Zebra: Peer To Peer Multicast for Live Streaming Video

Maya Dobuzhskaya, Rose Liu, Jim Roewe, Nidhi Sharma
{mdob,rliu,jimr,nidhi}@mit.edu

May 6, 2004

Abstract

The high bandwidth requirement for serving live streaming video and audio restrict residential Internet users from distributing content to multiple viewers. In a client-server model, the same data is simultaneously sent to all viewers. Instead of the server sending data independently to clients, Zebra uses a peer-to-peer multicast scheme that redistributes the serving load among the clients. This shift dramatically reduces the server's resource requirement, enabling low bandwidth sources to serve high quality live media to up to 100 clients. Zebra is resilient to client failures and takes advantage of locality to reduce network traffic.

1 Introduction

Individuals use the Internet to express themselves through daily web-site logs, audio broadcasting, and web cams. However, live video, the most expressive form, is not currently possible due to the large bandwidth requirement. Clear, smooth video requires approximately 100 kbps of upstream speed. Even with a high speed DSL or cable connection of several hundred kbps, the source can only send the video to a couple of people.

1.1 Background

Commercial content providers use the traditional client-server model to distribute live streaming media. For instance, a Shoutcast [1] server receives audio content from an individual and distributes it to thousands of clients on its high bandwidth connection. The drawback of Shoutcast is the cost to the source individual, who has to pay for this distribution service. In principle, a scheme similar to Shoutcast can be applied to serving live streaming video. However, the high bandwidth requirement for serving video would significantly increase the cost to the source individual.

For live media, all clients receive the same content at the same time. The ideal solution for simultaneous data distribution is IP multicast. In this model, the server sends out one copy of each packet, which is duplicated by routers to reach clients on different paths. Unfortunately, multicast is not available because of difficulties in implementation and agreements between ISPs. The only viable alternative is end-layer multicast [2].

Application-level multicast [3] distributes the responsibility of disseminating data to the clients, in a peer-to-peer manner. SplitStream and P2PCAST use application-level multicast in cooperative environments to distribute high-bandwidth content [4, 5]. These systems are intended for 1000s of nodes and require complex lookup schemes such as Chord [6] or Pastry [7]. For an individual hosting content, the target audience will be much smaller, allowing a more tightly controlled, easier to use system.

1.2 Proposed Solution

Zebra is designed to support about 100 clients. The source server only needs to send one complete copy of the data, which the clients then distribute amongst themselves. These clients can be managed by a single coordinating entity, allowing for better control over the entire system. Because the video source is already a single point of failure in the system, making the coordinating entity share this fate does not decrease the fault tolerance of the system.

1.3 Goals

For our system to be useful, it should have the following features:

- *Functionality.* All clients must receive the full media stream.

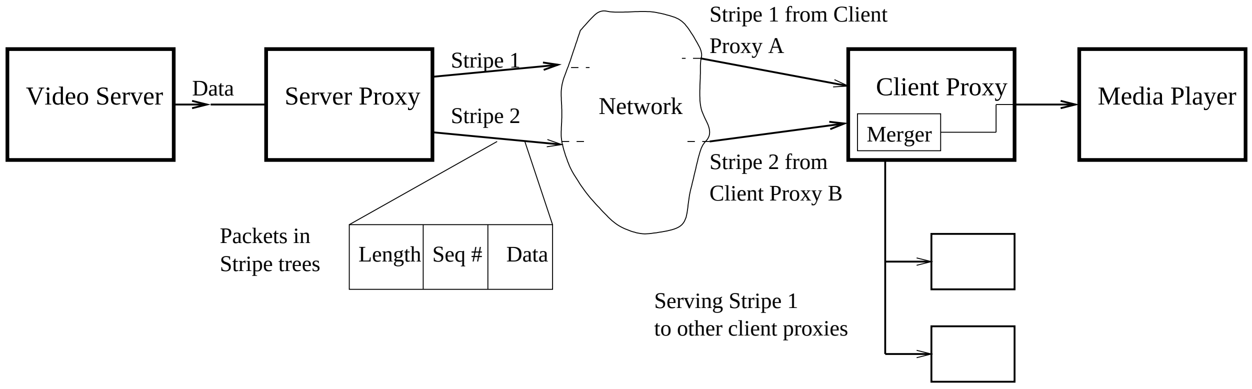

Figure 1: General system layout.

- *Robustness.* The system should be resilient against client node failures.
- *Network Traffic.* Overall network traffic should not increase and should decrease if possible.

