

A VoD System for Massively Scaled, Heterogeneous Environments: Design and Implementation

Kangwook Lee, Lisa Yan, Abhay Parekh and Kannan Ramchandran
Department of EECS
University of California at Berkeley, Berkeley, CA
kwjjang@eecs.berkeley.edu, lyan2@berkeley.edu, {parekh, kannanr}@eecs.berkeley.edu

Abstract—We propose, analyze and implement a general architecture for massively parallel VoD content distribution. We allow for devices that have a wide range of reliability, storage and bandwidth constraints. Each device can act as a cache for other devices and can also communicate with a central server. Some devices may be dedicated caches with no co-located users. Our goal is to allow each user device to be able to stream any movie from a large catalog, while minimizing the load of the central server.

First, we architect and formulate a static optimization problem that accounts for various network bandwidth and storage capacity constraints, as well as the maximum number of network connections for each device. Not surprisingly this formulation is NP-hard. We then use a Markov approximation technique in a primal-dual framework to devise a highly distributed algorithm which is provably close to the optimal. Next we test the practical effectiveness of the distributed algorithm in several ways. We demonstrate remarkable robustness to system scale and changes in demand, user churn, network failure and node failures via a packet level simulation of the system. Finally, we describe our results from numerous experiments on a full implementation of the system with 60 caches and 120 users on 20 Amazon EC2 instances.

In addition to corroborating our analytical and simulation-based findings, the implementation allows us to examine various system-level tradeoffs. Examples of this include: (i) the split between server to cache and cache to device traffic, (ii) the tradeoff between cache update intervals and the time taken for the system to adjust to changes in demand, and (iii) the tradeoff between the rate of virtual topology updates and convergence. These insights give us the confidence to claim that a much larger system on the scale of hundreds of thousands of highly heterogeneous nodes would perform as well as our current implementation.

I. INTRODUCTION

The shape of the internet is changing. On the one hand, large internet exchanges and datacenters have made it possible to centralize a lot of compute and storage resources. It is clear that established players such as Netflix, Google and Amazon are more likely to exploit the reliability and economies of scale that come from such architectures, and that is of course what they are doing. On the other hand, the edge of the internet is growing exponentially. Phones, tablets, laptops, and ebook readers are growing in power and sophistication to the point that they resemble the desktop computers of a few years ago. But fully 50% of the traffic of the internet does not originate from data centers and hierarchical CDNs [1]. This traffic consists of applications such as file sharing and P2P, and relies more heavily on edge-devices which take on the role of potentially unreliable servers. As we play out the evolution of content and the internet, it seems clear that both kinds of content distribution will co-exist. Videos of police action in an oppressive state are easier to detect and throttle in a centralized architecture than a distributed one, and there will always be situations in which small groups of individuals will want to share content without the “prying eyes” of a media giant.

We have been working on a highly distributed edge-based scheme where the content is streamed video and has quality of service constraints. The demand for these videos is not specified to the system. Each edge device has limited storage but can store content from any movie (even ones that the owner of that device is not watching). The devices are assumed to have limited connectivity and the network is allowed to be unreliable. As demand and network connectivity fluctuate, so does what is stored at each node. In other words our goal is to test the limits of how unreliable and distributed we can make the infrastructure and still meet the stringent quality of service constraints of streamed video. Central to our architecture is the existence of a single reliable server or Seedbox [2], that can fill in the gaps of service when our distributed algorithms cannot meet QoS constraints and the objective of our distributed algorithms is to ensure that for any set of adverse network conditions, the load on the Seedbox is minimized.

Our approach is the following. First, we formulate the problem as a static convex optimization problem that is analytically tractable. In [3] we explained the theoretical justification of our approach, and “solved” this NP-hard problem through a novel relaxation based on a Markov approximation technique in a primal-dual formulation, that results in a highly distributed algorithm that converges to a near-optimal solution. The algorithm approximately specifies the optimal allocation of rate and storage resources at each node, as well as the best network topology that respects the network connectivity constraints. In this paper we focus on the systems level issues involved in taking an algorithm derived from theory to a full implementation. As an interme-
diate step, we have designed and implemented an extensive packet-level simulator of the system. This proved to be very useful in studying and learning the dynamic and robustness properties of the algorithm, and helped in the implementation phase. In this paper we present a number of results to show that our approach would work well if massively deployed in highly heterogeneous environments.

