Reliable Real-time Stream Distribution Using an Internet Multicast Overlay

by

Ilia Mirkin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2006

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2006

Certified by .....
Andrew B. Lippman
Senior Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Reliable Real-time Stream Distribution Using an Internet Multicast Overlay

by

Ilia Mirkin

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2006, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

A real-time peer-to-peer stream distribution system is proposed. Distribution network adapts over time as users are added and removed, as well as due to changing network conditions. Every node both receives and forwards traffic, cooperating to minimize the bandwidth requirement on the source. The goal is to demonstrate that such a system is feasible, and that it can reduce the resource requirements of special purpose broadcasting.

Thesis Supervisor: Andrew B. Lippman
Title: Senior Research Scientist

Acknowledgements

First, I would like to thank Andy Lippman for his helpfulness with bringing many ideas back to reality as well as his infinite patience in helping understand how technological problems affect end-users.

Second, I would like to thank David Reed for helping with early versions of this document by providing invaluable feedback and direction, as well as perspective from extensive life-long experience.

Last, but certainly not least, I would like to thank Dimitris Vyzovitis without whom this project would have probably never seen fruition. His part in the implementation of many key parts of the underlying library as well as a few parts of the VidTorrent system, along with the early morning (or really late evening) discussions were of immeasurable help.

Contents

Chapter 1

Introduction

We have recently entered the age where video on demand is the norm and not the exception or desire. People time-shift shows, recording them on VCR's and DVR's with great ease. This also means that they watch the shows that they record at different times. However, while sometimes extremely convenient, this detracts from one of the main advantages of a simultaneous broadcast -- community.

One of the main advantages to watching (or listening to) shows at the same time is the ability to comment and discuss them with others, especially during the show. Broadcasting works well over the air waves, but distributing a video stream, or, in general, a stream of bytes to many people on the Internet over the heterogenous network presented therein is currently done using rather inefficient techniques.

This communications problems can be defined as the *multicast* distribution of *real-time streams*. A *stream* is a continuous string of bits, without a beginning or an end. This string comes in at a certain rate, and should be distributed to other nodes at the same rate for the system to operate in *real-time*. *Multicast* distribution in this case is the concept that the same stream will be distributed (copied) to many destinations, rather than an allusion to

the particular implementation details.

The motivation behind this project is to lower the cost to the stream originator of stream distribution in terms of both computational power and bandwidth requirements on a shared network like the Internet. The stream content can be audio, video, or anything else that can be represented by a computer. Trying to send the same stream to many hosts on the Internet generally results in multiple unicast streams. This in turn means high uplink bandwidth usage on the side of the provider, resulting in higher costs and poor scaling properties.

Let us consider instead a system that is enhanced by client nodes, not decimated by them. A stream originates from a single source that has a certain sequence of data that it wants to send out to the Internet at large, received by anyone interested in listening. Any listener (node) must have enough capacity to receive the bandwidth of the stream. Furthermore, we want to create a system that behaves in a cooperative fashion, so we require that any node must be able to resend the stream to at least one more node[1]. (While any given node may not be using its capacity to retransmit the stream, in order to guarantee entry-points, this must be a requirement.) Given these restrictions we want the resulting network to adapt dynamically to ever-changing network conditions which occur in a real network along with node failures.

There are few proposals and fewer implementations for doing this currently available, and they are discussed below along with their shortcomings. Following this discussion, a new approach is proposed along with some calculations and measurements of its performance.

[1]This requirement is relaxed in discussions in sections 2.5 and 3.1

1.1 IP Multicast

Ethernet multicast has existed for dozens of years, but its primary limitation is that it only works on a local Ethernet segment. This has been partially addressed by the Mbone[6][7] project and IP multicast. Through careful routing, they provide people with the ability to join multicast groups, and thus achieve the ability to do multicast over the Internet by duplicating the packets that are sent at the router. This leads to two big limitations. First, the way that routers figure out the overlay that determines the routing is not based on a stream's bandwidth utilization. This is a problem because it does not consider the link's capacity, only whether it is present or not. Secondly, very few networks route Mbone properly, and certainly no regular Internet service providers will allow their subscribers to obtain a multicast stream. This reduces the usefulness of the Mbone project to networks where fine control over the routers is possible.

