tree or fails, it or its children respectively will report the situation to the Tracker which will subsequently relocate the children to new parents using the same join algorithm with an additional requirement: new parents should have a depth similar to the old parents.

The Tracker maintains a complete representation of the client tree in its local memory. This representation gives the Tracker all the information it needs to perform the join algorithm.

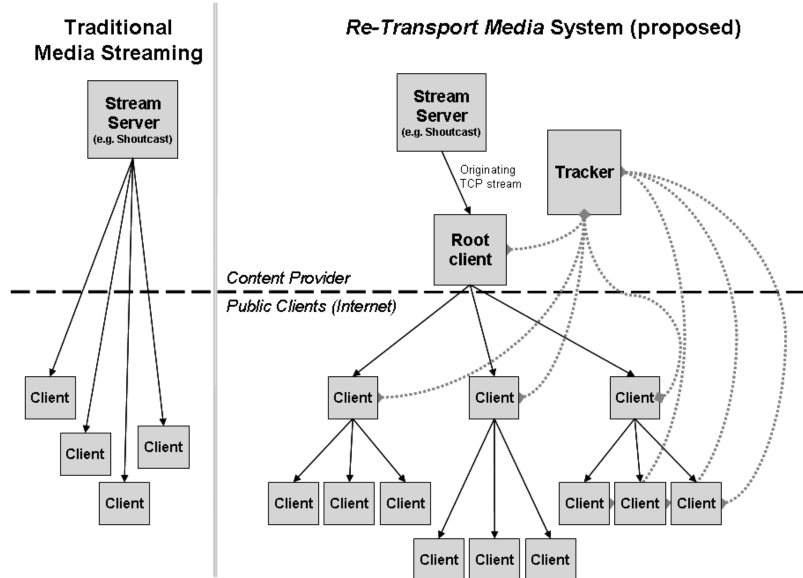


Figure 1: System overview diagram

3.2.1 Join Algorithm

Clients request access to the streaming service by asking the Tracker for a parent. The parent performs the algorithm shown in Listing 1. The join algorithm measures the bit distance (described in the following paragraph) between the joining node and each Client in the tree with fewer than k children, looking for the Client with the minimum bit distance. If multiple Clients have the same bit distance from the joining Client, the algorithm selects the one with minimum depth in the tree. If two nodes share the same bit distance and depth, either may be selected.

The Tracker uses a measure called bit distance to determine where best to locate hosts in the Client tree. Bit distance is a distance measure of two hosts based on the length of IP address prefix they share. If two IP addresses have a common prefix of 8 bits, their bit distance is 24. If two addresses first bits differ (i.e. they share no common prefix) their bit distance is 32. This measure allows us to make reasonable choices in the interest of placing hosts which are near one another on the underlying network near one another in the Client tree. Bit distance is a useful heuristic, however it is only an approximation. The actual network distance depends on the underlying subnet structure, of which we have no knowledge. For example, if the best possible parent has a bit distance of 16 from the joining Client, it is possible their respective class B networks are routed to entirely different geographical locations.

```
JOIN(V, u):
  mdist <- 33
  mnode <- nil
  FOREACH v IN V SUCH THAT v.num_children < k
    IF BIT_DISTANCE(u, v) < mdist
      mdist = BIT_DISTANCE(u, v)
      mnode = v

    IF BIT_DISTANCE(u, v) = mdist
      IF u.depth < mnode.depth
      mnode = u

  SEND_LOCATE_RPC(u, mnode)
  ADD_CHILD(mnode, u)
```

Listing 1: Join algorithm pseudocode. V is the set of tree vertices and u is the joining vertex

3.2.2 Reassignment Algorithm

Inevitably, Clients will decide to stop listening to the stream or will disconnect due to failure. Such Clients may have dependent child Clients to whom they are retransmitting the stream at the time of disconnection. The Tracker must be able to handle these events and respond by reassigning these children Clients to new parents in a reasonable manner.

If the disconnection is user-initiated, the Client sends a disconnection announcement message to the Tracker. The disconnecting Client operates normally for a "cooldown" period to allow its children a chance to obtain the source from elsewhere. In response, the Tracker performs the reassignment algorithm in Listing 2. At a high level, this algorithm redistributes the departing Client's children in a similar manner to the join algorithm. However, in addition to finding a bit-near host with less than k children, the reassignment