A. Problem formulation

Our goal is to build a system that jointly solves the following problems:

1) Content Placement: What content should be stored at each device/node given the storage constraints, network capacity and current demand?
2) Overlay Topology: Given that each node can support only a bounded number of end devices, how should end devices be matched to the nodes?
3) Minimal Server Load: When there is no available node that can serve an end device to watch a specific piece of content from, the Seedbox (or central server) “fills in the gap” by streaming directly to it. We wish to minimize the load on this sever.

We illustrate these problems further in the example depicted in Figure 1. The system has two 1 GB videos, A and B, which must be delivered at a streaming rate of 1 Mbps. There are 4 users: two request video A and two request video B. The three cache nodes are constrained by bandwidth, maximum degree (a bound on the number of simultaneously supported streaming connections) and storage. Figure 1b shows that under a certain “bad” topology and a “bad” content allocation scheme, demand cannot be satisfied. In Figure 1c, a “good” content placement strategy is chosen. Further, taken in isolation each problem is hard: there are an exponential number of possible topologies (from which we must select in a distributed manner) and as we will see (although it may be clear to some readers even at this point), the content selection problem for a fixed topology, is also NP-hard.

II. Related Work

Distributed video-on-demand systems such as those offered by Netflix rely on a network of reliable well-connected servers. As we have explained earlier, our work is not attempting to improve on these systems, rather it is to propose a solution under much more unreliable settings. Pure peer-to-peer networks such as BitTorrent are built for sharing files whereas our system includes a Seedbox server and is designed to accommodate quality of service constraints. Our system also goes beyond the traditional torrent architecture in that it accommodates inter-torrent caching and cross-torrent content sharing in a VoD setting.

The optimization of VoD systems has received wide attention in the academic literature. [4]–[10]. Almeida et al. [4] studied the delivery cost minimization problem under a fixed topology by optimizing over content replication and routing. Boufkhad et al. [5] investigated the problem of maximizing the number of videos that can be served by a collection of peers. Zhou et al. [6] focused on minimizing the load imbalance of video servers while maximizing the system throughput. Tan and Massoulie [9] studied the problem of optimal content placement in P2P networks. Their goal is to maximize the utilization of peer uplink bandwidth resources. Optimal content placement strategies are identified in a particular scenario of limited content catalog under the framework of loss networks. Their work assumes that the peers’ storage capacity grows unboundedly with system size. In contrast, our work does not make any assumption on the storage capacities and also takes into account the overlay topology. Applegate et al. [10] formulated the problem of content placement into a mixed integer program (MIP) that takes into account constraints such as disk space and link bandwidth. However, they assume knowledge of the popularity of content under a fixed topology, with a video being stored either in full or not at all. In our work, we use a class of network codes that enables fractional storage, and we further do not assume any prior knowledge on the demand distribution. We also optimize over the choices of all feasible topology graphs.

With regard to network resource utilization, Borst et al. [11] solved a link bandwidth utilization problem assuming a tree structure with limited depth. A Linear Program (LP) is formulated, and under the assumption of symmetric link bandwidth, demand, and cache size, a simple local greedy algorithm is designed to find a close-to-optimal solution. Valancius et al. [12] propose an LP-based heuristic to calculate the number of video copies placed at customer home gateways. The network topology in our work is not constrained to be a tree, and the video request patterns can be arbitrary in different network areas. Zhou and Xu [13] aimed to minimize the load imbalance among servers subject to disk space egress link capacity from servers. In contrast, we consider the link capacity constraints that may exist anywhere in the network.