Another problem with IP multicast is billing. When you have a unicast packet traveling across the Internet, it is clear who the packet is from, and where it goes to. The sender pays an ISP and the receiver pays an ISP. With multicast, it is more complicated, since the whole advantage of it is that it does not use as many resources as unicast. The sender only sends one "stream" to the ISP, but the ISP is then supposed to replicate the packet to a number of different routes and send them on. Moreover, the sender is unable to control who the data gets sent to, thus one could incur costs to a sender without them even knowing it. This is most likely the reason that IP multicast has not been deployed on the Internet at large.

Even though IP multicast in general does not directly address streaming, only packet de-

livery, some protocols have been developed that do so, within the given multicast framework. The protocol that best addresses the streaming problem is the Resource ReSerVation Protocol (RSVP)[12]. This protocol is implemented in some Cisco routers, however, in addition to having the same problems as IP multicast, it faces even less support on the Internet as well as much higher algorithmic complexity. The protocol calls for reservation of resources such that a router will make sure that there is enough bandwidth on a link for the multicast data to go through. Since this has to be assured along the entire length of the path that the multicast data takes, this involves a lot of extra overhead.

1.2 Bullet

Bullet[9] is a system proposed by researchers at Duke University. It is designed for data dissemination over a mesh structure. Their observation was that if trees are used for delivery, then the system suffers because a weak link in the tree will cause all of the lower down nodes to be limited by this weak link (and Bullet addresses this problem). This is true if the assumption is that the amount of data to be delivered varies, and/or that the stream can be split up such that not all parts are required in order to decode the data, as with Multiple Description Coding (MDC), or with added redundancy. However if we assume a (relatively) constant stream bitrate, then it does not matter where in the tree a given node is – either it is getting the full stream or it is not. If the node is getting the full stream then it should be able to retransmit the data to its children, independent of the bandwidth capacity between the node and its parent, so long as that capacity is sufficient for obtaining a full copy of the

stream in the first place.

As a result, the Bullet architecture is not well-suited for a stream multicast situation, but is better suited towards its original purpose – raw data distribution. Once we constrain the problem to a constant stream, we can make many assumptions that the Bullet team could not, which lead to more efficient algorithms for the particular problem of single-source streaming.

1.3 SplitStream

SplitStream[2] is a system that is much closer in scope and capabilities to the desired system than the previous examples. In my opinion, its overall key contribution is the idea of partitioning the stream up and sending each stream partition separately. This allows for nodes to be used in the cooperative network which have asymmetric links that allow them to receive the full stream but not be able to retransmit it in full. Splitting the stream up into many pieces allows such nodes to just retransmit one of those pieces. Furthermore, it configures the nodes as a mesh.

If we have a stream with bitrate N, and we split it up into M partitions, then we end up with M separate streams with bitrate N/M each. This effectively lowers the bandwidth requirement for a peer to be a useful node in the network, and thus allows more peers to participate in the redistribution effort. This addresses the common problem of asynchronous connections where the download capacity is big enough to receive a full stream but not big enough to resend it to another node.

Unfortunately, SplitStream does not guarantee the delivery of each stripe, meaning that many nodes may end up with only part of a stream. If MDC or redundant coding is used, then this may have little ill effect, however we want this system to be able to reliably send a full real-time stream. Moreover, SplitStream relies on a few layers of software underneath in order to perform routing, which increases the protocol overhead, which is undesirable when one wants to squeeze the most possible out of a network connection.

It also seems that SplitStream does not explicitly deal with congestion in the middle of the distribution tree rather than near the source. This is something that can and will occur in real Internet overlays since a node's uplink capacity can vary, sometimes drastically, over the period of it being a member of the overlay.

Lastly, there is little mention of adaptation due to changing network conditions. This is a key factor, as throughout the day overall usage of network lines changes, and there are even changes on smaller timescales also. The ideal system would dynamically perform congestion control in a non-intrusive fashion by acting as a single TCP stream for the whole application. (For example, BitTorrent uses TCP for each of its connections, but it makes hundreds of them thus being unfair towards other applications that use single TCP streams for their operations.)

In general, it seems that it would be hard to make SplitStream adapt to a variety of network conditions because it does not have the ability to directly deal with the network but instead relies on the Pastry DHT infrastructure in order to pass data between nodes.

1.4 BitTorrent

BitTorrent is a system for distributing files using the peer-to-peer concept[5]. File distribution, however, is a completely different problem than distributing a stream. A file is an array of bits, all of which are known at all times. A stream, on the other hand, comes in at a certain rate, and does not have a beginning nor an end. Moreover, it is impossible to tell what it will be in the future, and it is furthermore impossible to remember everything that it was in the past.