Our system divides the constant stream of data into stripes to improve performance and robustness. In a peer-to-peer system, the stream of data is disrupted whenever a client leaves the system either due to a failure or a regular disconnect. Since clients receive pieces of the content from different senders, they can continue to receive some data even if one of the senders disconnects.

To further hide disruptions from the user, a client keeps a buffer of several seconds of data. When a client reconnects after being cut off from a sender, the buffer allows the video to play smoothly as the client catches up on the data that was missed during the disconnection. In a one-directional live video streaming system, it is allowable for the video to be viewed a few seconds after its creation.

Since video bandwidth is large, a naive distribution tree can lead to higher network traffic by sending data to further clients instead of closer ones. In order to reduce network traffic, Zebra estimates the network distances between clients to build a distribution tree that increases locality within the system.

2 Design

2.1 System Structure

Our system is divided into two parts: the server proxy and client proxies. We use proxies in order to make our system easily portable to different video servers and media players. As shown in Figure 1, the server proxy sits between the video server and client proxies. The client proxies sits between the client media players and the server proxy.

Data is sent using a peer-to-peer multicast scheme with striping. The video server sends one copy of the data to the server proxy, which forwards this data to two client proxies. These client proxies then distribute to other client proxies and their respective media players. In the rest of this paper, we use the term node to refer to a client proxy.

2.2 Data Distribution - Striping

The server proxy divides the video data-stream into two stripes, which it sends down different distribution trees (see Figure 2). A node serving in one distribution tree must be a leaf in the other tree. This constraint maintains the invariant that each node can only serve data in one stripe. Therefore, if a serving node dies, its children still receive data on the other stripe, provided that the children's second parents do not simultaneously disconnect.

In order to maintain certain nodes as leaves in a distribution tree, a new node assigned to serve may need to splice in front of a leaf or another serving node, as shown in Figure 4. Splicing will only cause a brief disconnection in the tree because all nodes involved are known before the disconnection.

2.3 System State

Since our system is designed to only support around 100 nodes, the server proxy can maintain full system

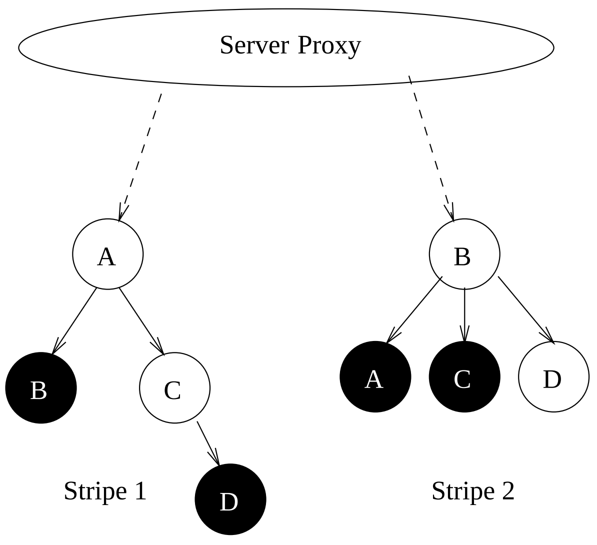

Figure 2: The two stripe distribution trees.

state. It maintains state about the distribution tree for each stripe, including which nodes are connected to each other. It also maintains specific information about each client in order to provide information to requesting clients upon connection.

Client proxies are responsible for updating the server proxy when important events occur. Important events include client connections and disconnections. When a new client connects to the system, it informs the server proxy of its parents. When a client node fails, its immediate parents and children notify the server proxy.

2.4 Connection Protocol

All clients wishing to connect to the system must first communicate with the server proxy. The server proxy determines which stripe the new client should serve, based on the number of nodes currently serving each stripe. It tries to keep the number of nodes serving each stripe approximately equal. If more nodes serve one stripe, then the other stripe would not contain enough serving nodes. The server proxy also searches through the distribution trees of each stripe to find potential parent nodes for the new client.

For each stripe, the server proxy returns up to ten potential parent nodes selected at random. These ten nodes are divided into two categories: available nodes and splice-able nodes. Available nodes are capable of serving at least one more child, while splice-able nodes may not be able to support an extra child. The system does not have enough information to be certain of the splice-able node's serving capacity, so it tries to add an extra node, and monitors the children's data rate. Therefore new connections to splice-able nodes either require splicing or forcing a splice-able node to serve another child.

A new client always tries to connect to the closest available node in a stripe. If no available nodes exist, the new client connects to the closest splice-able node. In the stripe it is assigned to serve, a new client connecting to a splice-able node splices between the splice-able node and one of its children. In the stripe in which it must be a leaf, the new client forces the splice-able node to become its parent. If the new client node detects that data is getting sent at a reasonable pace, then the system records that the parent node can serve one more child than predicted. However, if the added stress of an extra node causes the connection to be slow, the new node should disconnect and try to reconnect to the system.