Topology building is also an important design dimension and has been studied in various works [14]–[16]. While most works focus on enforcing locality-awareness and/or improving ISP-friendliness, they make the simple assumption that the graph is fully connected, i.e., no node-degree-bound is taken into consideration. Zhang et al. solves the problem of optimal P2P streaming under node degree constraints [17]. However their topology selection algorithm depends on global statistics which are easily accessible under a live-streaming scenario. However, directly applying their technique in the Video-on-Demand setting that is of interest in this paper, requires global statistics of all users’ utility functions, which can create and enormous overhead. Our distributed algorithm requires knowledge of only local information of neighboring overlay link rates.

To the best of our knowledge, we are unaware of any other work to jointly optimize topology graph selection, content placement and link rate allocation. Our solution is fully distributed and adapts well to system dynamics.
The routing matrix $A$ in the overlay graph by setting up TCP/UDP connections.

The link capacity constraints can exist arbitrarily anywhere as shown in (a). The problem is to decide, for each cache, which videos to store, which users to connect to, and how much bandwidth to allocate for each user. These questions are coupled. The connections between the cache nodes and users in (b) form a “bad” topology, and the content placement is non-optimal. The content placement in (c) is “good”. In (d), the topology is “good”, and with the same content placement strategy in (c), only one user is in deficit of half of a video. In general, finding the “best” storage, bandwidth and topology combination is a combinatorially-hard problem.

III. MATHEMATICAL PROBLEM FORMULATION

In this section, we cast the VoD optimization problem as a convex optimization problem. We analyzed this formulation extensively in [3] and showed that a distributed algorithm closely approximates the optimal. Our formulation assumes a static setting: the video catalog, the set of users and subscriptions, and the set of caches are fixed.

As illustrated in Figure 2, a set of caches($H$) and a set of users($U$) are connected by a fixed physical network which consists of links with capacity. Caches and users are connected by an overlay graph configuration, $g$, which is expressed by the set of overlay links $R$ and the corresponding routing matrix $A$. We denote the set of overlay links and the routing matrix by $R^g$ and $A^g$ under a graph configuration $g$. Each overlay link $r \in R$, consists of a set of underlay links, $L_r \subseteq L$ and we say $l \in r$ if $l \in L_r$. An overlay link $r = (h,u)$ enables cache node $h$ to send data to node $u$ in the overlay graph by setting up TCP/UDP connections.

The routing matrix $A := (A_{lr}, (l,r) \in L \times R)$ is defined as usual where $A_{lr} = 1$ if $l \in r$ and 0 otherwise. We denote the connected neighbor of node $v$ by $N^g_v$. Node $v$ cannot connect to more than $B_v$ users, which leads the node-I/O constraints. Let $G = \{g| |N^g_v| \leq B_v \ \forall v \in H \cup U\}$ be the set of possible overlay graphs. Let link $l \in L$ have a capacity $c_l$, and let $x_r$ be the rate on the overlay link $r$. This leads to natural routing constraints as follows: $Ax \leq c$, where $x$ is the column vector of the overlay rates $x_r$ and $c$ is the column vector of link capacity constraints $c_l$.

The video catalog consists of videos in the set $M$. Each video $m$ has size $\beta_m$ and is streamed at a constant rate of $\gamma_m$. Since we assume a fixed demand in this model, we denote the set of users watching $m$ as $U_m$. To model the storage constraints, let $s_h$ be the storage capacity of cache node $h$, and denote by $s := (s_h, h \in H)$ the column vector of the storage capacities. Let $W := (W_{hm}, h \in H, m \in M)$ be the storage matrix where $W_{hm} \in \{0,1\}$ indicates if video $m$ is stored on cache node $h$ ($W_{hm} = 1$) or not ($W_{hm} = 0$). Denote by $\beta := (\beta_m, m \in M)$ the vector of the sizes (in MB) of all videos. The storage constraints can then be expressed by $Wz \leq s$. Availability constraints are also modeled: caches can only serve stored movies. From cache $h$ to user $u$, the streaming rate can be only 0 if cache does not store the movie $m$ which is being viewed by the user. If the cache stores the movie, the streaming rate can be anything no greater than $\gamma_m$. It can be equivalently expressed as $x_r := (h,u) \leq W_{hm} \gamma_m$.