This leads to an entirely different technique for content distribution. The file-based content can be split up into many little chunks and the chunks can be sent to as many peers as possible, at which point these new peers will be able to redistribute those files. There needs to be no order on the reception of these, so any peer can receive any chunk at any time. The connection pattern that results is a mesh, as there needs not be any implicit hierarchy between the peers.

BitTorrent can be modified so that it only sends out the "current" chunk at any given time. However, since it does not provide any timely delivery guarantees, there are many scenarios where its performance is suboptimal compared to an approach that uses a structured tree. A direct peer-to-peer solution requires a hierarchy between the peers, and is used in VidTorrent.

1.5 End System Multicast

ESM is a system developed at CMU with many of the same goals as VidTorrent. It uses a tree-based overlay to distribute data, but takes a much more video-based approach to the data than VidTorrent.

In the implementation (or at least the way that it is described, source code is unfortunately not available), the source will send out three separate streams along the same distribution tree – a high bitrate video, a low bitrate video, and an audio stream. Thus based on a node's bandwidth, it may receive all three, or only the latter two, or just the audio. (This is one way of doing MDC's, although without the advantage in data size that they are supposed to present.) If a client does not receive all three streams, then any of its children will, of course, only be able to get just the streams that the client is getting.

The earlier paper[4] that describes their system lists many of the characteristics achieved in this thesis. However the presented scheme has a lot of reliance on centralized resources by rather arbitrary use of waypoints (well-connected servers) as well as a reliance on the details of the stream to be encoded as specified.

The later paper[10] introduces many improvements, also seen in this thesis. Primarily, this is their description of multiple tree use to achieve greater diversity. However they use this in the context of MDCs instead of achieving diversity on an arbitrary data stream. Also, this paper's suggestions are not found in the implementation provided (at least as described).

1.6 Resilient Peer-to-Peer Streaming

Microsoft research had a second system[3] built around the same time as SplitStream. In this system, they take the more direct tree-based approach rather than building on top of the Pastry infrastructure. This system focuses on improving multicast distribution of video encoded with the use of MDCs. Much like SplitStream, they use the multiple tree concept, and try to build trees that minimize the number of nodes between the leaves and the root. This is based on the argument that the fewer nodes there are in between, the lower the chances of one of those nodes failing, for any constant number of nodes in the tree.

Due to this paper's focus on video, they were able to analyze the distribution method's performance in a rather novel way, by considering the SNR of the incoming image. This is affected by packet loss (presumably using UDP or the like), and how many of the MDC parts there are available. Their results showed that increasing diversity, i.e. more trees greatly improved SNR in the face of packet loss.

Chapter 2

VidTorrent

2.1 Rationale

The high-level VidTorrent approach to streaming is the simplest one to rationalize conceptually. Instead of dealing with a file, we must deal with a stream. A stream means that there is a concept of a present time (subject to some finite amount of buffering), a past that has already been forgotten, and a future that we can't predict. As such, whenever we receive any data, we must send it on immediately, otherwise it becomes an uphill battle to try to keep track of more and more data that is received.

These constraints naturally lead to a tree-based design[1], since any node that already has the data does not need to receive it again, and upon reception, the node must send the data on to some other node due to the finite buffer problem. As such, each packet travels down a spanning tree of the fully connected graph of nodes. It is possible to also consider frameworks where each individual packet travels down a different spanning tree of this graph, and we do this to a certain extent in the multi-tree section below (2.5). Generally, a distribution tree is also known as an *overlay*; the two terms are used more or less interchangeably in this paper.

The main question that is addressed in this chapter is how to construct these spanning trees while maintaining minimal information on each node about other nodes in the overlay, and without any explicit global view on the network deciding on the fate of incoming nodes.

2.2 Overview

With VidTorrent, we try to achieve an end-to-end solution which allows data to flow from a source to many clients with as few requirements as possible on the source or any given peer in the overlay network being used to carry the data. In addition, each node must maintain only limited local knowledge about the state of the distribution overlay.

We also want the overlay to be content-agnostic. While the name of this protocol does start with "Vid", VidTorrent does not necessarily have anything to do with video per se – it is just a very compelling application for sending streams of bits from one source to many clients.

