P2P On Demand Video Streaming Based on Tit-For-Tat Strategy

Johann Lingohr, Eva Ng, Sadaf Yadegari, David Karchynski
https://github.ugrad.cs.ubc.ca/CPSC416-2018W-T1/P2-n3u0b-x4w7-u9w7-y7v9a

Introduction and Motivation

In recent years, on demand video streaming services have grown in popularity. However, a number of ways to restrict users' access to content have emerged – location content access restriction is a common example. Most Peer to Peer (P2P) networks offer a solution to share content among users without location restrictions and free of charge. But as a free-for-all system, it leads to bandwidth availability issues, as well as free-rider problem – users who stream files from the network without contributing back. This paper outlines a solution to free-loader problem in P2P networks using tit-for-tat strategy. Also, our prototype system for P2P on demand video streaming aims to provide a way to explore available content without the use of a central repository for indexing the available files.

System Architecture

Terminology

Seeder: a node streaming data to another node
Leecher: a node streaming data from another node
Gossip network: collection of all nodes spreading information about each other's state and the files they host
Swarm: collection of nodes downloading the same segment of the file
Cluster: collection of nodes in swarms downloading the same file
Advancement: In terms of seeding, advancement refers to the amount of seeded content a swarm has

System Components

Our system consists of a network of peer nodes. Nodes can act as both as hosts for contents as well as clients that send request to other peers for content. As a host, each node makes the files it has locally available to other nodes on the gossip network. Following the terminology used in BitTorrent, when a host node is actively providing content to others we call it a *seeder.* As a client, a node can send out requests to seeder for the content they are hosting. When a client node is receiving content from a peer we call it a *leecher.*

Prototype Design

Similar to the BitTorrent protocol we assume clients are self-interested and care only about receiving resources they are immediately interested in. Unlike file-sharing services in which clients interested in the same file can exchange pieces of the file randomly so long as they eventually receive the entire file, clients in a video-streaming service care, to some extent, about the order in which they receive resources. That is, clients are interested in a specific subset of the resource that allows them to continue watching the stream at their video playback position uninterrupted and this subset changes over time. A consequence of this assumption is that when any two peers interested in streaming the same video communicate, they will exchange resources only if each has something the other is interested in. For example, if a peer p_1 has been streaming a resource for some time and a new peer p_2 joins at a much later time, then p_1 and p_2 will exchange resource only if the other has something they do not. Since p_2 has recently joined it cannot have resources p_1 is interested in and so they will not share. While we can naively have a seeder send the desired resources to each peer independently so they can stream, such a solution cannot scale when a single seeder has multiple streaming clients and the client may have limited upload bandwidth, potentially deteriorating the quality of streaming for all clients of that seeder.

This means the design of a P2P video-streaming service must avoid these suboptimal outcomes through some tit-for-tat mechanism and consider how to incentivize peers to participate in seeding. In our prototype we handle these issues by considering 1) how to manage peers in the network and peers interested in the same stream, 2) how to design a seeding policy between a seeder and peer and 3) how to design an exchange policy between two peers streaming the same content.

Peer Management

In order for a seeding node to service multiple streaming nodes the seeder must be able to effectively organize streaming clients. Our prototype organizes groups of nodes interested in the streaming the same video from a seeder into a cluster. Within this cluster nodes are further partitioned into swarms, groups of streaming nodes interested in similar segments of a video. For example, partitioning a video file into *n* segments means there are *n* swarms, each interested in a different segment depending on their playback position. Our implementation will support maximum of 4 swarms but does not put a cap on the number of nodes inside a swarm.