Let $z_u = \sum_{r=(h,u), h \in N^g_u} x_r$ be the total received rate of user $u$, and $V^u(z)$ be a concave function that represents the utility of user $u$ when the received rate is $z$. Table I lists all relevant notations.

Now, we have the following optimization problem. The objective is to find a graph $g$, content placement $W$, and rate allocation $x$, which jointly maximize the sum of user utilities under constraints.

$$\max_{g \in G, x \geq 0, W_{hm} \in \{0,1\}} \sum_{u \in U} V^u(z_u)$$

subject to

$$s.t. \quad x_r = (h,u) \leq W_{hm} \gamma_m, \quad A^g x \leq c, \quad W\beta \leq s.$$
TABLE I: Key Notations

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H, U, M$</td>
<td>set of caches / users / movies</td>
</tr>
<tr>
<td>$U_m$</td>
<td>set of users watching video $m$</td>
</tr>
<tr>
<td>$\gamma_m, \beta_m$</td>
<td>video $m$’s streaming rate and size</td>
</tr>
<tr>
<td>$\gamma_h, \beta_h$</td>
<td>storage capacity of cache $h$</td>
</tr>
<tr>
<td>$G, B_v$</td>
<td>set of feasible overlay graphs / I/O constraints</td>
</tr>
<tr>
<td>$L, c_l$</td>
<td>set of underlay links / link capacity</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Auxiliary Variables</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta_l$</td>
<td>shadow price of link $l$</td>
</tr>
<tr>
<td>$q_r$</td>
<td>$q_r = \sum_{l \in r} \theta_l$ is the shadow price of route $r$</td>
</tr>
<tr>
<td>$\lambda_r, \Sigma_h, m$</td>
<td>demand index of route $r$ / movie $m$ at cache $h$</td>
</tr>
<tr>
<td>$\omega_h$</td>
<td>storage price of cache $h$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decision Variables</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_r$</td>
<td>route rate of $r$</td>
</tr>
<tr>
<td>$W_{h,m}$</td>
<td>storage of video $m$ on cache $h$</td>
</tr>
<tr>
<td>$g, A^g, R^g$</td>
<td>overlay graph / routing matrix / overlay links</td>
</tr>
<tr>
<td>$N^g_v$</td>
<td>connected neighborhood of $v$ under $g$</td>
</tr>
<tr>
<td>$p_g$</td>
<td>probability of each topology graph $g$</td>
</tr>
</tbody>
</table>

the exponentially large size of the feasible graph set $G$ and integer constraints of storage matrix $W$.

IV. SYSTEM

In this section, we describe the overall architecture of our system. Before introducing the architecture and the system components, we first briefly illustrate how we apply codes under the context of video streaming and caching. The coding technique is used to eliminate the combinatorial nature of the resource allocation problem by allowing for “fractional” content streaming and caching. Then, we describe the overall architecture of our system, and provide a detailed explanation of each component. The distributed algorithms to optimally utilize cache resources are presented with details.

A. Streaming and caching with codes

![Fig. 3: A pictorial illustration of codes for video streaming.](image)

Each video is divided into multiple scenes and each scene is sliced and then encoded using MDS codes. In this example, MDS codes with $(n, k) = (32, 16)$ are used.

Figure 3 shows the conceptual way of applying codes for streaming videos. We first cut a video into multiple scenes, each with a fixed duration. Then, we slice each scene into $k$ equal sized chunks\(^1\). Lastly, for each scene, we encode $k$ chunks using $(n, k)$ MDS codes, and take $n$ coded chunks. Media servers will store all of these $n$ chunks of all scenes.

Caches can partially store a video by storing an equal number of chunks of all scenes. For example, Figure 4 shows how caches store a half of a video in our system. Assume that both Cache A and Cache B decide to store a half of a video. Then each cache will first randomly choose a subset of $\{1, 2, ..., n\}$ with size $k/2$. After the choice of a random set of indices, it will store the corresponding encoded chunks of all scenes. Users will need any $k$ encoded chunks to decode a scene.

