6.033 SECTION 11AM KATABI

Design Project 2: YouCast

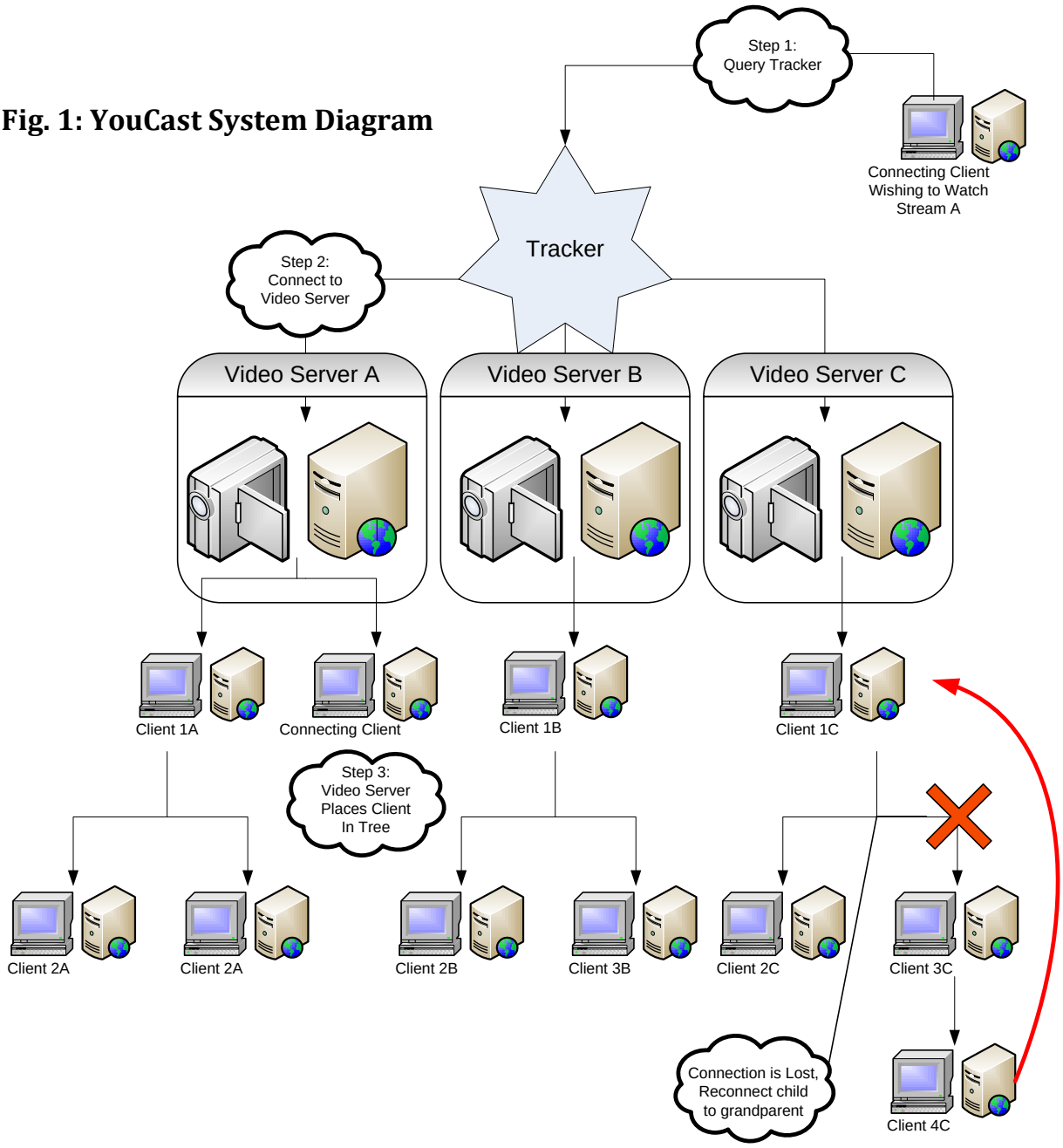
A Peer-to-Peer Streaming Service

Austin Chu
Benjamin Gleitzman
Alexander Patrikalakis
5/9/2007

Abstract

The YouCast service provides real-time peer-to-peer video streaming of multiple channels over the Internet. Using a tracking server, an individual video server for each channel, and client machines running the video server protocol, a video stream can be efficiently and reliably served to a large number of clients.