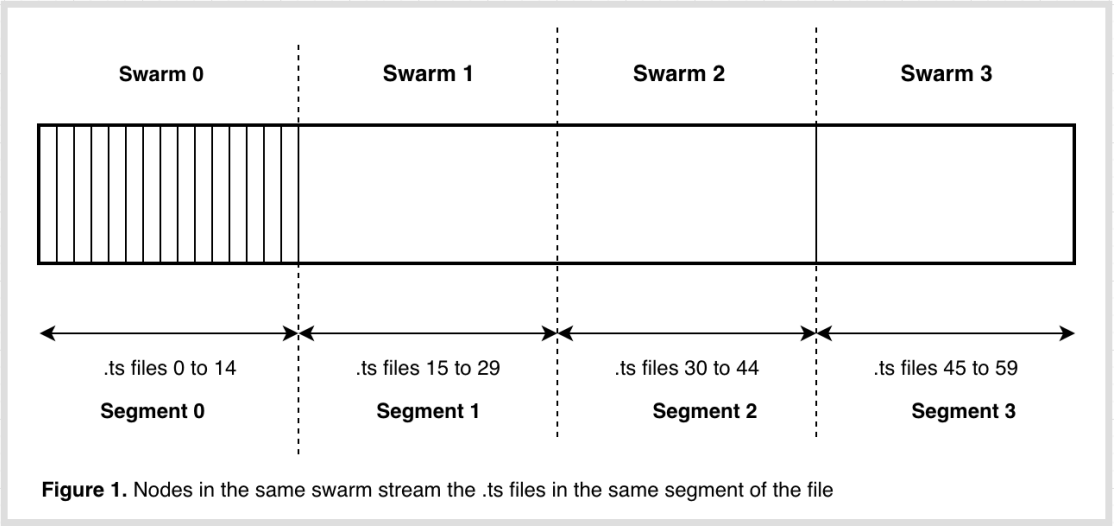

Figure 1. Nodes in the same swarm stream the .ts files in the same segment of the file

Cluster Formation

Nodes downloading the same file belong to a seeder's cluster. Nodes inside this cluster are grouped into swarms. All nodes in the same swarm download the chunks in the same segment.

When a node is started, they contact their peers in the file gossip network. If there is already a cluster at the peer's address, they will join the cluster. Otherwise they will start a cluster at the *ClusterBindAddress* provided to the node upon initialization. Nodes find out about what files other peers are hosting in the network through gossip.

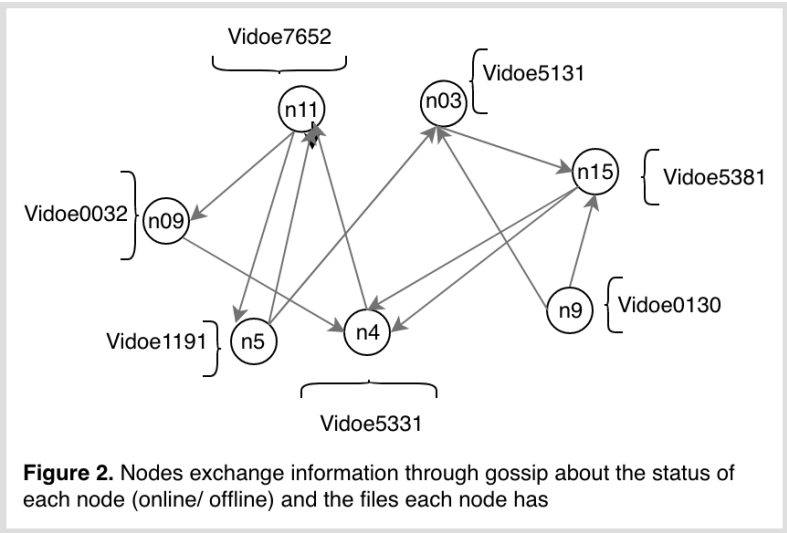

Figure 2. Nodes exchange information through gossip about the status of each node (online/ offline) and the files each node has

They can then contact the seeder and get the file metadata from them. The file metadata includes information about the segments and the chunks that make up the file. When a node wants to stream a video, they contact the seeder directly and seeder sends them the

file metadata over TCP connection. They will then join seeder's cluster and will receive chunks from the first segment of the file.

Each node runs a Trackers that gets updated every time a node joins the seeder's cluster, a node move to more advanced swarms, or when a node leaves the cluster. Upon joining the seeder's cluster the seeder sends the node a chunk required by the most advanced swarm. This will allow the new node to barter with the nodes in the same or advanced swarm for chunks it needs.

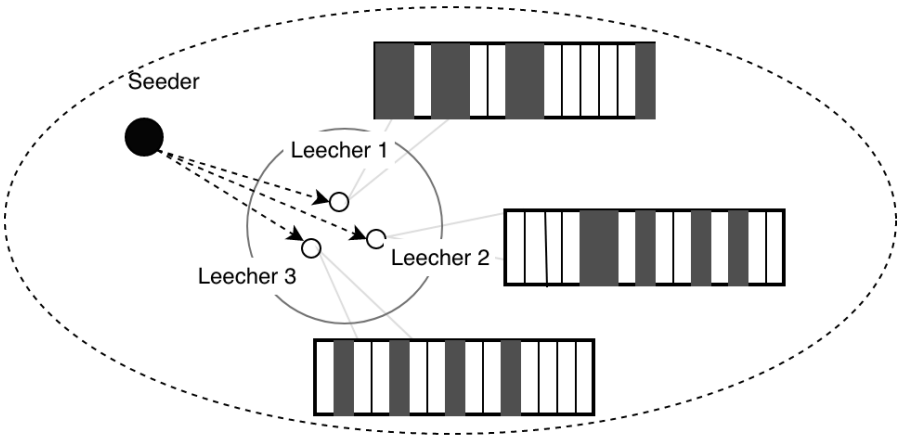

Figure 3. Seeder's cluster with one swarm. Seeder passes random chunks within the same segment to each node in the swarm

As node p in swarm s_0 receives all the *.ts* files belonging to s_0's segment of the file, it sends a broadcast message letting the seeder and others know that they will be moving to a more advanced swarm. All nodes that receive this broadcast, update the swarm to node ID mapping for p. Seeder then starts passing the chunks from segment 1 of the file to nodes in swarm s_0, who store these chunks and also pass them on to node p in the more advanced cluster. Following the tat-for-tat strategy, node p then reciprocates by sending back chunks from segment 0 of the file.