Once the new node has established a connection to both parents, it informs the server proxy of its parents. The server proxy updates the state of each parent node and creates a state for the new client node.

Zebra initially assumes that each node can send as much data as it receives (two stripes worth of data). If this assumption holds, then there would always be available nodes in each stripe. However, since our system supports clients of various bandwidths, there might exist nodes that can serve less than two children. Zebra compensates by having clients with higher bandwidth serve more than two children. Since it is impossible to accurately determine the total number of children a node can support, it is necessary to evaluate this node characteristic by asking nodes to serve extra clients when needed.

When there are no available nodes in a stripe, Zebra should force extra children onto nodes, and have a way of evaluating the arrangement. One possible scheme requires children nodes to detect the rate at which they receive data, and determine if their parents are fit to serve additional children. In the case of a new connection, if the new child determines that it is receiving data too slowly, it can find a new parent and tell the server to decrement the number of children the old parent can serve.

Lastly, in a system that supports nodes who serve less than they receive, it is possible for the system to become temporarily full. However, since arbitrary clients may disconnect at any time, new clients can attempt to connect until they are successful.

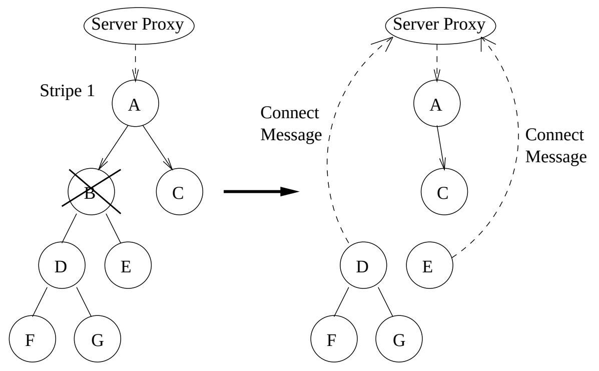

Figure 3: Disconnection and reconnection.

2.5 Disconnection Protocol

A serving node disconnecting from a distribution tree also disconnects its immediate children from the tree. If these children are also serving, the disconnection causes the subtree to lose service on that stripe. If all disconnected nodes individually contact the server proxy to re-establish connection, the server proxy may be flooded with requests.

To reduce reconnection overhead, Zebra requires the immediate children of a failed node to keep their respective subtrees intact. Only the immediate children contact the server proxy for reconnection (see Figure 3). They inform the server proxy of changes within the distribution tree (i.e. which nodes are detached). The server proxy temporarily keeps the state of disconnected nodes and their subtrees in case they attempt to reconnect.

If the root of a disconnected subtree requests to reconnect when there are no available nodes in a stripe, it has to splice. Unlike the case of a new client splicing, this reconnecting node already has children. Therefore, splicing forces this node to support an extra child. If it cannot support this extra child, then its children could detect slower service and the extra child should reconnect in that stripe.

3 Implementation

Zebra's server and client proxies extend the Real Networks'[8] application level RTSP proxy for UNIX. This section describes the implementation of the salient features of our system; namely, communication between system entities, striping, splicing, maintenance of server state, and locality in the connection protocol.

3.1 System Communication

The video server, proxies, and media players all communicate through messages sent over TCP. Zebra employs the RTSP protocol [9] to communicate between the server proxy and video server, as well as between a client proxy and media player. In addition, Zebra has separate messages for communication between proxies. As shown below, each of these messages is composed of a length, followed by header-value pairs.

```
00054
"MessageType":"ListOfAddresses"
"StripeToServe":"Stripe 1"
```

3.2 Striping

The server proxy splits data it receives from the video server into two stripes. It appends a length and sequence number to each packet, and sends it over TCP to the root node for each stripe. Odd packets are sent on one stripe and even packets on the other. The sequence number is incremented with each stripe segment sent. Client proxies use this number to reconstruct the total packet order, and to figure out which packets need to be forwarded on to children.

Upon receiving data from its parents, each client performs two tasks: forward the data from one stripe on to its children, and merge and send data to its media player (see Figure 1). The client proxy forwards packets to its children as it receives them. If packets arrive in order, it also forwards them to the media player. If they arrive out of order, the client proxy reorders them according to the sequence numbers. Furthermore, the client proxy timestamps incoming packets and waits for missing packets only until a preset time interval expires.

