Multicast Streaming with SplitStream

Jon Ludwig
Department of Computer Science
Rochester Institute of Technology
Email: jcl1827@cs.rit.edu

Brad Israel
Department of Computer Science
Rochester Institute of Technology
Email: bdi8241@cs.rit.edu

***Abstract*—Streaming multimedia systems are an important area of research in distributed systems because of the large bandwidth and low latency requirements. In this work we propose a content distribution system which utilizes scalable overlay multicasting algorithms to efficiently publish large amounts of content to many subscribers. We present a new, light-weight implementation of the Pastry overlay network which provides a scalable peer-to-peer overlay for addressing communication. A light-weight implementation of Scribe and SplitStream is also developed which build upon Pastry to provide a scalable multicast infrastructure. Erasure codes are used to provide reliable transfer in cases of network failures.**

I. Introduction

Streaming multimedia has become a major networking task in recent years. It presents many challenges for distribution including high bandwidth and low latency requirements. Multimedia typically consists of audio and video which, while often compressed, require relatively large amounts of bandwidth. With the increase of high bandwidth connections made available to consumers, demands for high fidelity audio and video have also increased. This puts higher demands on content distributors to make high-fi content available to many consumers. In addition, audio and video are often consumed in real time or near real time, making latency a major issue.

In this work we present a system that is fit for distributing content throughout a content distribution network and for distributing content to large number of end-users. This system provides low packet duplication which reduces link stress and overall bandwidth consumption while maintaining properties which provide low latency. The system is logarithmically scalable to a high number of nodes and also functions well with a low number of nodes.

This work is based upon the work of [6], [8], [1]. We present a new, light-weight implementation of the Pastry, Scribe, and SplitStream systems which provide scalable content distribution via a peer-to-peer overlay network. Section II presents the issues with streaming multimedia and multicast technologies, Section III describes the overlay algorithms developed for this work, and Section IV discusses the use of erasure codes to provide fault tolerance.

II. Streaming

The problem of streaming multimedia, or any other content, is a problem of delivering a relatively large amount of information to a possibly large number of users in a timely fashion. Traditionally, one way to do this is to open a separate connection to each of the consumers and stream the content separately to each of them. This presents many problems, including a high degree of link stress on links, especially those near the source. There is a large amount of packet duplication and the number of connections scales linearly with the number of users, which may become infeasible for a large number of users.

This technique can be improved by only duplicating packets when they need to be sent out on different links. This severely reduces link stress on links near the source since the source is only sending out a single packet for each piece of information that should get to all users. This type of system naturally creates a spanning tree of the users within the network, a so called multicast tree.

One way in which a multicast system can be implemented is at the network layer, with IP multicasting. In this system the network routers are responsible for maintaining group membership and distributing packets correctly. This has the advantage of being efficient and creating optimal trees, however, there are several drawbacks. Implementing IP mulicasting means upgrading or replacing the routers in the entire infrastructure which may be very costly. Extra burden is placed on the routers to manage group information and distribute packets correctly. IP multicasting may require globally unique addresses for each multicast group. Additionally there are a host of non-technical problems which arise including router management and billing issues.

The solution we propose in this work is to utilize a peer-to-peer overlay network to perform multicasting. This avoids all of the problems associated with modifying the current infrastructure. The main problems with an overlay network are link stress and latency, which are very important for streaming multimedia applications. Our implementation of Pastry, Scribe, and SplitStream provide a scalable overlay network for multicasting with properties which mitigate link stress and latency.

While Pastry provides the underlying routing mechanism and Scribe provides the ability to multicast data, SplitStream addresses the problem of unfairness in multicast trees. The problem is that nodes which are interior nodes (e.g. not leaf nodes) in the multicast tree bear an unfair burden of having to forward content, while the leaf nodes do not. SplitStream constructs multiple trees and ensures that each node is only an interior node in one tree, therefore distributing the forwarding burden more fairly. It also addresses the problem of slow links

by allowing nodes to get pieces of the stream from multiple other nodes. Figure 1 illustrates the problems of a traditional multicast tree. We also utilize FEC codes which allow data to be reconstructed from multiple trees, even if a fraction of those trees are disabled.