Fig. 1: YouCast System Diagram



Join Protocol

Overview

The join protocol for our video streaming system routes clients from a tracker to a video server, which in turn routes the client to a node in a tree of clients maintained on the server that represents the connectivity of all clients streaming video on a particular channel. The video server and peers use heuristics such as the number of peers connected to a node when new peers join a stream. This ensures that the branching factor at any node of the tree connecting peers to a video server is kept below a certain value (to limit bandwidth utilization), but also that the height of the connectivity tree is kept below a certain value (to limit the latency of the stream at the bottom of the tree). The computational load on the video server is minimized by decentralizing the structure of the tree. The tree data is kept up to date both by join calls and by status propigation up the connectivity tree during dissemination.

Tracker

The YouCast tracker implements operations for adding, maintaining, and querying the IP addresses of video servers over TCP. The tracker maintains two hash maps: one that maps from a channel number to a video server address (*forwardMap*), and another that maps from channel numbers to a time value measured in seconds from the UNIX epoch as a double (*timeMap*). Also, a queue of unused channel numbers (*unused*) is maintained, which starts off containing all unsigned integers. The tracker implements the following RPC stubs:

1. *void addServer(string address): addServer* takes an IP address, pops the next unused channel number off *unused*, and adds the mapping (*address -> unused.pop()*) to forwardMap. Finally, this channel number is mapped in *timeMap* to the tracker's current system time.
2. *void deleteServer(int channel):* The delete call takes a channel number, removes the mappings associated with channel from *forwardMap* and *timeMap*, and pushes the channel number back onto the *unused* queue.
3. *string getServer(int channel): getServer* takes a channel number and looks it up in *timeMap*. If there was no mapping for the given channel, it returns the loopback address as a string. If there was a mapping for *channel* but the timestamp in *timeMap* was more than 120 seconds old, then it makes the call deleteServer(channel) and removes the video server from the tracker. Otherwise, it returns the IP address of the server.

To deal with video server failure, video servers are required to send TCP keepalives to the tracker every 60 to 120 seconds (the actual interval is randomly chosen by the video server to evenly distribute tracker maintenance traffic over time). The keepalive, which contains a timestamp and channel number (around 8 bytes of payload), is received by a separate process in the tracker, which updates timeMap with new time stamps as they are received.

Joining Procedure

PeerNode Data Structure

A node (video servers and peers) in the connectivity tree is represented by the data structure PeerNode, listed below in Figure 2.

```
struct PeerNode {
      string grandIP; //grandparent node
      string parentIP; //parent node
      map<string, int> children; //child nodes and sizes
      map<string, int> siblings; //set of nodes with same parent
      string thisIP; //ip address of this node
};
```

Fig. 2: The PeerNode data structure

The size of child and sibling nodes is an integer representing the number of nodes connected below a particular child; this number is updated through status digest propagation during dissemination. The parent and grandparent of the video server (the root node) are the same as that node's *thisIP*. The grandparent and parent of a node at the second level of the connectivity tree are the same. Starting with the third level, distinct addresses are used in all three IP fields.