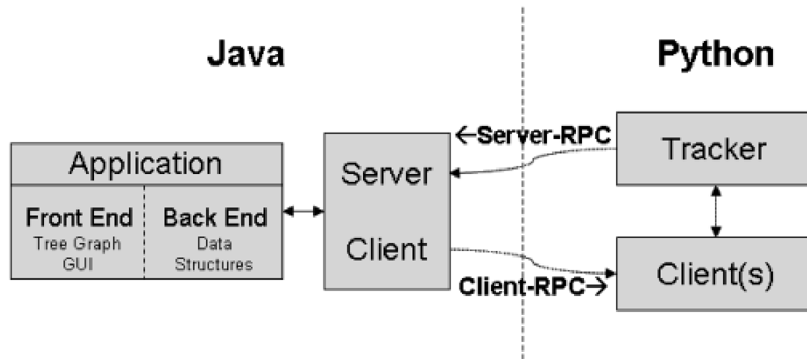


Figure 2: Overview of test application design

algorithm attempts to find a new parent that has similar latency with respect to the stream source as the departing Client. To do this, the reassignment algorithm runs the join algorithm for each of the Client's children, but prefers new parents of depth closest to that of the departing Client rather than nodes closest to the stream source. This prevents the children from being reassigned to nodes with a significantly different latency than its parent. This modified join algorithm is run for each child Client sequentially (without waiting for network messages to be sent to each), so it is possible that one of the children might become the parent of another once all reassignments have completed.

However, some disconnections due to network or host failure are inevitable. To handle these failures gracefully, all Clients buffer stream data ahead to "buy time" during which they report parent Client failures to the Tracker and request relocation in the tree. Clients also buffer data that has already been transmitted to all of its children for a period of time in anticipation of being assigned a child whose parent has failed. This allows the child Client to download data it missed before it could be relocated in the tree, getting a burst of data at a rate faster than the playback rate. These mechanisms allow the Tracker to execute the reassignment algorithm just as it would if it had been initiated by the failing parent via a disconnection announcement.

4 Testing

We used a collection of PlanetLab[8] nodes to function as our Client pool when testing our system. To demonstrate that the bit distance metric localized Clients well, our set of PlanetLab nodes included hosts in North America, Europe, the Middle East, and Asia.

We developed a GUI interface in Java to facilitate testing of the system. The GUI displays an interactive tree graph of nodes and connections in the RTM network. The GUI utilizes the Prefuse[9] graph visualization toolkit to draw and animate graph updates (as it receives information from the Tracker). This visualization allows the user not only to view the Tracker's join and reassignment algorithm decisions, but also to simulate failure conditions via popup menus on each node. Through these menus the user can inject faults at a selected node by specifying that a Client simulate a crash or network failure, halting communication on one or all data stream connections. Clients have a testing harness built in to receive these requests from the GUI and simulate the corresponding fault.

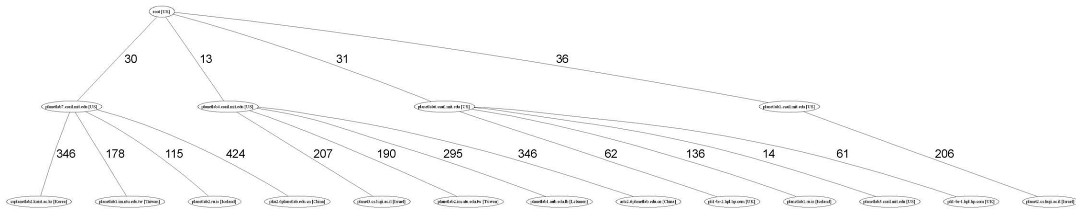
The Python implementation (Tracker and Client) and the Java GUI application communicate as depicted in Figure 2. The GUI first registers with the Tracker which informs it of the current state of the Client tree and subsequently transmits messages to the GUI any time a change in the tree occurs. Whenever a GUI user decides to inject a fault at a particular node, the application transmits and RPC directly to the node. The injected fault causes affected Clients to react by requesting the Tracker to reassign them. After running the reassignment algorithm, the Tracker informs the GUI of the change to the Client tree.

5 Results

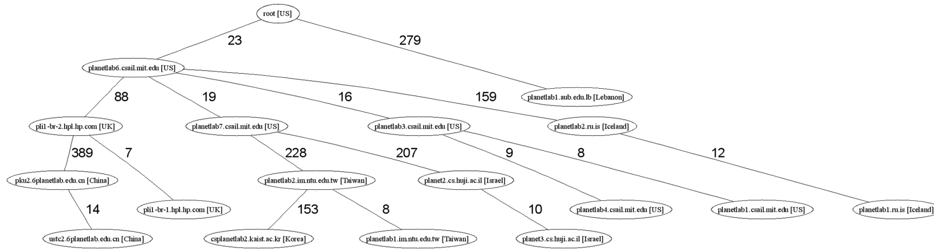
We performed our testing within the PlanetLab environment. We constructed a tree of about 20 nodes and the stream was successfully propagated down the tree; we were able to tune in to Shoutcast streams from peers situated in Taiwan, Eastern Europe, India, and so on. We created a test framework to measure the effectiveness of various tree organization algorithms, to analyze the "stream lag" across various levels of the tree, and to measure the time needed to relocate within the tree.