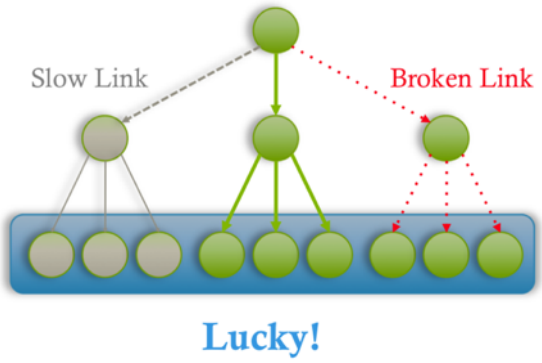

Fig. 1. Problems with traditional multicast trees

III. Overlay Networks

Overlay networks provide many desirable properties for a streaming multimedia distribution network mentioned in Section II. Here we discuss the design of Pastry, Scribe, and SplitStream, which are the core technologies in the overlay network used in this work. A light-weight implementation of each of these systems has been developed based on the works of [6], [7], [3], [8], [1], [2].

A. Pastry

Pastry is a scalable peer-to-peer system for addressing and communication between peers. A Pastry "ring" is formed by by randomly assigning nodes an address from a uniformly distributed address space. Each node in the overlay is aware of $O(\log(n))$ other nodes in the network, and messages take an average of $\log(n)$ hops from source to destination. This means that the system is scalable both in terms of resources consumed on each node and latency of message passing.

The assignment of node identifiers in a Pastry ring is commonly accomplished using a cryptographic hashing function to uniformly choose a unique node identifier with minimal collisions. The parameter n controls the number of bits used for each node ID, resulting in 2^n possible node IDs. The node IDs are interpreted as base 2^b for purposes of the routing algorithm, where b is a parameter. Each node maintains a routing table of other nodes in the system as shown in Figure 2. Each column of the routing table represents a digit $0 \ldots 2^b$. The first row contains entries for other nodes which have a node ID whose most significant bit matches the digit for that column. For example, in the first row, the 3rd entry will have a most significant bit of 2. The entry for the digit which matches the current node's ID is left blank. The second row contains entries whose most significant bit matches the current node's most significant bit, and whose next-to-most significant bit matches the column number. In this way the routing table is constructed, such that for row r and column c the entry will have digit r have a value of c and for all digits less than r their value will match the corresponding digit in the current node's ID.

When a node wishes to route a message to another node it first determines how long of a prefix their node IDs share. This will indicate which row in the table to look at. For example, if a node $65A1x$ wants to send to $65B2x$, where x is any arbitrary suffix, it will look at row 3 since all nodes in this row start with $65x$ and differ in the 3rd bit. The entry for column B will be a node whose node ID starts with $65Bx$, which may or may not be $65B2x$, however in this way Pastry can route to a node which has a closer node ID until it reaches the destination node. With proper routing tables each message should take a maximum of 2^b hops since each node will get the message to a node with at least one digit more correct. With $N = 2^n$ possible nodes, a message will take $\lceil \log_{2^b} N \rceil$ hops.

Each node also keeps a leaf set of nodes whose node IDs are numerically close. In this way if a node does not have a proper routing table entry it can route to a closer node by looking in its leaf set. This guarantees message delivery unless $l/2$ nodes fail from the leaf set, where the leaf set has $l/2$ nodes whose IDs are less than this node's ID, and $l/2$ nodes whose IDs are greater. Figure 3 shows an example of how multiple nodes will route to a node with ID $E7A9$.

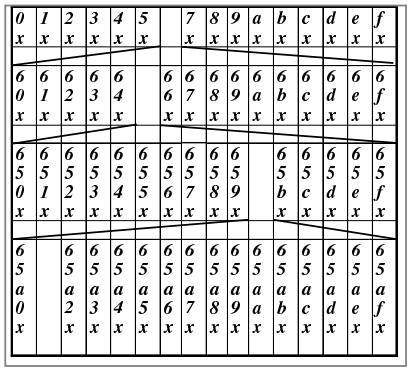

Fig. 2. Routing Table of a Pastry node with ID $65A1x$

When a node wishes to join a Pastry ring it must know the address of at least one node in the ring, but it may be any node. The node joining is referred to as the joining node, and the known node is the bootstrap node. The joining node sends a $JOIN$ message to the bootstrap node requesting a randomly generated node ID D. Then the bootstrap node uses the Pastry routing mechanism to route a message to the node whose ID most closely matches the requested node ID. If the ID matches, then the bootstrapping node must generate a new ID and try again. For each hop that the $JOIN$ message takes from the bootstrap node to the destination node, each intermediate node sends back the appropriate routing table information to the joining node. If an intermediate node shares a prefix of length 4 with the joining node, then the intermediate node sends its

first 4 routing table rows to the joining node; these rows will be valid. The destination node then also sends its leaf set to the joining node, since the joining node's ID is closer to the destination node's ID than any other node, the leaf set will be valid as well. In our implementation, when the joining node receives these updates, it sends an acknowledgment back to each node. If a node's routing table has changed after it sent information to the joining node, but before it received an acknowledgment, it resends the information. This prevents the joining node from receiving invalid or old information. If the node's information has not changed it sends an ACK-ACK back to the joining node. When the joining node has received and ACK-ACK for each ACK it has sent out, it knows it has a complete set of routing information and has joined the network. At this point the joining node sends an update that it has joined to all nodes in its routing table and leaf set, who consequently update their routing information to reflect this new node.