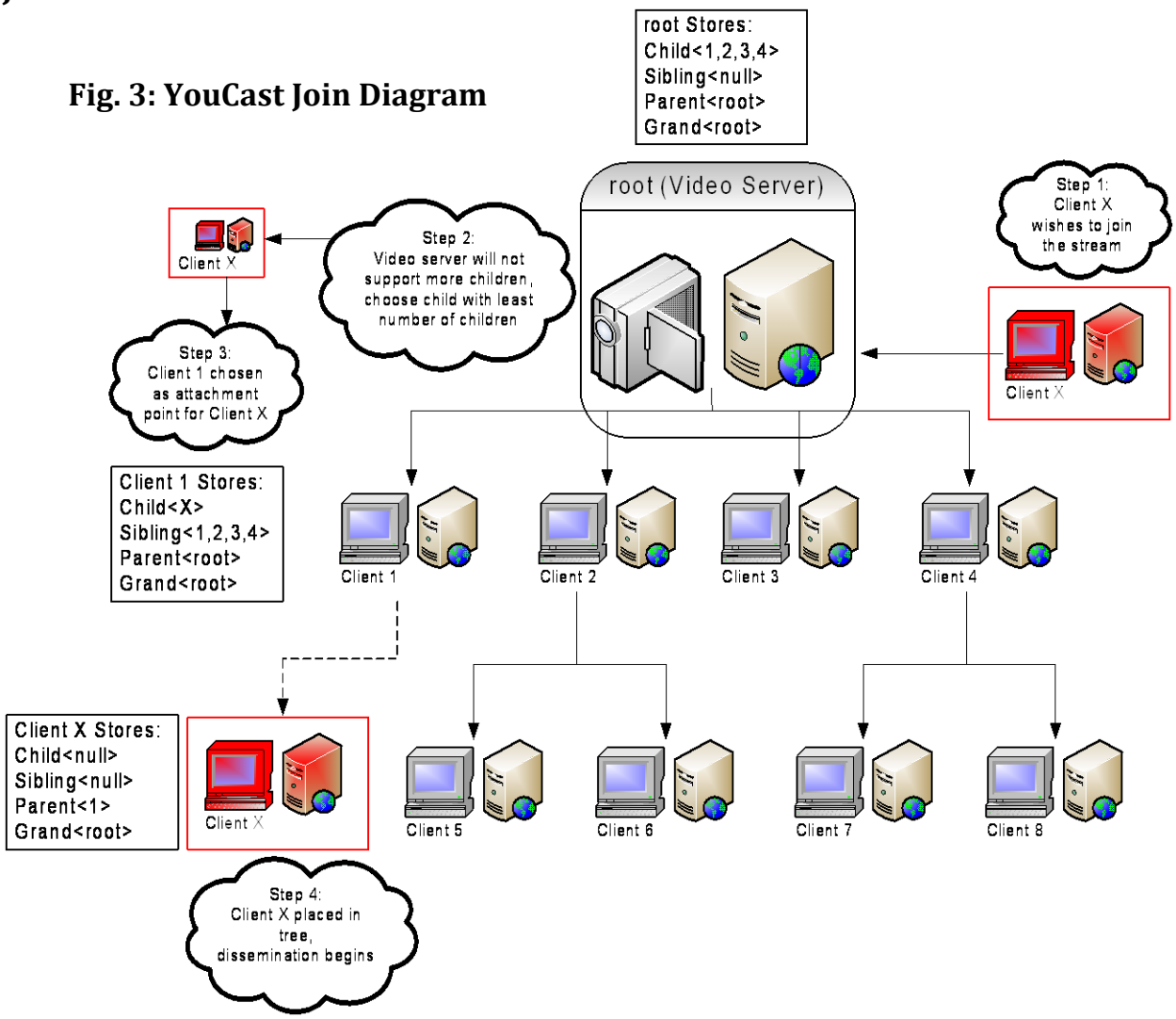
PeerNode RPC API

The PeerNode structure, maintained by each peer and all video servers, supports the following RPC calls:

1. *void addChild(string node, string child, int childsize):* This call adds *child* to *node.children* and maps the number *childsize* to that *child*. *AddChild*, if given a child string already in *node.children*, overwrites the value with *childsize*. A negative *childsize* will instead remove the child from *node.children*.
2. *set<string> getChildren(string node):* This call returns *node*'s set of children address strings of node, which are the keys of *node.children*.
3. *void addSibling(string node, string sibling, int sibsize):* This call adds sibling to node.siblings and maps the number sibsize to that sibling. If *AddSibling* is given a sibling string already in *node.siblings*, it overwrites the value with *sibsize*. A negative *sibsize* will instead remove the child from *node.siblings*.
4. *set<string> getSiblings(string node):* This call returns *node*'s set of sibling address strings, which are the keys of *node.siblings*.
5. *string getParent(string node):* This call returns *node.parent.*
6. *string getGrandParent(string node):* This call returns *node.grand*.
7. *void setParent(string node, string newPar):* This call sets *node.parent* equal to *newPar*.
8. *void setGrandParent(string node, string newGrand*): This call sets *node.grand* equal to *newGrand*.

Join Procedure

Fig. 3: YouCast Join Diagram



A new peer gets the IP address for the stream it wishes to watch from the tracker using the *getServer(channel)* call. Then, the new peer uses the address of the video server that was returned by the tracker to find a suitable place to join the channel.

Both the video server and all peers implement a join RPC stub that runs over TCP, *bool joinNode(string node, string& joinPoint*). If *node* has enough bandwidth to support another child, then *joinNode* returns true, and node is written to *joinPoint*. Otherwise, *joinNode* returns false, and the address of the child of node that has the smallest number of children is written to *joinPoint*. The new peer then repeatedly calls *joinNode* until it finds a node in the connectivity tree that returns true. The pseudocode for the *joinNode* call as well as for the *joinThread* procedure listed in Figure 4.

```
global int channel; //user has chosen this someplace else
bool joinNode(string node, string& joinPoint) { //RPC
        if(system(has 0.5MBps left)) {
                joinPoint = node;
                return true;
        } else {
                string child = node with least number of children in this.children;
                joinPoint = child;
                return false;
        }
}

void* joinThread(void*) { //LOCAL
        string vs = getServer(channel);
        bool result = false;
        string tgrand = vs;
        string tparent = vs;
        int depth = 0;
        while(result == false) {
                string where;
                result = joinNode(tparent, where); //one RTT
                tgrand = tparent;
                tparent = where;
                depth++;
                //make sure depth is reasonable
                if(depth > MAXDEPTH) { //MAXDEPTH is 9
                        //to guarantee real-time delivery of frames at the bottom
                        //as well as termination of thread
                        throw new UnsupportedOperationException("channel full");

                }
        }

        //now we found a good place to join
        this.parent = tparent;
        this.grand = tgrand;
        //push the new child to the parent
        addChild(this.parent, this.thisIP, 0); //one RTT
        //update this sibling list
        this.siblings = getChildren(this.parent); //another RTT
        push self onto siblings list of each node in this.siblings in parallel //one RTT
}
```

Fig. 4: joinNode and joinThread

Performance

Unless the channel is full, a maximum of 9 joinNode RPC calls will be performed, implying a maximum of 9 RTTs until a new peer can join a channel and begin buffering under the dissemination protocol. If the maximum latency between any two nodes is 1 second, then the maximum time it would take for a new peer to join a stream would be approximately 12*RTT+18*1sec + 3 sec, or about 21.012 seconds. Assuming average latency (100ms) between any two nodes, it would take approximately 2 seconds for a new peer to join.

Decentralizing the connectivity structure of the channel away from the video server takes computational and bandwidth load off it, allowing it to focus on disseminating frames to its children. Localizing the connectivity structure of a channel to individual nodes, by means of the children (used to disseminate) and sibling (used to repair) maps, allows node repair to be localized without burdening unaffected portions of the connectivity tree. Localization of connectivity structure also allows the status messages to be localized to small groups of nodes (child sets, sibling sets), reducing unnecessary network traffic that would otherwise need to propagate up the tree to a managing authority.

