PeerStreamer

Nolan Eastin (neastin@mit.edu)
Louis Sobel (sobel@mit.edu)

6.824 Spring 2014 Final Project
5/11/2014

Problem Description

Increasing popularity of video streaming sites is applying pressure on the current infrastructure of the internet merely because of the large amount of data being streamed every day. Video streaming accounts for over 50% of all traffic. This project will attempt to localize requests by limiting the amount of requests outside of clusters. Previous work includes CoralCDN a peer to peer content distribution network that uses a distributed hash table.

Design Overview

We designed and implemented PeerStreamer, a combination peer to peer and centralized video streaming and caching service. Through a hierarchical layer of nodes, video streams can be widely distributed by taking advantage of locality within the network to get content from a peer that is as close as possible, falling back to a centralized master if necessary. The design is robust in the face of peer and master failure. The project was implemented in javascript using the `node.js` server-side javascript runtime. The ZeroRPC javascript RPC library was used. The code is available at https://github.com/louissobel/peerstreamer/tree/submission.

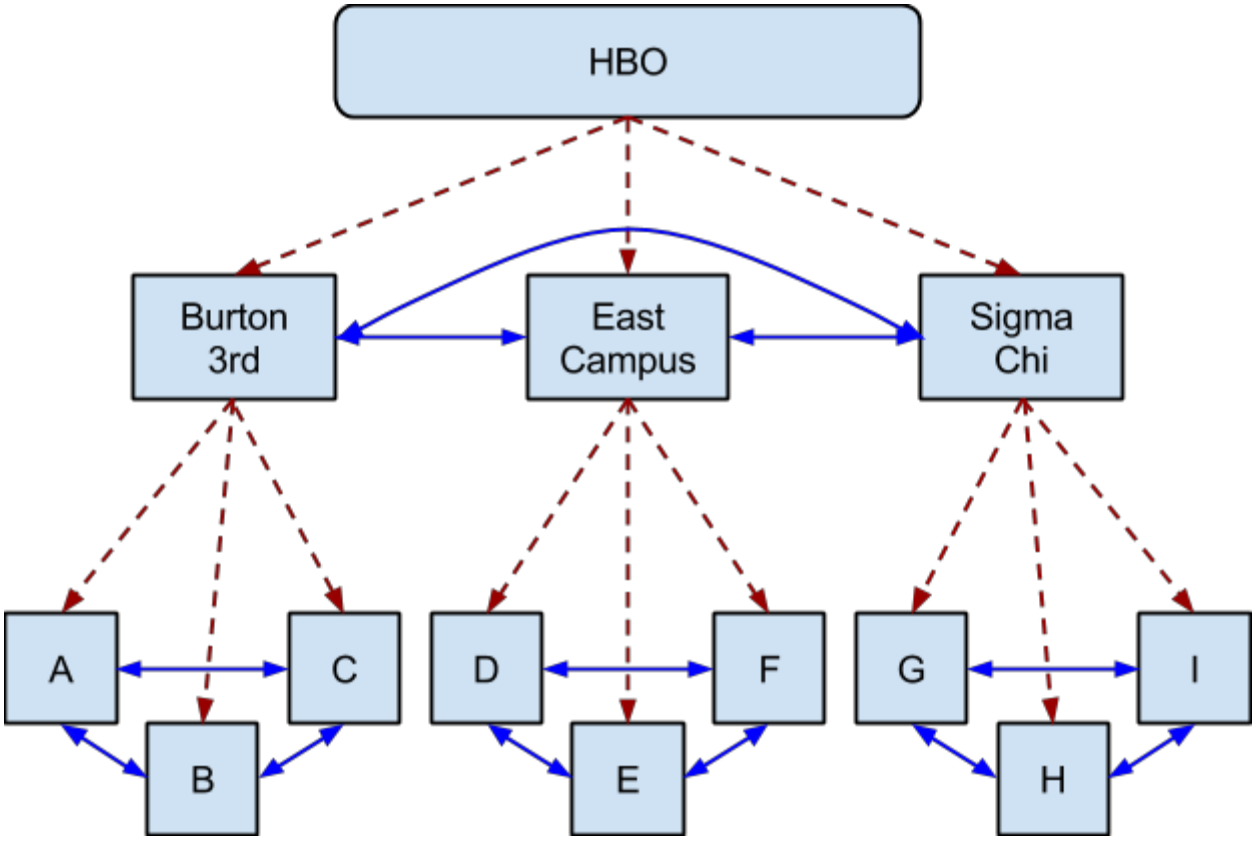

Figure 1: An example hierarchy of peers and masters.

Hierarchy

PeerStreamer is designed to support a hierarchy of cooperating peers. Figure 1 shows an example of such a hierarchy. Red / dashed links represent master / peer relationships, while blue solid ones, peer to peer. When a node comes up, it first registers with its master. For example, HBO is the master of Burton 3rd, Burton 3rd is the master of A. When a node requires a chunk, say, A wants `gameofthrones:0`, A first asks its master where it can get that chunk. If the master knows of any peers that have it, it will tell A to get the chunk from a Peer. In this case, say neither B nor C have the needed chunk, so it is Burton 3rd's responsibility to find it. Burton 3rd asks it's master, HBO, where `gameofthrones:0` is. Neither of Burton 3rd's peers, East Campus or Sigma Chi, have the chunk, so HBO provides it directly to Burton 3rd, who caches it and passes it to A. Say B now wants `gameofthrones:0`. B asks Burton 3rd where it can find it. Burton 3rd tells B that A has it, so B gets it from A. Say E now wants that same chunk. E asks its master, East Campus, where to find the chunk. Neither D nor F have it, so East Campus needs to find it. East Campus asks HBO where `gameofthrones:0` is, and HBO tells East Campus to check with Burton 3rd, who has it. East Campus then gets it from Burton 3rd, caches it, and passes it back to E. This hierarchy could be extended further, with a 'MIT' server as the master of Burton 3rd, East Campus, and Sigma Chi, which has as it peers Harvard and BU, et cetera.