5.1 Tree Organization

The naive tree organization algorithm, upon addition of a node, conducted a simple breadth-first search to locate an open slot. This led to inefficient trees, as paths like USA→Korea→UK→US were created (rather than



(a) Breadth-first join algorithm



(b) Bit-distance join algorithm

Figure 3: Breadth-first vs. bit-distance join algorithms. Edges represent latency in milliseconds.

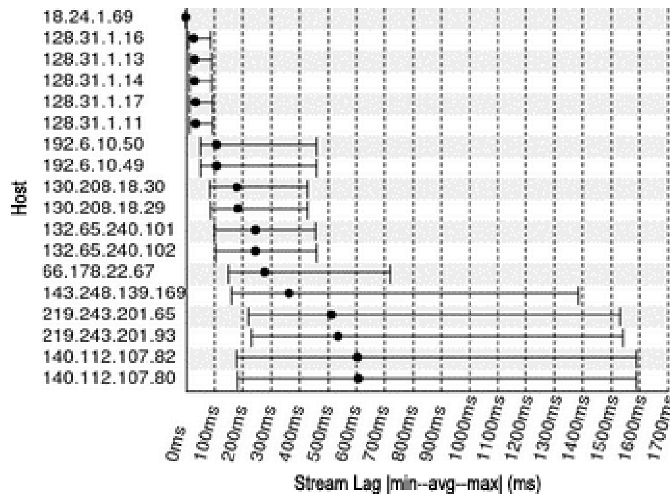


Figure 4: Stream lag in milliseconds

US→US→UK→Korea or some other more efficient alternative.)

The tree was much cleaner with the bit-distance-based join algorithm, and different localities formed clusters within the tree. For example, all of the Israeli nodes were grouped together and all of the Korean nodes were grouped together and routed via Taiwan, etc. Figure 3 compares the bit-distance and breadth-first join algorithms.

5.2 Stream Lag

To analyze performance, all nodes send a UDP datagram every time they pass a 128KB boundary in the stream. The timestamps of these incoming UDP datagrams (less one-half of the round-trip time to the host) enabled us to determine the lag between various levels of the tree, as the root node would return a packet first, followed by the first level nodes a certain time interval later, and so on.

In one test with 19 nodes (using the optimized tree-join algorithm), the steady-state "stream lag" from the root node to bottom nodes of the tree was around 600ms in the average case (with hosts in Asia.) In other words, stream data took about 600ms to propagate from the root to the leaves. This is shown in Figure 4.

Average stream lag stayed consistent over time; any clients that started out trailing the stream usually caught up within about thirty seconds to a minute.

5.3 Relocation

We were also able to simulate failures and record the amount of time required for abandoned children to complete a full relocation to other parts of the tree. Recovery from single, isolated failures was quick - 419ms on average, 1042ms worst case (across 17 individual relocations.) The variance in recovery time was much higher when multiple failures were simulated within a span of a few seconds; in one trial, with recovery times ranging from a minimum of about 300ms to a maximum of 4200ms and a mean of around 1020ms (across 42 relocations.)

6 Performance Analysis

We were pleased overall with the system's performance, and the stream was successfully propagated through a tree of 20 nodes. The system proved reliable and ran unattended for periods of several hours without incident. In addition, the stream lag (top-to-bottom skew) was less than anticipated (on the order of seconds in the worst case); we had planned for potential skew on the order of tens of seconds or minutes. Furthermore, the bit-distance join algorithm successfully clustered nodes within the same continent, minimizing the use of slower transcontinental links. The stream played continuously through both single- and multiple-node failures (simulated) and abandoned nodes were reassigned more quickly than presumed. We assumed that we would need to endlessly tweak the size of the buffer for optimal performance, but our initial one-megabyte forward and backward buffers proved sufficient in nearly all cases.

However, the PlanetLab testbed was a bit too homogenous. First, unlike anticipated real-world clients, it was extremely reliable and few nodes ever crashed; all failures needed to be simulated. Second, all PlanetLab nodes had more than enough bandwidth to forward the stream; we imagine that slow clients (whose downstream bandwidth is less than the bit rate of the stream) could cause disasters at lower levels of the tree. Finally, no PlanetLab nodes were behind firewalls; in reality, we expect NATs and firewalls to cause headaches because our current implementation relies on peers to be able to connect to each other. We believe that these problems would be even worse, though, for systems such as NICE and Narada which depend heavily on peer-to-peer communication for their clustering algorithms; our single communication channel to the tracker is more NAT-friendly, is simpler, and has less overhead.