Assumptions

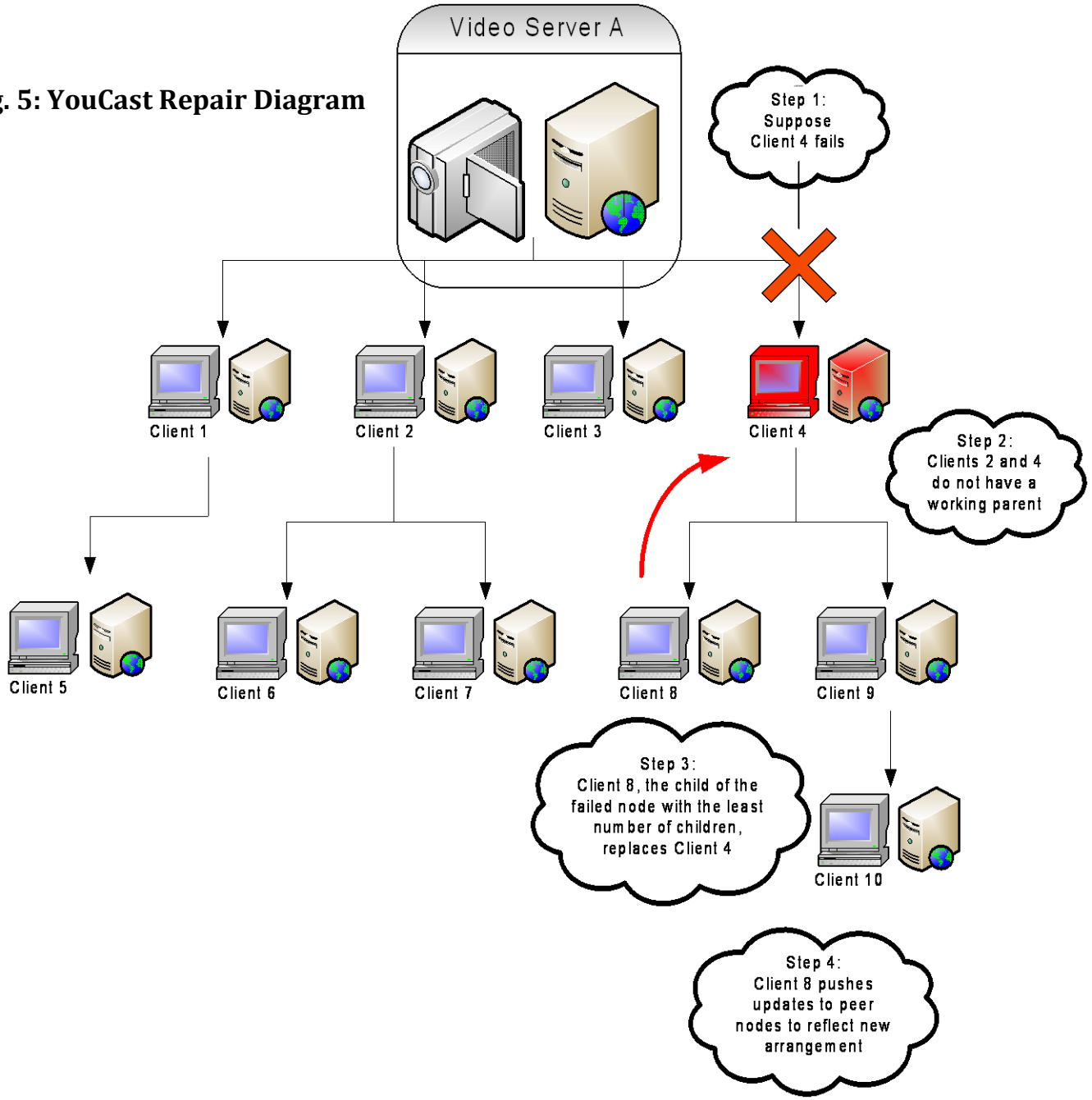
NTP is used on all servers, peers, and the tracker to synchronize YouCast to second-level accuracy. Furthermore, we assume that there is some omnipotent process on each host that can tell us *node*'s upstream bandwidth. We assume that failures do not occur when joining. The bandwidth required for a single stream to a child peer is exactly 0.5 MBps. Furthermore, assume that any given node supports a maximum upstream bandwidth of at least 1.0+alpha (for keepalives and other small maintenance packets) MBps and at most 2.0 MBps, implying that the maximum number of children of a peer (the max branching factor of the connectivity tree) is 4, while that the smallest number of supported children is 2. Downstream bandwidth on all peers must be at least 0.5MBps.