Nodes update their routing information when they receive a notification that a new node has joined. If a node already has a corresponding entry in its routing table, it chooses the node with the lowest latency to occupy that entry. In this way the routing table maintains a list of the nodes with the lowest latency. This latency measurement allows Pastry to display a property of convergence. A low latency node will often occupy the appropriate routing table entries in many nodes. Thus when these nodes route to a node close to the low latency node, the messages will tend to converge on the low latency node. This property allows Scribe to generate efficient multicast trees.

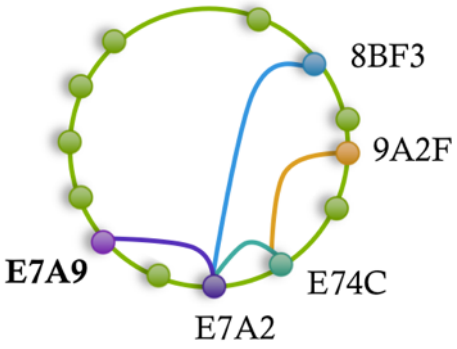

Fig. 3. Pastry Routing Example

B. Scribe

Scribe uses Pastry's routing mechanism to construct multicast trees. To construct a multicast tree Scribe generates a group ID by hashing a topic or some data that is known between all nodes that wish to join the multicast group. It then routes a $SUBSCRIBE$ message to the node with the ID closest to this group ID. Each intermediate node that receives a subscribe message joins the multicast tree and adds the node it received the message from to its list of children for that topic. As discussed previously, the convergence property of Pastry's routing mechanism means that messages from different nodes will often converge on the same node. This typically results in an efficient tree.

Any node may route a $PUBLISH$ message with some content to the root of the tree, who will then send this message to its children. Each node does this until the published content is disseminated throughout the entire tree. Figure 4 shows an illustration of how this works.

In this system we also implemented a push-down mechanism which allows nodes to limit the number of children they have. When a node receives a $SUBSCRIBE$ request and has a maximum number of children already, it send back a list of its children to the subscribing node, who chooses a new parent and routes the $SUBSCRIBE$ message through that node. In this way a node may "push down" a potentially new child to become a child of one of the node's current children.

C. SplitStream

SplitStream builds upon Scribe's mulicasting abilities by generating multiple multicast trees. SplitStream generates multiple Scribe groups, each differing in the most significant bit. If the routing tables of each node are properly constructed the first hop will immediately route to a node whose most significant bit matches that of the Scribe group. In this way the network is partitioned by the most significant bit of the node ID. All nodes who join a tree starting with $0x$ will route directly to a node whose ID starts with $0x$ and then progressively get closer to the actual group ID, but never will the message be routed to a node with a different most significant bit. This ensures that for the tree corresponding to the group ID starting with $0x$, only node's whose ID starts with $0x$ will be interior nodes of this tree. Figure 5 shows what one SplitStream tree might look like, while Figure 6 shows how a ring might be organized into 2 SplitStream trees.

Fig. 5. One SplitStream Tree

IV. FEC Codes

Forward Error Correction, FEC, codes are the technique that we used in our project to provide robustness to packet delivery in the splitstream network. The basic idea of the FEC technique is that the server sends redundant packets through