Overall latency need not be a concern for the system. Since this is a one-way broadcast system, there do not need to be any delay guarantees from the source to any given peer in the overlay. The only guarantee required is that the stream is delivered at the rate that it is coming in. The analogy to this is that one doesn't care if they are watching television five seconds off from it being recorded – but if there are breaks in the video, stream, then the effect becomes perceptible to the user.

As with the other viable approaches discussed in the introduction, this system uses a tree-based distribution system for delivery. The problem pointed out by the Bullet[9] paper

regarding the capacity of the tree decreasing as one descends it is not a big concern since as long as a node can receive a stream, it can resend the same stream to its children – there is no effective decrease based on how much "empty" pipe is left-over.

Another key point of this system is that it will adapt to changing network conditions. This is crucial since it is a generally accepted fact that channel capacities between any two nodes on the Internet at large will change throughout the day. That means that any distribution tree will need to adapt over time to account for these as, e.g. a node may be able to support many children at night but only one during the day. This implies that there needs to be active congestion detection and mitigation.

Lastly, as this must be able to be deployed in a real network environment, including home connections, the system must have the capability to include hosts behind Network Address Translation (NAT) firewalls/gateways. A recent study[8] has found that most commercial (especially recent) hardware is amenable to either TCP or UDP hole-punching, which means that given at least some public nodes available for intermediation, it is possible to initiate direct connections between hosts behind NATs by using the public node to set the connection up (but have all the data flow directly between two hosts behind different NATs without going through the public host). This thesis will not describe in detail how this is integrated into the larger framework, however the implementation does include this feature.

Before going into details of operation, it is useful to take a look at the basic constraints of the system. Let us say that we have n nodes, and a stream of b bits per second. All of the nodes (except the source, which, let's say is not counted in the n nodes) must receive

the stream. Thus, the total flux of data into nodes must be nb bits per second. In turn, this must mean that the flux of the uploads out of nodes must be at least that figure. If the source has uplink capacity u_s and each node i has uplink capacity u_i, then the inequality $nb \leq u_s + \sum_n u_i$ must hold in order for all n nodes to receive the stream. (To be even more accurate, at least one of the nodes can't be retransmitting in this scenario, but this detail only serves to muddle the inequality.)

In deploying such a system, this inequality would have to be considered, along with the expected uplinks of the clients in order to provide enough capacity at the source, or, alternatively, by supplying high-capacity nodes to help out with the overlay by making up for the expected low-capacity nodes.

2.3 Joining

The most important part of distributing a stream is being able to accommodate as many nodes into the distribution tree as possible. One way of knowing how to connect into the stream is to require everyone to know the source in order to receive the stream content. The big disadvantage here is that the source will be put under unnecessarily high network load since it will be queried each time a node tries to join into the overlay.

Thus the join algorithm needs to be able to handle joins that start their search at any node, not exclusively the source, such that if there is an available spot anywhere in the tree, then the joining node will eventually find it. A rendez-vous infrastructure for finding such nodes in the stream is provided in the implementation, but goes beyond the scope of this

thesis.

In a peer-to-peer system, it is hard to prevent nodes from accumulating knowledge about the layout of the overlay. This case is no different – in order to be able to join in an appropriate place in the overlay, the node must be able to find more nodes given a starting set of nodes. The joining node selects on its own which node it will join to, given that node's willingness to accept a new child.

The willingness to accept a new child is determined by a few factors. First, a node may have a hard limit on resources, e.g. number of children, or total outgoing bandwidth. Additionally, the channel capacity between the joining node and the node already in the tree may not allow for reliable operation, which is a very important fact that needs to be detected before a join is authorized.

In order to join under a node, a bandwidth test must be done. There are many ways to do a bandwidth test, however in this case, we do not want to overload the network and possibly "steal" other connections' bandwidth (with TCP backoff). A light solution that does not incur such problems is to send two packets consecutively, of known size, and to detect the delay between them. So if there is a delay t between two packets of (large, e.g. MTU) size s, then the estimated channel capacity is $\frac{s}{t}$. Testing has shown that while this is not as accurate as a full probe, it does get the capacity to within 15%.