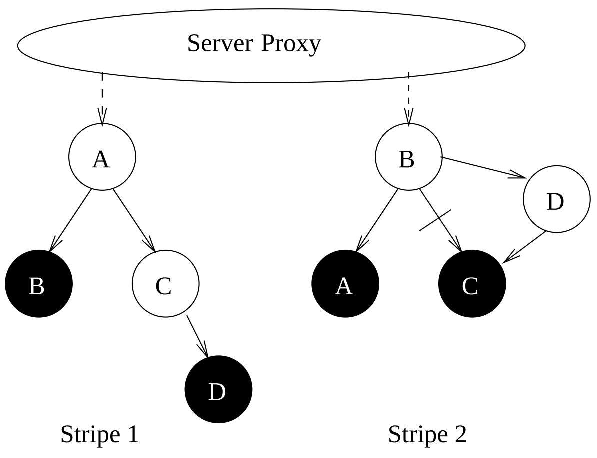

Figure 4: The splicing process.

3.3 Splicing

Splicing is required when there are no available nodes for a client to connect to within the stripe it should serve. In Figure 4, node D is a new client that has been assigned to serve in stripe 2 to keep the number of serving nodes in each stripe balanced. While node C can serve stripe 1 to D, there are no available nodes to serve stripe 2 because node B is serving at its capacity and nodes A and C have to remain as leaves within this tree. Therefore D has to splice after node B in the distribution tree for stripe 2.

Upon splicing, D sends a message to B indicating that it will splice behind B. The message contains D's IP address and message port number. B accepts D's connection and then randomly chooses one of its children to reconnect to D. In Figure 4, B sends C a message to reconnect to node D, and then B closes its connection to C. Upon receiving this message, C reconnects to D, who is now receiving stripe 2 from B.

3.4 Server State

The server proxy maintains the stripe distribution trees, lists of disconnected nodes, and some client state.

In particular, it manages the following state for each stripe:

- The root node of the distribution tree.
- A list of disconnected nodes and their subtrees.
- The number of nodes currently serving in that stripe.

The server proxy manages the following state for each client node:

- The client's IP address and port number for messages.
- The stripe it serves.
- The additional number of nodes it can serve.
- A list of children nodes.

3.4.1 Stripe Distribution Trees

Since the server keeps track of the root nodes in each stripe distribution tree, it can traverse down a tree to lookup any client within the system and update its state. When a new client contacts the server proxy, the proxy searches down the trees to find available and splice-able nodes. The server proxy also traverses and modifies these trees when notified that a node has connected or disconnected from the system. In the disconnection case, the server proxy puts the immediate children of the disconnected node and their subtrees, in the list of disconnected nodes for a stripe. If these children reconnect, the are removed from the list and their state is reintroduced into the distribution trees.

3.5 Locality in the Connection Protocol

Zebra tries to reduce the network traffic it generates by connecting clients who are closer to each other. When a new client receives a list of available or splice-able nodes to connect to, it sends five ping packets to each client and chooses a parent by picking the lowest average round trip time. Sending five packets provides a reasonable estimate of round trip times without causing additional overhead of sending many packets. Since sending five ping packets takes a few seconds (4-5 seconds) our implementation pings multiple client nodes in parallel. The client proxy forks a new process for each client node it pings and the parent process parses each ping output to find the client with the lowest average ping time.

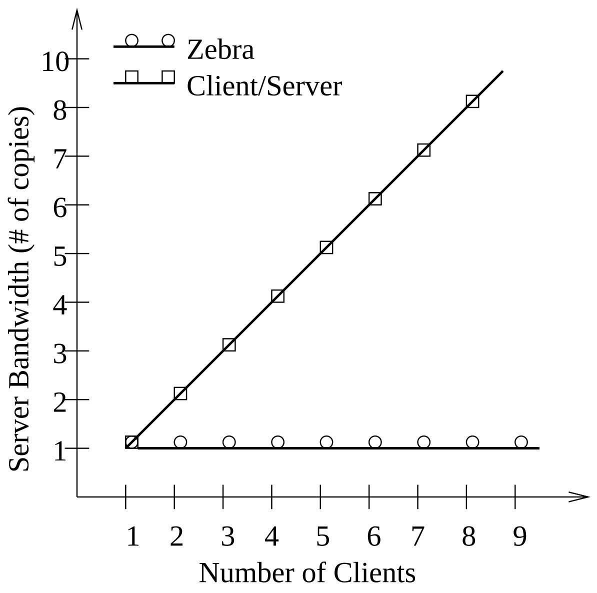

Figure 5: Server bandwidth requirement for Zebra and client-server systems.