Seeding Policy

The seeding policy is concerned with how a seeding node should submit chunks of a video into the cluster it is feeding. To enforce a tit-for-tat mechanism to incentivize streaming nodes to share what they already have we require the seeding policy to also allow newer nodes in the cluster to participate in the network by sharing resources older nodes are interested in. Our seeding policy does this by always seeding parts of a stream that are required by nodes furthest along in their playback position, but seeding it only to the nodes that are the least further in their playback position. For example, if peer 1 is at playback position p_1 and peer 2 is at playback position p_2, with $p_1 < p_2$, and each is

interested in segment s_1 and s_2, respectively, with $s_1 < s_2$, then the seeder sends a chunk c_2 to p_1, where c_2 is a part of s_2. Then p_1 can save the piece and can use it as a bargaining tool to obtain a chunk c_1 that it is interested in by exchanging c_1 for c_2 with p_2. This is advantageous because not only is the seeder always introducing new resources into the network, but it gives newer peers a way to participate in exchanging resources with others according to some exchange policy.

Exchange Policy

The exchange policy is concerned with whether two streaming nodes should exchange resources and how they can do so. Aside from our assumption that streaming nodes are self-interested and exchange resources only if they they gain something from an exchange, we enforce the constraint that two nodes will exchange only if they either belong to the same swarm or belong to neighboring swarms. So our exchange policy must consider two cases. First, the exchange of chunks between nodes belonging to the same swarm. In this case as long as each has a chunk in the current segment that the other does not have they can exchange. This step ensures that a node can get chunks relevant to the current playback position and be able to start streaming the video.

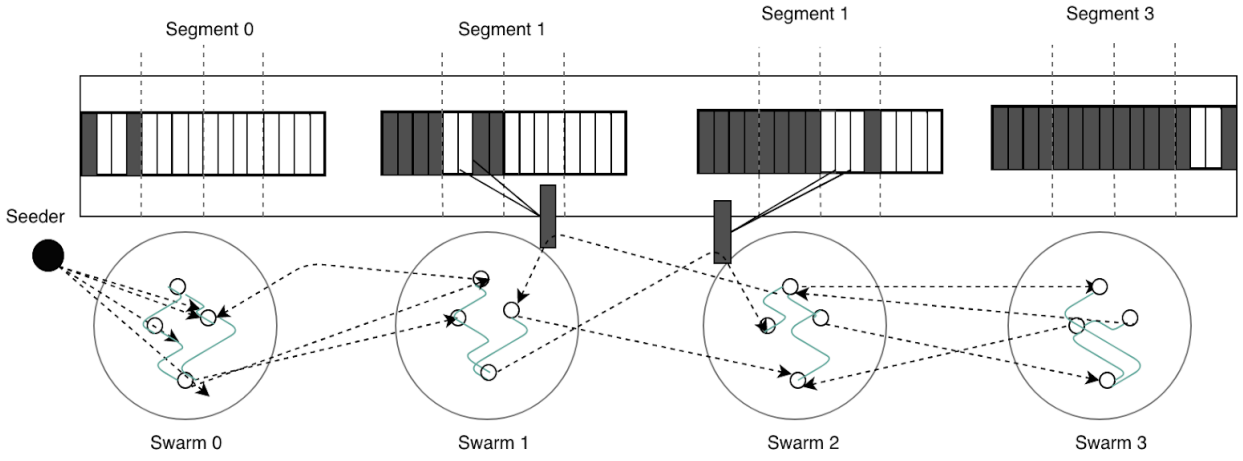

Figure 4. Nodes in Swarm 0 get chunks belonging to Segment 3. They trade this chunk with the neighboring Swarm to get the chunks for Segment 0. Nodes in Swarm 1 follow suit exchanging segment 3 with segment 1 chunks from Swarm 2. This continues until Swarm 3 gets all the chunks it needs. Green lines represents the internal exchange of the chunk among swarm members.