Limitations

A maximum of 2^32 concurrent video streams on the tracker are supported. A given video server can support a maximum of 4^9 = 262144 connected peers to limit the depth of the connectivity tree (and hence latency of the real time stream), given a maximum of four upstream connections per node.

Repair Protocol

Fig. 5: YouCast Repair Diagram



The YouCast repair protocol takes advantage of locally maintained PeerNode connectivity tree data to localize and compartmentalize the effects of failed streaming peers, as well as to take the computational burden of reattaching disconnected subtrees off the video server and onto individual peers.

Detection

Failure of a node N is detected by N's immediate children as a lack of downstream frames for about 15 seconds. N's children then each proceed to reattach themselves to the stream by running the *rejoin* process.

Rejoin Procedure

As the status messages sent during dissemination keep the children and sibling maps of all peers updated, all the children of the failed node N will have identical sibling maps. The child under the dead node N that has the least number of children is then computed by each sibling under N, and the repair process exits on all nodes except on the node that was the same child as the one computed.

If the node with the least number of children is not a monk (no children), then the connectivity tree is traversed down its length until a monk is found; the number of calls to getChildren performed is limited by the maximum height of the tree, as well as by the height of the failed node N in the tree. If the node M with the least number of children was a monk (no children), then it replaces the failed node N as is described below.

The rejoin procedure updates *N.parent* (the grandparent of children of the failed node N) by removing N and adding M to the children map of the grandparent with the *addChild* call. Then, *rejoin* updates M's parent field to be the grandparent of the executing node (the reattachment point). Next, the sibling maps in the siblings of the failed node N are updated, using two parallelized RPC calls:

1. *addSibling(s, N, -1)* for all *s* in *this.grand.children* is executed in parallel for all children of the parent of the failed node to remove the failed node from the sibling maps of each of its siblings, and
2. *addSibling(s, M, 1)* for all *s* in *this.grand.children* is executed in parallel to add the new node M to the sibling lists of the siblings of the failed node N. The actual size is of this sibling is updated by the status messages in dissemination.

Next, the parents of the children of the failed node N are pushed to be M by a parallelized call to setParent. Finally, the monk M updates its own sibling map exactly as they would be initialized in the joinThread procedure, specifically by pulling the children map of its new parent and assigning it to its sibling map.

The base case of this quasi-recursive loop is when a node with exactly one child fails; the child simply notices a lack of frames and replaces its parent. Failure of the video servers means all bets are off. We add a special case for when the *parent* and *grand* and the *thisIP* are all the same (implying a video server); the children of the video server give up with a timer and let failure propagate throughout the connectivity tree. The tracker protocol makes sure that new peers cannot connect to a failed video server.

Performance

The search for a monk node to replace a failed node is optimized for speed, as it heuristically guided by choosing child nodes with the least number of children, reducing the depth to which the connectivity tree must be searched for a replacement node. The search for a monk to replace a failed node is optimized for locality, computationally encumbering exactly one child of the failed node, and limiting possible propagation of failure only to the subtree of the connectivity tree rooted at the failed node; the remainder of the connectivity tree remains unaffected by the failure. The time to repair is limited by the following RPC calls:

1. O(height) queries down the connectivity tree to find a monk
2. O(2) pushes for updating the grandparent's children and the monk's parent
3. O(3) parallelized pushes for updating the siblings of the monk and the new parents of the children of the monk

These RPC calls give an upper bound of (height+5) RTT's for repair; assuming an average RTT of 100ms, the MTTR is 14*RTT = 1.4 seconds.