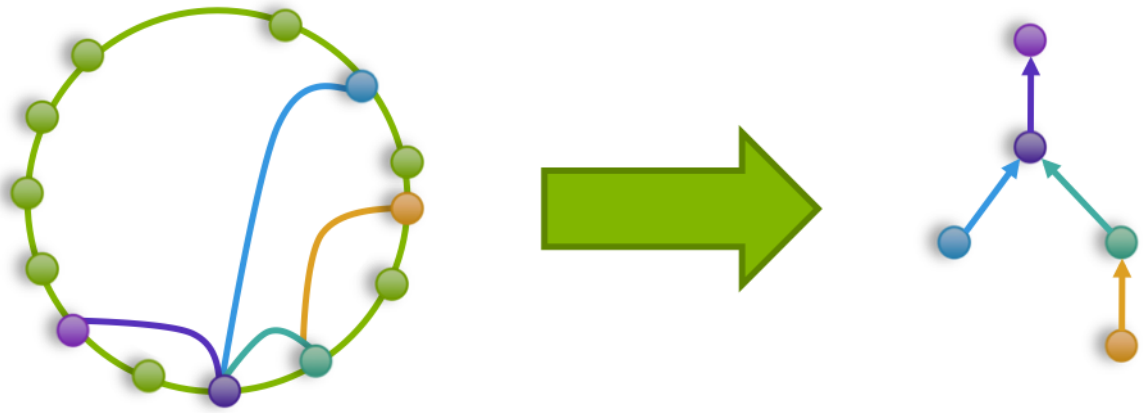

Fig. 4. Generating a Scribe Tree

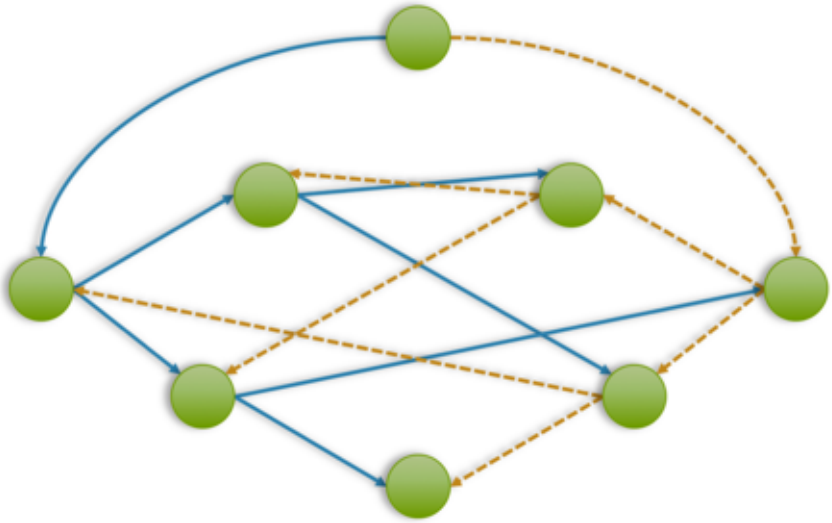

Fig. 6. Two SplitStream Trees

the network, which allows the clients to lose a certain amount of packets and still be able to reconstruct the proper packet. The library we used in our project [4] was based on Rizzo's paper [5] that describes how to FEC techniques for reliable packet transmission. Rizzo's technique takes a k sized block of data and encodes the data so that it becomes n size, where n is larger than k. This is denoted a (n,k) code. The new encoded packets each contain a certain amount of redundant information so that any client that receives at least n-k packets can decode the received packets to produce the original k sized block of data. An example of this is shown in Figure 7

The exact process of encoding and decoding the data is described in Rizzo's paper [5]. The basic overview of how encoding is handled, is that the data is placed into a matrix and uses linear algebra techniques to expand the matrix from size k to size n to create an encoded matrix, G, of size n x k. The decoding process is the linear algebra equivalent of reversing the encoding algorithm and requires some extra information, namely the row number of the matrix G that the received data was originally on. For our implementation purposes, this means that all of our packets needed to contain the data's original index in the encoded n sized block of data.

For our project, we used the library to initialize a (4,2) FEC code on both the server and all client. A (4,2) code means that every MP3 data block is encoded to be four data blocks and then sent over the network by the server. Using splitstream, the server will send each of the four blocks to a different head node in the four trees. The client side can then receive any two blocks to reconstruct, or decode, the original data block and play that piece of the song, the other two blocks could have been lost or corrupted. This is important to our project because when interior nodes are added or removed from a tree, the tree has to regenerate and can cause packet loss. If a tree is regenerating or fails, the nodes that are supposed to be in the tree still receive packets from the other three trees and can continue to decode and play the song even though