The result of a bandwidth test is that if it passes, the node in the tree gives the potential joiner a ticket which will have to be presented in order to actually join. The joiner tries to get a selection of nodes to test against (by finding nodes topologically close to the node that

it starts off with), and out of this selection, of the nodes which gave it a join ticket, it finds the node with the lowest RTT. It then looks at this node's children, and tests the RTT to these, selects the smallest one, and repeats. It then joins to the one of all those nodes with the lowest RTT. This is done in order to maximize the chances of two nodes topologically close on the Internet to also be topologically close in the distribution tree.

Rejoining to the tree upon disconnection happens in the same way as the initial join, except that while in the tree, each node acquires a bit of local state in terms of information about the tree (a few parents up, and a few children down), and the initial "seed" selection is pulled from this acquired local state rather than a lookup on some initial node.

2.4 Buffering

Unfortunately, rejoining into the tree is an operation that can take some time – up to a few seconds. In this time, some data will be missed. To combat this, each node can store a buffer of the last 5 seconds of data that flows through the tree. To facilitate this, the source splits up data into MTU-sized blobs, and assigns each one a sequence number.

When a node receives a data packet flowing through the tree, it stores the data along with the sequence number in an internal FIFO (that is sized to keep the last few seconds of data). When a node rejoins under a new parent, it asks the parent for all data that the parent has with sequence numbers greater than the last sequence number in the joining node's FIFO.

Also, it is important to fill the FIFO first, and play from the back of it rather than the

front as far as the output is concerned (data should be forwarded down the tree immediately, as previously determined). This allows interruptions to go unnoticed, assuming that they are sufficiently few and short, by playing back the data in the FIFO while the node re-enters into the tree. When the connection is repaired, the buffer module will arrange to refill the FIFO with missed data, in addition to receiving the regularly scheduled "current" data.

This buffering step allows the system to be content-agnostic since there is a decent guarantee of data delivery. The guarantee is not absolute, or even as strong as it is with TCP, but assuming no extenuating network conditions occur, it will flow from end to end.

While this section addresses buffering in the VidTorrent trees themselves, there is more to be said about buffering of the data inside a given node before giving it to the player when there is more than one tree involved. See this discussion towards the end of the next section.

2.5 Multiple Trees

Section 2.2 lists an inequality that must be satisfied in order for every node to receive the stream. However one point that it neglects is the granularity of the uplink. If the stream is of bandwidth b, having uplink capacity $b + \varepsilon$ will not help for $\varepsilon < b$ because with a single tree, you can only redistribute an integer number of copies of the stream.

One way to decrease the granularity of the uplink is to split the stream[2][10] into a number of substreams. This is achieved in conjunction with sequence numbering by sending into the ith tree sequence numbers n s.t. $i \equiv n (\mathrm{mod}\ k)$ if there are k trees/substreams. The granularity is hence reduced from b to $\frac{b}{k}$. We can adjust k arbitrarily to satisfy any minimum

uplink capacity desired in order for a node to become a useful in the overlay.

This approach also opens up distribution of a stream encoded using an MDC which would allow one to not be in all trees and still be able to decode the data. Unfortunately, at the time of writing, there are no MDCs widely used to experiment with. Additionally, this allows us to experiment with different approaches to full stream distribution. A RAID-like approach can be used, such that if one of the trees becomes temporarily unavailable, all the data of the stream still is (subject to the RAID type used). This sort of system would be preferable where absolute guarantee of delivery is required as it eliminates the requirement of a tree to rejoin quickly, but would also incur the overhead associated with RAID.

With a single source splitting up data amongst trees, each node must reconstruct the original stream from the incoming packets. Since each packet contains a sequence number, it is straightforward to reconstruct the stream if the packets arrive via the many trees in order. As the latter condition will rarely be the case, each node must contain a *sequencer* which will take a non-sequential input of packets and product a sequential output, representing the original stream. At this point , we can also remove most of the intelligence from the buffer, in favor of placing it into the sequencer.

There are a few reasons why the packets may be coming in out of sync via the different trees. First, there may have been a failure somewhere higher up in one of the tree structures, and this failure has caused a temporary pause in data transmission in that tree, while the other trees are still going. Fortunately, given how the join algorithm works, there should be no long outages like this, though a temporary out-of-sync condition can occur. This is, as

usual, fixed by having a buffer sized to accommodate the time a rejoin can take.