The monk must have the bandwidth ability to serve all the children of the failed node.

Dissemination Protocol

Overview

The dissemination protocol encapsulates all of the regular communication among the peers, including the video server. It has two major tasks. It is in charge of transporting video from the video server to all of the clients in a timely manner. It also carries data about the state of the tree between nodes, helping to keep each node's PeerNode data structure up to date.

The dissemination protocol must be able to transport a reliable, high quality video stream from the video server to all of its clients in the face of the unreliability of the Internet. Most status messages are transported over TCP for reliability, but because streaming video is a real-time application in which keeping a steady stream of video frames flowing is more important than necessarily guaranteeing delivery of every last frame in the stream, the video itself is transported over UDP. This combination allows the YouCast system to avoid the cost of guaranteeing delivery of frames even when they're not useful anymore while still providing some mechanism to re-request dropped frames that have been detected early enough.

The TCP connection maintained by the dissemination protocol also does double duty as the vehicle by which nodes pass information about the state of the tree. Periodic keep-alive messages from children to their parents reassure the parents that the child still wants to be sent the video stream. Less frequent updates both from children to their parents and from parents to their children allow each node to keep the data stored in their PeerNode structure fresh.

Video Streaming Protocol

The main goal of the video streaming portion of the dissemination protocol is to get as many frames of the video stream to as many clients as quickly as possible. To achieve this, the default mode of each peer is to resend each frame of the video stream as soon as it receives it. For the video server, that means that frames are sent to the server's children as soon as the camera has generated them. For the other peers, it means that frames are sent to a peer's children as soon as they are received from the peer's parent.

In order to ensure timely transmission of video from the server with a delay of less than 60 seconds from frame generation to display, the video server attaches a timestamp to each frame before it propagates the frame to the consumers of the stream. Since we are assuming that all of the system clocks are at least loosely synchronized, each client can use the timestamp to determine about how much time has elapsed since the original recording of each frame. Using this information, each client then buffers frames for playback such that all video will play with about a 60 second delay. Although having the maximum allowable delay does degrade the real-time effect, we belive that the increase in video quality and extra robustness gained from allowing the system more time to work around network congestion and peer failures is worth the wait.

Since the frames themselves are distributed using UDP, frames can be silently dropped or delivered out of order by the UDP transport layer. To combat this problem, the video server also attaches a sequence number to each frame. Using this sequence number, recipients can detect dropped or reordered frames.

When a client detects that it has received a frame out of order (when it receives a frame with a sequence number that is at least two frames after the last frame received), the client will start a timer of one second. Since it is specified that the network has a maximum delay of one second, if the timer expires without the arrival of the missing frame, the client can assume that that frame has been dropped by the network. Once a client detects a dropped frame, it adds the frame's sequence number to a list of specific frames that it would like its parent to resend.

Whenever a client's list of frames to request is non-empty, the client sends a special request for those frames to its parent using the TCP connection. Clients may batch such requests and send special requests no more than once every two seconds to allow for the maximum round trip time delay of the network. When a peer receives such a request from its child, it will try to resend all the requested frames to the requesting client.

Since frames that are too old are not useful, clients remove frames from the list of frames to request when those frames are to be played within 2 seconds. Having given up on receiving the missing frame in time for it to be useful, the client can simply leave the frame before the missing frame on the screen for an extra 50 milliseconds before moving to the next frame in the stream.

If the TCP connection fails or if the client does not receive any new frames from the server for about fifteen seconds, then the client assumes that the server has failed, and the client executes a join request to the main video server to get connected to a new parent server.

PeerNode Updating Protocol

Using the TCP connection that is kept open between every pair of parent and child nodes, the dissemination protocol also distributes information about the immediate structure of the tree to each node.

The most basic information that is passed in the TCP connection is whether a child is still interested in the video stream being sent by a parent. Every ten seconds, clients send a small keep-alive ping to its parent, just to reassure that the child has not failed or left the stream. Thus, if a parent does not hear anything from a child for fifteen seconds, or if a parent detects that a child has closed the TCP connection, the parent can assume that the child no longer wants to receive the video stream and can stop streaming to that child. The parent then also removes that child's mapping from the children map of its PeerNode data structure.