4 Evaluation

Zebra reduces the required bandwidth for the server since it only needs to serve a single copy of the data. As shown in Figure 5, the amount of data a Zebra server sends out remains constant with an increasing number of clients. On the other hand, in a traditional client-server system, the amount of data distributed by the server increases linearly with the number of clients.

Zebra is successful in sending video data to all clients, satisfying our functionality goal. It has been tested for up to 10 clients. Due to resource and time constraints we were not able to more aggressively test our system. However, in terms of protocols, adding clients at this point is not different from when the 5th or 6th client was added. Since the only message overhead in the system is during transient times when clients enter or leave, the system design should scale without trouble up to 50 or 100 nodes. Even though 10 clients in the system is not many, this scenario already provides functionality that previously did not exist. For one of our test cases, a server sent content at 40 kbps to 10 clients, showing Zebra allows a video source on a cable modem to broadcast video to 10 people. Furthermore, since our system only requires command line configuration of the server address, Zebra is simple enough to be run by a regular Internet user.

In order to evaluate how well the stripe technique increases tolerance to node failures, we designed a test case where a client completely loses one of its incoming stripes, and is unable to instantaneously reconnect. In this case, the client still received some data, allowing for a degraded level of service as opposed to no service. Video still appeared on the client media player, although it was choppy and included some artifacts. This test proves that data from the working stripe was able to be used by the client. Hence, striping proves to be an effective technique in improving system robustness.

We also tested how well Zebra evaluates physical distances between clients by performing several tests using ping. We found that hosts that are substantially further away in fact return longer ping results. Since Zebra chooses connections based on this information, it prefers to connect to clients that are closer, reducing the physical distance that data travels. For most networks, this reduces network traffic by reducing the number of connection segments the data traverses.

5 Future Work

Currently our system uses TCP to transport data between proxies. TCP's congestion management keeps packets from being dropped during bursty sends, ensuring that all nodes in the system receive all the data. If UDP were used, then when packets in our system become clumped, the loss of a clump during transmission would mean the loss of several packets, and nodes at the bottom of distribution trees could see degraded performance. However, TCP provides ordered delivery, which hurts Zebra's overall performance. If a packet is dropped between two clients, TCP stops the delivery of all subsequent data until the missed packet is resent. In live video, packets have a limited useful lifetime. If data is delayed while waiting for a packet to re-send, it will expire while sitting in a TCP buffer on the receiving side. The ideal solution is to add only congestion management on top of the UDP transport. We did not have time for this enhancement, but it would be a valuable addition to our system.

Another extension that would improve the ability of Zebra to handle adverse network conditions would be for clients to constantly monitor the speed at which they receive both of their stripes. If the receive speed for a stripe falls below what it should be, the client proxy could try to determine where the bottleneck is. If both stripes are being received at a low speed, then the bottleneck is likely to be the receiving client. If only one stripe is slow, then the bottleneck is likely to be the parent node serving that stripe. This information could be used to decide whether to force a client out of the network or to have it receive data from a different sender.

6 Conclusion

Zebra shows that a simple peer-to-peer distribution system allows a user with limited sending bandwidth to serve live video on the Internet. Separating the video stream into stripes makes the system resilient to disruptions caused by connecting and disconnecting clients. Connecting peers by physical locality reduces Zebra's network traffic. By targeting a distribution size of 100 clients, the system can be centrally coordinated by maintaining full state.

7 Acknowledgements

We take this opportunity to thank Professor Robert Morris for his ideas on improving and focusing our design and motivation. We also thank Sanjit Biswas for his feedback and helpful ideas.

References

[1] Shoutcast, `http://www.shoutcast.com`

[2] Y. Chu, S. Rao, S. Seshan, and H Zhang. *A case for end system multicast*, IEEE Journal on Selected Areas of Communications (JSAC), October2002.

[3] Y. Chawathe, S. McCanne, and E. Brewer.*An architecture for Internet content distribution as an infrastructure service*, Unpublished work, February 2000.

[4] M. Castro, P Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. *SplitStream: High-bandwidth content distribution in cooperative environments*, In IPTPS '03, Berkeley, CA, USA, 2003

[5] A. Nicolosi, S. Annapureddy. *P2PCAST: A Peer-to-Peer Multicast Scheme for Streaming Data*

[6] I. Stoica, R. Morris, M. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for Internet applications*, In Proc. Of ACM SIGCOMM01, San Diego, CA, USA, August 2001.

[7] A. Rowstron and P. Druschel.*Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, In Proc. Of 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, November 2001.

[8] Real Networks, `http://www.real.com`

[9] Real-Time Streaming Protocol, `http://www.cs.columbia.edu/ hgs/rtsp/`