The second and more problematic case is due to the inherent latency of the system, determined by the number of hops (and their individual latencies) from the source. If a node joins somewhere close to the source in one tree but far away from the source in another, then the data coming in via one of these trees will be relatively constantly ahead of the data coming in via the other tree. For cases where this difference in latency is just a second, there is no problem because the local buffering will take care of it. However one can imagine very large numbers of nodes such that there can be minutes of latency from the source to the "bottom". In such cases, an excessive amount of buffering would need to be done, and instead it is better to simply disconnect/rejoin one of the problem trees and hope for a more similar latency difference as compared to the other trees. This condition is easily detectable when all the trees are well connected but the incoming data has large discrepancies in sequence numbers.

In the general case, the sequencer must keep a buffer of the data that it is given, and output data sequentially per the sequence number. The problem is that a data packet may forever be lost, and so the sequencer would end up waiting for it forever, increasing its buffer and not passing any of the data to the decoder. This means that the sequential condition must be relaxed a little. Instead, it should try to do its best, but, same as with the buffer, it should be resource-constrained. Once the sequencer's buffer grows to a certain number of packets (or, alternatively, a certain span of sequence numbers), it should give up on waiting for whatever packet that it is waiting on and move on to the next one. (If a packet that has

been skipped comes in later, it should be ignored, as outputting it will confuse whatever is receiving the stream even more.)

The amount that the sequencer buffers, as well as where in that buffer it plays is very important. Since we try to assure elsewhere that no data is lost, the sequencer can assume a constant average throughput. However, the buffer must be large enough to account for time spent doing reconfiguration, and other network-related fluctuations, in order to assure that the end output is coming out at a similar rate as that going in.

In order to do this, one must have an idea of the maximum time that it would take to reconfigure everything. This is a very complicated measurement to make theoretically, since it depends on the number of failures in any given tree, as well as the number of trees in which there are failures. This measurement is much more suited to be taken in a simulation with the real code running but in a controlled network environment.

2.6 Implementation

VidTorrent was built on top of a library which provides a very simple but powerful RPC interface. This section will first address the convenient features of the RPC engine, and then a few details of implementation of VidTorrent itself.

The RPC engine is mostly written in C++, with bindings to Python. The RPC spec is written in a language specially developed for the engine, and is compiled by a meta-compiler written in Scheme into C++ and Python bindings (also written in C++). The engine provides the generic aspects of RPC, while the generated files deal with specifics such

as marshalling/unmarshalling of the specific calls that are defined by the interface.

This allows all of the protocol to be implemented in a high-level functional language where more time can be spent worrying about the details of the protocol rather than trivialities like list management and hashes, as these are not the subject of research here, merely computation tools. Using a high-level language also greatly enhances the clarity and readability of the code, without exposing too many of the unnecessary details of doing RPC.

VidTorrent itself is split into a few interfaces that are logically separated but work together to achieve the overall result. There is a metadata interface that allows different nodes to pass various information about the streams, such as number of substreams, expected bitrate, a name, etc. There is a buffer interface which allows nodes to access other nodes' buffers in order to get catch-up information. And finally, there is a tree interface which deals with all of the details of joining and staying in a particular tree. (Note that a node may be in multiple trees.)

Chapter 3

Analysis

3.1 Optimality

We consider optimality of the system on a theoretical level by analyzing worst-case scenarios and making sure that every bit of uplink capacity will be able to be utilized constructively, as well as any situations that may be detrimental to either the mesh or a particular node. Direct calculations are usually impossible, but an analysis of what to look for as well as simulation results are presented.

A few cases will be considered. The first is a big problem that will undoubtedly dog any system that does not use a global algorithm for overlay configuration. If the source can only send to one other peer, and a node joins under it that can't send to any more peers, then the system still works fine. However if another node wants to join in, it can't, even if it can redistribute the stream to more nodes. In order to solve this problem, some sort of algorithm needs to be devised in order to drop a "bad" node in order to let a "good" one join. The case that I described here is simple, but much more complex cases can arise that would be much harder to detect. Such an algorithm is beyond the scope of this thesis.

Another problem that needs to be analyzed is whether it is needed to move nodes around throughout the lifetime of the tree. For example, if a high-uplink node joins somewhere far down the tree, should it be propagated up the tree to reside closer to the source? The answer is not straightforward. If a node has many children, then whatever ill effects occur on the node will also occur on the children. As such, we want to take one of two approaches: either minimize the number of children, or minimize the chances of failure. To minimize the number of children, the node needs to be further down in the tree, to minimize the chances of failure, it needs to be further up in the tree. My conclusion on the issue is that it should be moved up in the tree, since a node with a large number of children is more likely to be a node on a stable connection, and thus is unlikely to fail on its own. Furthermore, it makes sense to have a tree with higher fanout towards the top than towards the bottom since that minimizes latency to the "last" node in the tree due to decreased height.