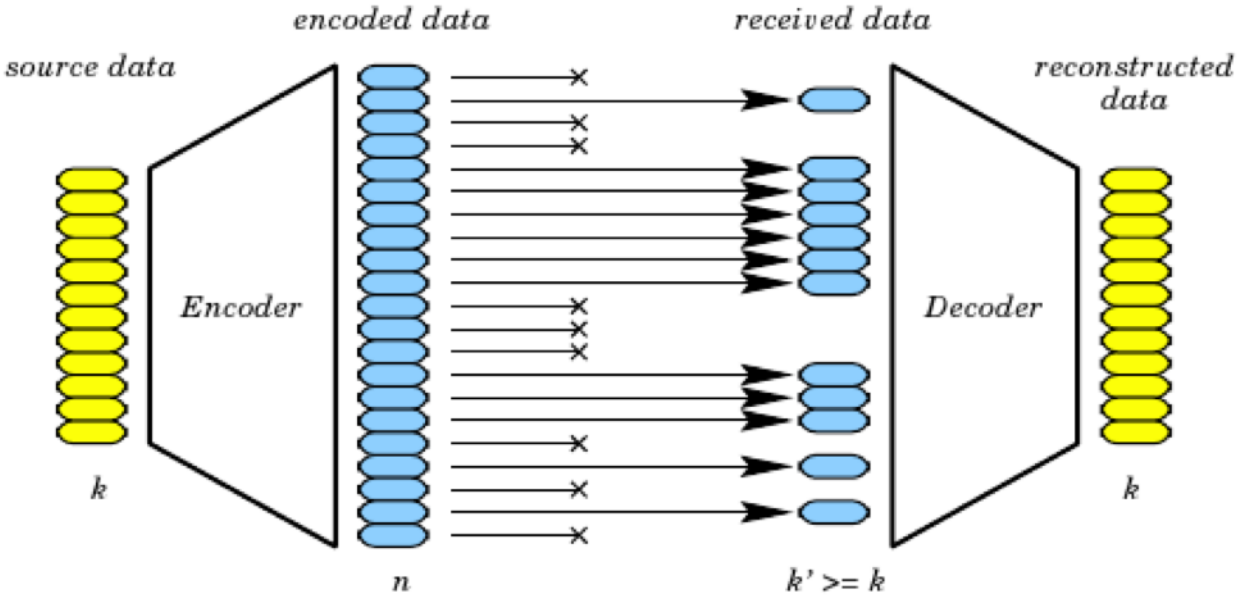

Fig. 7. FEC Codes

they are not receiving packets from their parents in one tree. This increases the level of robustness that is needed, especially for streaming applications. In our implementation, we send out each data block inside of an object that also contains the block's index for FEC decoding and also the index into the song. This allows the client to decode packets and place them in the correct order to be played by our media player. The advantage to using FEC codes, other than the robustness, is that the algorithm can encode and decode anything that can be represented by a byte array. In our case, we were streaming music, but it would be very easy to extend the code to stream video or even files across the splitstream network in a reliable manner. Also, if a project required more or less robustness, it is as simple as raising or lowering the n value and k value accordingly. If the data transfer needs to happen no matter what, the client and server could create a (4,1) code so that the server sends out 4 packets, but the client only needs to receive 1 of them in order to decode the packet, and the same goes for a less robust (4,3) code where the client would need to get any 3 out of 4 packets. This also allows the number of splitstream trees to expand easily, so if the number of trees increases to 16, the server and clients can, for example, create (16,4) codes which allows the server to continue to send each encoded piece to a different tree. The disadvantage of using FEC codes is that when redundancy is added, it creates more traffic overhead. The designers of a project like ours need to take into consideration the tradeoff between packet overhead and robustness.

V. Conclusions

This work successfully demonstrates that the combination of Pastry, Scribe, and SplitStream can form an overlay network that is capable of efficiently streaming media to a large group of hosts. It also adds in FEC coding to ensure that the media will be transmitted reliably to the clients, even during SplitStream tree reconstruction and other network errors. Our lightweight implementations of the overlay network protocols allow for a simple way to test the network and view the tree structures to gain a greater understanding of how SplitStream networks work and how data is passed through them. The project is also very extensible and would provide a great starting point for any future projects that were looking to push the boundaries of what SplitStream is capable of.

References

[1] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Content Distribution in Cooperative Environments. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 292–303, 2003.

[2] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313. ACM New York, NY, USA, 2003.

[3] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[4] Onion Networks, Inc. Onion Networks. Web page, 11 2008. Available from World Wide Web: http://onionnetworks.com/developers/.

[5] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997. Available from World Wide Web: http://www.acm.org/sigs/sigcomm/ccr/archive/1997/apr97/ccr-9704-rizzo.pdf.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes In Computer Science*, 2218:329–350, 2001.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[8] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Networked Group Communication: Third International COST264 Workshop, NGC 2001, London, UK, November 7-9, 2001: Proceedings*. Springer, 2001.