Second, the exchange of chunks between nodes p_i and p_j, $i < j$, belonging to neighboring swarms. In this case if the more advanced node p_j has chunks from segments more advanced than its own then it should offer the most advanced chunk $c_j > s_j$ that it can in exchange for a chunk c_i chosen by p_i that p_j does not currently have. Otherwise p_j should select a random chunk c_j relevant to segment s_j in exchange for c_i, if such a chunk

exists. By doing this in combination with our seeding policy, we ensure that each seeded chunk c_n that is required by nodes in the most advanced segment s_n is first used by nodes in the least advanced segment, s_1, as a way for them to obtain chunks they are interested by exchanging with nodes interested in segments s_1 and s_2. As this chunk is exchanged between neighboring swarms it makes its way to the most advanced swarm that wants it. Throughout this process each pair of nodes p_i and p_j, $p_i < p_j$ will exchange chunks c_i and c_j, where $c_i > c_j$, and so chunks seeded earlier and only to nodes with a more advanced playback positions are traded back down to the nodes in less advanced playback positions that want those chunks. This design incentivizes peers to keep participating and sharing resources because without doing so they cannot obtain chunks of the stream they are interested in and so will be unable to properly watch the stream.

Handling Failures

To manage peers in the network we use memberlist, a gossip-based cluster-membership library that implements the SWIM (Scalable Weakly-Consistent Infection-style Membership) protocol [1]. Our choice of memberlist over other open source implementations of SWIM was motivated by the peer discovery and failure detection mechanisms incorporated in this library. SWIM implementation in Memberlist has been extended by a set of features that attempts to reduce false positive in node failures. Aside from the regular gossip done over UDP, every once in a while nodes engage in a full-state sync with one of its peers over TCP. This increases the likelihood that the state of the network is converged and merged.

Prototype Evaluation

To test our system, we ran different instances of our code locally and on Azure cloud and verified that the nodes are made aware of each other and the files each node on the network hosts through exchange of gossip. Also when nodes fail to respond to gossip messages, other nodes update their internal state to reflect the failed nodes.

We ran the nodes locally and also on Azure cloud and we verified that once a node joins a seeder's cluster, it can correctly move from one swarm to the next and the chunks in each segment are cached locally in the correct directory. We also verified that the nodes that have joined the cluster earlier, move to more advanced cluster before nodes that joined the seeder cluster later.

Our implementation of swarming can handle the scenarios where all but one node in a swarm die and the surviving node will be able to carry on trading chunks they receive from neighbours or the seeder for the chunks in the segment they are streaming. In case a swarm other than least and most advanced swarm goes down, nodes in neighbouring swarm are be able to update their state and connect to one another, in effect repairing the

chain of swarms. The same logic holds for scenarios where all swarms except the most advanced fail; seeder and the surviving nodes will be notified of the failure and seeder will supply the chunks the remaining nodes need.

Nodes in our implementation can seed and leech at the same time. That is, leeching from another seeder does not prevent node p to accept streaming requests from other nodes in the network.

Demo Plan

Our demo will cover three scenarios. We will first show the normal operation of our system. In the context of P2P video-streaming we will show that a system of 3 nodes is able to exchange all the data, and ideally demonstrate this by actually streaming video content. Our script will launch one azure instance that offers to seed a video and will also launch two more client instances that should be able to learn about the seeded video and contact the seeder to begin streaming. We should the be able to check one of the client instances that it is obtaining the video file and ideally be able to watch the video streaming.

We will then show that our system can survive failures. In our case this means failure of some of the nodes who are all streaming from a single seeder. Our script will again launch a single instance that offers to seed a video. We will then launch multiple leeching instances all wanting to stream that video. We will stagger these launches so that nodes are interested in different parts of the video. Once running we will kill some number of these instances. After doing so we should expect to be able to continue streaming as per the previous case.

Finally, we will show that our system can use newly joined nodes. Our script will start by re-running the steps in the first case. We will then join a suitable number of new nodes in such a way that different groups of nodes are interested in the different parts of the video. We will then join a final node which we will use to stream the video ourselves. Because resources this new node is interested in already exists in the network and can be obtained by exchanging chunks given to it by the seeder, we should expect this final node to be able to feed the rest of the network and allow us to be able to stream the video.

Work in Progress

In preparation for our demo, we will be writing a bash script that will provision the Azure VMs with all the dependencies for our code.
We will continue to extend out logic to gracefully handle the case where seed node in the cluster dies.

References

1. Abhinandan Das, Indranil Gupta, Ashish Motivala “SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol.” *DSN* (2002).
2. Armon Dadgar, James Phillips, Jon Currey “Lifeguard: Local Health Awareness for More Accurate Failure Detection” cs.DC (2018)
3. Kévin Huguenin, Anne-Marie Kermarrec, Vivek Rai, Maarten Van Steen “Designing a Tit-for-Tat Based Peer-to-Peer Video-on-Demand System” 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), (2010), Amsterdam, Netherlands. <10.1145/1806565.1806589>. <inria-00467786>