Upon node failure in a single tree, the node must rejoin into the tree. The time that it takes to rejoin is determined by the length of time that it takes to detect the disconnect, as well as the time to find a new node to connect under. The former is governed by implementation details while the latter is governed by the general algorithm. Finding a new node to connect to depends on the length of time that it takes to consider a node, as well as the probability that a node will be suitable. Also, due to the algorithm that tries to find the lowest RTT node, the depth of the tree is a contributing factor.

A contributing factor to the probability of being able to join under a node is the frequency of leech nodes, i.e. nodes that can not redistribute the stream. In general, it is beneficial to

the tree for these nodes to be as far down as possible. The closer a node is to the source, the lower the chances of a break in the tree from that node's perspective. As such, it is best for nodes that are closer to the tree to have more children, since a break above a node results in failures in all of its subtrees. (Of course, the more children there are, the more nodes will be effected by a failure.)

3.2 Reliability

Reliability in the system is determined by the time it takes to reconfigure the tree after nodes drop out, either gracefully (e.g. application exit) or ungracefully (e.g. network failure). The resulting tree should still exhibit the desired qualities outlined in the previous section.

Furthermore, the reliability of the system is measured by how often (if ever) data is dropped from nodes or even entire subtrees. This can occur if a node takes too long to rejoin a tree and the buffer is gone. Along the same line, we can measure the average fullness of the buffer. If the buffer ever gets low, then that means that we were almost ready to reach the buffer frontier which leads to pauses in playback. While this is not fatal (i.e. no data loss), it is undesirable to the user.

To consider reliability from a node's point of view, there are two causes for drops in buffer levels. First, a parent in a tree dying will cause that tree to not receive data, and second parents of parents in a tree may be dying and causing data from those trees to not be propagated down. In actuality, the two are rather similar, since the effect on data flow from the tree is the same – there is no data. With multiple failures, be it multiple nodes in

a single tree, or multiple trees, or both, this effect can be compounded.

If the system is operating at full capacity all the time (i.e. all the links are saturated), then multiple failures will have adverse effect on the integrity of the data, i.e. packets will be dropped. However if there is residual bandwidth, then upon recovering from a single failure all the data that was missed will be recovered and immediately propagated down the tree at a faster rate than the one it would normally arrive at.

3.3 Method

Two different modes of testing can be used. The coarser one is a framework that allows spawning and control of nodes along with a visualization of the overlay. Every node reports to a central visualizer, which can then in turn either spawn more nodes or kill any particular node. This allows a very simple view of the behavior of the join algorithm, as well as of the multi-tree dynamics. Unfortunately the nodes are usually running on a small number of computers, which are in turn interconnected by high-bandwidth links. This scenario is very far from the real-world one in terms of RTT and bandwidth variation.

Despite its lack of ability to approximate real-world scenarios, this framework has proved to be quite useful. First, it allows one to analyze join/reconfiguration dynamics visually, and directly in result to external stimuli such as killing a node. Further than showing the proof of concept, it allows us to test the system easily by leaving nodes running for a long time and seeing if anything "bad" happens (node failure due to code or algorithm errors, other effects of long-term use). This proved an invaluable debugging tool in the early stages.

The improved framework is one where, in addition to the capabilities of the previous framework, we can also easily control the link properties between any two nodes, in terms of both RTT and bandwidth capacity. Specifying each one of these manually is rather tedious, but it is possible to automatically assign them under some realistic real-world distribution. This allows us to provide an arbitrarily accurate representation of real-world use. We can have nodes that are constrained, e.g. home users as well as unconstrained nodes, such as infrastructure nodes or well-connected users.

The final question is what sorts of distributions of RTT and bandwidth should be used. One way to do this is to place nodes randomly onto a Cartesian plane, and determine RTT based on the distance between them. The trees built by the VidTorrent join algorithm should be spatially related, in that we should ideally see a minimum spanning tree of the fully connected graph of nodes with weights proportional to the RTT (assuming no bandwidth restrictions). However due to a current lack of a balancing infrastructure, the tree will be heavily influenced by join order. We can also demonstrate clustering effects by creating a distribution of nodes on the plane with pre-determined "hot spots" around which nodes are created. This should lead to a tree with very few connections between the hot spots.