Stream

Our implemented solution gives each client a Stream manager. This stream manager is responsible for getting video chunks from a source. Upon a request from a peer to a master, a stream manager creates a stream with a unique id. This stream id is used to re-access the same stream and request the following chunks. Each client can have multiple streams open at any given time (one stream per file). This stream is an abstract representation of the data requests sent from the peer to a master. This stream generates a 'buffer' (arbitrarily set to 10 chunks) which given a request for chunk 0, will proceed to also request chunks 1-10. This decision was made on the assumptions that users trying to watch video would do so for more than multiple seconds, with the goal of decreasing latency. As chunks of video are consumed, callback functions prompt the stream to request more chunks, which ensures buffer size is maintained.

Streams check local storage, and if the next chunk is not there, it requests it from the master and stores it in local storage. Video stream will check the local storage as needed. Streams query the master for a given chunk, and the master will then return a list of peers that have the file. A stream will request chunks in turn from different peers, and on failures will switch to a different peer. If all peers fail, a stream will default back to the master which always has the file requested. Any time a master fails, the stream will trigger a "masterFailed" event, which will make the client switch their source to a "super master". (So if Burton 3rd fails, stream will start requesting from HBO. The client periodically retries the master, and if successful, switches back to him. The stream periodically re-queries the master for a file to ensure it doesn't have stale state. Streams are marked as 'done' once EOF is reached. Clients are then required to stop sending requests on the same stream, which causes a stream to be abandoned. Each stream includes a "last access" property which allows for easy garbage collecting.

ChunkStore

In order to have the stream be able to look ahead and nodes to use each other, each node has a ChunkStore which it uses to save chunks that is receives. This is a persistent, multi-tiered, and LRU cache. The ChunkStore is persistent so that even across reboot, a node knows what chunks it has. It is multi-tiered in that it has a in-memory cache, to make recently inserted or accessed items readily available. It is LRU because it has a limited size, in our case 50 chunks,
because a user would have only so much space they'd want to allocate for PeerStreamer.

The persistence strategy is complicated to make sure that the chunk store never believes that it has a chunk that it actually doesn't. When a chunk is inserted into the ChunkStore, it is immediately placed into the hot-cache, and an asynchronous write is started. The chunk is marked as present in the chunk store, but not persisted. When that write finishes, the chunk is then marked as persisted. When a chunk is evicted from the ChunkStore because there is no more space, the chunk is marked as not present, but the file is not actually yet deleted from disk. Periodically, the ChunkStore synchronizes it's state to disk by writing out its internal data structures into a file called `manifest.json`. Only files that are marked as persisted will get written out. This ensures that if some write that is ongoing fails, the data structure on disk will not include the file that failed to write. After the `manifest.json` file is written to disk, then all of the deletions scheduled to occur take place. This ensures that there is never a file that is in the datastructure on disk that is not actually present. As a precaution, when restoring its state from disk, the ChunkStore asserts that all the files listed as present actually are.

Protocol

- *Register*

  A peer registers with the master, telling him he is now available. He also sends a manifest informing the master of which file chunks the peer has at start up (loaded from disk). Masters will forget all assumptions they had about a child before the register and only trust the state sent on register (increased by report)

- *Query*

  Peer sends a filename, chunk request to master. The master will return a list of peers that have the particular filename, chunk pair.

- *Get*

  Request sent from a peer to a master or another peer. A chunk of video data is sent back. Peers prioritize asking other peers, but will switch to master if no other peers have it.

- *Report*

  Peers send a 'report' to a master alerting him of any files they have. This happens after a successful get (from peer or master). A peer also informs its master when a chunk is evicted from the ChunkStore. This guarantees master always has correct state.

Video Streaming

In order to demonstrate that our system was actually capable of showing video, it was important that we could play video out of the system. To that end, we wrote `video_streamer.js`, which acts as a simple client, repeatedly getting subsequent chunks of a stream. It takes advantage of VLC player's (reference) ability to play from a file descriptor, including STDIN. The video streamer takes advantage of this by starting VLC with access to its STDIN file descriptor. Then, as chunks are obtained from PeerStreamer, they are written to this file handle so VLC can play them. VideoStreamer works over the uses the same RPC interface used within nodes, but is intended to run on the same machine as a PeerStreamer node, not remotely, as it does not do any cacheing nor act as a peer.

Conclusion

Our design was able to successfully distribute network load throughout the system rather than the old top-heavy traffic loads. It is able to handle peer and master failures, and restarts are handled gracefully through peer registration, persistent storage on disk and syncing, and reporting to master. Our design minimizes the amount of messages passed and limits system latency by timing out quickly and switching sources often to reduce wasted time trying to find a source. The buffers we build provide us with reasonable time windows to switch sources or default back to a master/supermaster.