# StreamTree – A combination of peer-to-peer, hierarchical Video Streaming and Distributed Caching service.

# Tejas Bondre

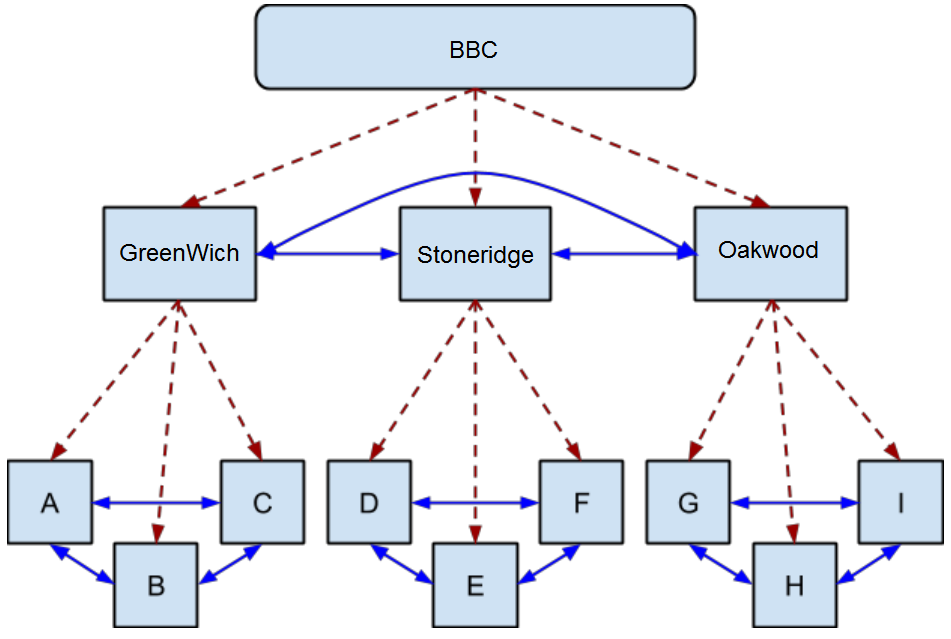
https://github.com/TejasBondre

**Problem:**

Increasing popularity of video streaming sites is putting 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 attempts to localize requests by limiting the amount of requests outside of the clusters, building upon the CoralCDN (a peer-to-peer CDN that uses a DHT).

**Design Overview:**

I’ve designed and implemented StreamTree – a combination of peer-to-peer, hierarchical video streaming and distributed caching service. Through a hierarchy of nodes, video streams can be widely distributed by taking advantage of locality within the network to get content from the nearest possible peer, falling back to a centralized master if necessary. The design is robust in the face of peer failure as well as master failure. The project is primarily implemented in javascript using the node.js server-side javascript runtime.



**Hierarchy:**

StreamTree is designed to support a hierarchy of co-operating peers. Figure 1 shows an example of such a hierarchy.

Red/dashed links represent master-peer relationships, while blue/solid links represent the peer-peer relationships. When a node comes up, it first registers with it’s master. For example, BBC is the master of GreenWich, and GreenWich is the master of A. When a node requires a chunk, say A wants ‘sherlockholmes:0’, A first asks it’s 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 GreenWich’s responsibility to find it. GreenWich asks it’s master, BBC, where ‘sherlockholmes:0” is. Neither of GreenWich’s peers, Stoneridge nor Oakwood, have the chunk, so BBC provides it directly to GreenWich, who caches it, and passes it to A. Say B now gets it from A. Say E now wants that same chunk. E asks it’s master, Stoneridge, where to find that chunk. Neither D nor F have it, so Stoneridge needs to find it. Stoneridge asks BBC where ‘sherlockholmes:0’ is, and BBC tells Stoneridge to check with GreenWich, who has it. Stoneridge then gets it from GreenWich, caches it, and passes it to E. This hierarchy could be extended further, with a UF server as the master of GreenWich, Stoneridge and Oakwood, which has as it’s peers UCF, Santa Fe College, etc.

**Stream:**

This particular solution gives each client a Stream manager. This stream manager is responsible for getting video chunks from a source. Upon request from a peer to a master, a stream manager creates a stream with 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 ent from peer to a master. This stream generates a buffer (arbitrarilily set to 10 chunks) which given a request for chunks 0, will proceed to also request chunks 1 to 10. This decision was made on the assumptions that users trying to watch video would do so for more than a few 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 GreenWich fails, streams will start requesting from BBC). The client periodically retries the master, and if successful, switched back to it. 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 requird to stop spending 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 it receives. This is a persistent, multi-tiered, and LRU cache. The ChunkStore is persistent so that even across reboots, a node knows what chunk it has. It is multi-tiered in that it has an in-memory cache, to make recently inserted or accessed items readily available. It is LRU cache because it has a limited size, in this case 50 chunks, because a user would have only so much space they’d want to allocate to StreamTree.

The persistent strategy is complicated to make sure that the ChunkStore 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 asynchronos write is started. The chunk is marked as present in the ChunkStore, 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 yet actually deleted from disk. Periodically, the ChunkStore synchronizes its 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 present.

**Protocol:**

*Register*: A peer registers with the master, telling the master that it is now available. It also sends a manifest informing the master of which files chunks the peer has at startup (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 it of any files they have. This happens after a successful get (from peer to master). A peer also informs it’s master when a chunk is evicted from the ChunkStore. This guarantees that the master always has correct state.

**Video Streaming**:

In order to demonstrate that this system is actually capable of showing video, it is important that I could play video out of the system. For which, I wrote video\_streamer.js, which acts as a simple client, repeatedly getting subsequent chunks of a stream. It takes advantage of VLC player’s ability to play from a file descriptor, including STDIN. The video streamer takes advantage of this by starting VLC with access to it’s STDIN file descriptor. Then as chunks are obtained from StreamTree, they are written to this file handle so VLC can play them. Video\_streamer works over the same RPC interface used within nodes, but is intended to run on the same machine as StreamTree node, not remotely, as it does not do any cacheing nor act as a peer.

**Conclusion:**

This application is 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, persisten storage on disk, and syncing, and reporting to master. This 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 provide reasonable time windows to switch sources or default back to a master/supermaster.