Another conceivable problem that was not manifested under the PlanetLab testbed is a stalled client (caused by an infinite loop, saturated connection, etc.) Stalled clients could introduce severe delays into large portions of the tree, so special care should be taken by clients to relocate if they discover their parent has not sent data within a certain time interval and is not responding to keepalive or ping requests (Our current implementation does not take special measures to detect this scenario.)

Assuming that the NAT/firewall and stalled client issues could be overcome through careful implementation, we feel that our system could scale to thousands of clients (as long as the tracker had enough computational power, bandwidth, file descriptors, and so on to handle the incoming control traffic.) Each individual node maintains a constant number of connections and a constant amount of buffer space, etc., and computational demands on the tracker seem to rise quadratically (for example, in the worst case, the join algorithm would iterate over every node in the tree.)

7 Conclusion

RTM provides application-layer multicasting functionality using a very simple design. By organizing Clients in a tree structure, a single low-bandwidth source is able to serve many Clients well beyond the number possible in traditional unicast system. The design emphasis minimal per Client load by bounding the number of children served per Client, thereby allowing many low-bandwidth hosts to participate in the service.

For ease of prototyping and development, we implemented the system in the Python programming language. Python has a highly capable standard library (including the framework on which our RPC mechanism shall be built), and because it is cross-platform, RTM can be used on many operating systems. In contrast, NICE has only a poorly-maintained implementation for FreeBSD (though it apparently compiles on Linux as well.)

The choice to leverage existing streaming frameworks and the use of Python proved invaluable for rapid development; changes could often be made to certain parts of the Tracker source code and incorporated without requiring a restart of the entire system. We were able to put together not only a fully working system but also visualization tools and a testing framework by demonstration day.

Altogether, the RTM implementation was comprised of a mere 4100 lines of code. The Tracker and Client code together comprised a mere 1300 lines of Python while the GUI testing application comprised 2800 lines of Java. In comparison, the NICE project software comprised 10500 lines of C code without including a GUI interface.

RTM requires the least aggregate control overhead of all the application-layer multicasting systems we examined. Other application-layer multicasting systems require significant inter-client control chatter to function properly. RTM eliminates inter-client control chatter altogether. Control operations in RTM only require communication between the Tracker and a single Client. The choice of our Tracker-based design provides two benefits. First, control operations are always limited to a constant amount of network traffic per operation. Second, hosts behind firewalls and NATs can use RTM (albeit only as leaf nodes since children cannot connect to them) whereas they would be prevented from participating in NICE or Narada which require them to be able to receive control messages as well as transmit them. RTM (and all other application-layer multicast systems) already has a single point of failure at the stream source. Co-locating the Tracker with the source of stream data on a single host does not diminish robustness because the failure probabilities of the two components will have an extremely high correlation.

The design of RTM is built leveraging existing streaming technology. Service providers and listeners are able to use the same software to which they are accustomed. We worked with Shoutcast in our reference implementation, and used standard software to provide and listen to the stream. However, RTM could be easily adapted to any streaming application including video, stock tickers, and gamecasts.

Clients located in lower tiers of the tree receive the streamed transmission with an amount of delay not experienced by Clients at higher tiers. Using a non-optimal best effort scheme based on IP address bit distance, RTM does a reasonable job of positioning nodes with relative topological locality close to one another in the Client tree, while minimizing tree depth. Other protocols do a better job of minimizing load on network resources, but suffer from inability to function on hosts behind firewalls and NATs which are particularly common in the relatively low-bandwidth commodity market where RTM is most useful.

References

1. http://www.shoutcast.com/
2. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh. SplitStream: High-Bandwidth Content Distribution in Cooperative Environments. IPTPS'03, Berkeley, CA, February, 2003.
3. H. Deshpande, M. Bawa, H. Garcia-Molina. Streaming Live Media Over Peers. Stanford Database Group, 2002.
4. V. N. Padmanabhan, H. J. Wang, P. A. Chou. Distributing Streaming Media Content Using Co-operative Networking. ACM, NOSSDAV'02, May 12-14, 2002, Miami, FL.
5. Y. Chu, S. Rao, H. Zhang. A Case for End System Multicast. ACM, SIGMETRICS'00, June, 2000, Santa Clara, CA.
6. S. Banerjee, B. Bhattacharjee, C. Kommareddy. Scalable Application Layer Multicast. ACM, SIGCOMM'02, August 19-23, 2002, Pittsburgh, PA.
7. M. Dobuzhskaya, R. Liu, J. Roewe, N. Sharma. Zebra: Peer To Peer Multicast for Live Streaming Video. MIT 6.824, May 6, 2004.
8. http://www.planet-lab.org/
9. http://prefuse.sourceforge.net/