Further, once this framework is developed, we still need to determine the health of the system, not only the connections made during steady-state operation. A number of factors affect reliability, as described above, and these all need to be evaluated. Changes in these values also needs to be assessed in response to various changes to the system.

The nodes must be randomly placed on a 2D plane, as detailed earlier in this chapter, and

information from them was collected every second. Since we use TCP for all transmission, inserting link losses does not affect the integrity of the transmissions, but is equivalent to having a higher RTT/lower bandwidth. The RTT can determined by the Cartesian distance between the nodes. Constraining the bandwidth is also an interesting effect, though it is quite tricky to implement. Constraining the RTT is almost equivalent to adjusting bandwidth for most settings, since even though a packet may travel at an arbitrary rate, you can only send packets so big and so often. The resulting simulated network has the property that data comes in bursts, instead of at a steady rate as would happen in a bandwidth-constrained system.

The number of children, as well as the fullness of the buffers should be collected for each tree from each VidTorrent node. The number of children allows us to measure the average out-degree, while the fullness of the buffer allows us to monitor recovery effects. Furthermore, the settled topology of the constructed trees is recorded and analyzed for problems. At each node we can also compare the difference in overall delays for each tree that the node is in. This difference is what determines the buffer length that the sequencer has to maintain in order to be able to reassemble the stream without dropping any packets.

3.4 Last-hop IP Multicast

While this system purely relies on unicast IP infrastructure, it would be a nice addition to add IP multicast for last-hop distribution. This would allow just one node on any native multicast enabled network to be in the VidTorrent tree, while the rest of the local nodes

would receive data over multicast from the it. This in turn presents problems in data transfer reliability which are beyond the scope of this thesis. This problem is nicely addressed in Dimitris Vyzovitis's Masters thesis[11]. The advantage of using this technique is that the local nodes do not need to know anything about VidTorrent, and that using native multicast is much more efficient than sending individual unicast streams.

Chapter 4

Conclusion

This thesis set out to explore some of the ways in which we can perform application-layer single-source multicast for live stream distribution, subject to many of the real-world constraints of the Internet. In particular, care was taken to not rely on inexistent technology (such as MDCs), or in fact, on any particular method of encoding. This system remains content-agnostic, but is still able to take advantage of diversity through splitting of the stream.

In relation to these, the problems of implementation were discussed and analyzed, and a testing framework was proposed. Furthermore, an implicit argument was made throughout the paper that the system can scale arbitrarily, and as long as constraints are met, anyone is able to use it to distribute live content at minimal cost.

Bibliography

[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. Technical Report TR-2002, UMIACS, 2002.

[2] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. 19th ACM Symposium on Operating Systems Principles, 2003. http://www.cs.rochester.edu/sosp2003/papers/p159-castro.pdf.

[3] P. Chou, V. Padmanabhan, and H. Wang. Resilient peer-to-peer streaming. Technical Report MSR-TR-2003-11, Microsoft Research, March 2003.

[4] Y. Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an internet broadcast system based on overlay multicast. USENIX Annual Technical Conference, June 2004.

[5] B. Cohen. Incentives build robustness in bittorrent, 2001. [Online]. Available: http://www.bittorrent.com/bittorrentecon.pdf.

[6] S. Deering and D. Cheridon. Multicast routing in datagram networks and extended lans. *ACM Transactions on Computer Systems*, 1990.

[7] H. Eriksson. Mbone: The multicast backbone. *Communications of ACM*, 37(8):54–60, August 1994.

[8] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. USENIX Annual Technical Conference, 2005. http://www.brynosaurus.com/pub/net/p2pnat.pdf.

[9] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. 19th ACM Symposium on Operating Systems Principles, 2003. http://www.cs.rochester.edu/sosp2003/papers/p183-kostic.pdf.

[10] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasability of supporting large-scale live streaming applications with dynamic application end-points. Proceedings of ACM SIGCOMM, August 2004.

[11] D. Vyzovitis. An active protocol architecture for collaborative media distribution. Master's thesis, Massachusetts Institute of Technology, June 2002.

[12] L. Zhang, S. Deering, and D. Estrin. RSVP: A new resource ReSerVation protocol. *IEEE network*, 7(5):8–?, September 1993.