In addition to the simple keep-alive pings from children to their parents, the YouCast system also sends information about the local state of the tree of clients using the TCP connections. This information flows periodically both from parents to their children and from children to their parents.

Each node updates its parent with data about the subtree rooted at that node. More specifically, each node calculates the number of descendants it has by calculating the sum of the integer values in the children map of the PeerNode structure, thus finding how many descendants its children have. The parent then adds the number of children, found by taking the size of the children map, and reports the current value of this calculation every thirty seconds to its parent (the actual time ranges between twenty and forty seconds to stagger network traffic).

In addition, each node tells its children updates about the status of their siblings. Every thirty seconds or so, it pushes a copy of the children map in its PeerNode data structure to all of its children; the children can then use this data directly as their new sibling map.

Through this procedure, each node's PeerNode data structure can be kept up to date efficiently and with minimal overhead.

Performance

The overhead incurred by the PeerNode updating protocol can be shown to be minimal. The actual payload of the updates from the children to their parent an integer, so even considering large amounts of overhead for transporting this number over TCP and possible RPC calls, such updates can't require more than 50 bytes every thirty seconds. If we assume that each node has on average 4 children, then the incoming data for parents is just 200 bytes per thirty seconds, still an amount dwarfed by the 500 kilobits per second required by the core video streaming.

The payload of the updates from the parents to their children is larger, but still relatively small in comparison to the bandwidth required by video streaming. Each key-value pair in the map that is being passed down consists of an IP address and in integer; even representing the IP address as a string, such pairs require no more than 20 bytes of raw data. If we assume that there are on average 4 children per node, then the entire mapping has on average 80 bytes of raw data. Now, conservatively estimating a 300% overhead from TCP and possibly RPC, we still have no more than 400 bytes passed to each child per 30 seconds, or 1600 bytes per 30 seconds that each server has to push. Once again, this amount is still dwarfed by the 500 kilobits per second required by the core video streaming.

Security

The YouCast system provides for streams to be protected from eavesdropping that may occur on the network. As frames are sent from the video server, the UDP packets will be augmented with a field indicating if that packet is encrypted. Encrypted streams will require a password to be provided by the user upon connection to the main video server in order to exchange a shared secret to decrypt the video stream. Packets from public streams will not set the flag in the encryption field, allowing anyone who receives the packet to view the content. This shared secret ensures that encrypted video streams will not be compromised by packet sniffers.

Conclusion

The YouCast system employs a series of commands to provide real-time peer-to-peer steaming of video. The join protocol contacts the tracker, which routes the client to the correct video server. The video server places the client into a tree of streaming peers. The dissemination protocol broadcasts video data to peer nodes, and also keeps tree information up to date between nodes. The repair protocol accounts for node failure by rotating a node’s position in the tree. Finally, UDP packet augmentation for encryption allows for secure transmission of video data.

Our YouCast system is lightweight because of the decentralized design. The video server will not become overrun with RPC calls because much of the communication between nodes is handled by the nodes within the connecting tree structure. The design is scalable because the depth of the tree is kept to a minimum. By using a broad tree, additional nodes can be added to the tree without reducing

performance or increasing latency. Finally, the design is robust and can handle fault tolerance without disruption of the video stream to client nodes.

However, the design does leave room for improvement. Dynamic reconfiguration of the connectivity tree could be implemented to prioritize peers with larger bandwidth toward the head of the tree. This would reduce bottlenecks in the streaming protocol and ensure efficient bandwidth utilization. Additionally, the design assumes that the join and repair protocol is an atomic action, and that no faults will occur during these processes. The design could be augmented to allow for faults to occur during these operations.

Via the described protocols, the YouCast service provides for a video stream to be efficiently and reliably served to a large number of clients.