B. Overall Architecture

![Fig. 4: Caches randomly choose and store a subset of the encoded chunks. Users successively download chunks from the connected caches and the server (if the caches cannot cumulatively provide the required $k$ chunks per scene). After receiving $k$ chunks, users decode the scene and watch it after the current scene.](image)

The system (Figure 5) includes four components: A light-weight Tracker, Servers, Caches, and Users. Each component can be on a separate machine or co-located. The tracker contains a list of videos and the identity of the server. (In case there are multiple seedbox servers, it would have to list multiple server identities). It also contains the IP addresses of each of the cache nodes in the system. Note that it does not contain information on which videos are stored on a given cache. In the following, we briefly explain the dynamics of the algorithm, and defer detailed explanations and pseudocode to subsequent sections.

\(^1\)The last chunk will be zero-padded to make all chunks equal-sized.
When a cache joins the system, it first registers itself to the tracker and immediately starts running the resource allocation algorithm (see Section IV-C for more details). The resource allocation algorithm is used to appropriately update how many chunks of which movies to cache, and to adjust upload rates to connected users in a distributed manner. Note that while running the algorithm, the cache does not need to communicate with the tracker. This is possible due to the fully distributed nature of our algorithm, which therefore reduces protocol chatter dramatically.

When a user joins the system, it retrieves from the tracker a list of videos available in the system. After choosing a video to watch, it registers itself to the tracker as a user watching the chosen video. Then it retrieves the IP address and the port number of one of the servers storing the chosen video, and connects to that server. After this server connection is established, the user retrieves a set of randomly sampled online caches (that may or may not have the desired content). After retrieving this set of available caches, the user picks a random subset of caches from this set and connects to them. The maximum number of caches that the cache connects to is strictly enforced either by the user device’s performance or by the user’s preferences.

After these connections to the server and caches have been established, a user requests the first video frame from the server so that it can start watching the video immediately (when a user controls its playback, the frame which corresponds to the playback’s time pointer is also requested from the server). While watching the first frame, it keeps downloading available chunks of the next frame from the connected caches at the rate determined by the caches. If the user successfully downloads the required number of chunks needed to decode that frame, it decodes and sends a ‘SATISFIED’ signal to the connected caches. On the other hand, if the connected caches fail to stream the required number of chunks needed for decoding the next frame by a certain deadline (e.g., a targeted number of seconds before the current frame is about to be played out), the user fills in the gaps by requesting the missing number of chunks from the server. Once the combination of the server and connecting caches supply the required number of chunks needed to decode the next frame, the user decodes this in preparation to watch the next frame, and signals an ‘UNSATISFIED’ message to the connecting caches.

Since it is possible that the initial set of caches that a user connects to do not have the video of interest, it also runs a topology update algorithm (see Section IV-D) to find a better matching set of caches that have the desired content. In [3], we proved that the topology updates are done in a manner that is guaranteed to approximately find the best matching set of caches for each user.

C. Cache Algorithms

The Resource allocation algorithm at each cache periodically updates 1) upload rates assigned to each connected user, and 2) the number of cached chunks for each video while satisfying storage, bandwidth, and connectivity degree constraints. Denote the upload rate to user $u$ as $x_u$, and the stored fraction of video $m$ as $W_m$. The upload rate variable $x_u$ is nonnegative and bounded by the streaming rate of the video being watched by user $u$. The cache variable of video $m$ is also nonnegative and bounded by 1, representing the fraction of cached video $m$. The cache node also updates some auxiliary variables needed to satisfy the resource constraints. The pseudo-code of the algorithm is described below. Every $T_{update}$ seconds, the cache node updates the upload rate variables and the storage variables. As the update equations indicate, these updates input the users’ signals relating to the current level of satisfaction (i.e., ‘SATISFIED’ or ‘UNSATISFIED’ as described earlier). The cache counts the number of received ‘UNSATISFIED’ signals for each movie, and increases the stored amount of that movie and the upload rate proportionally. Note that an outward drift of a variable is not applied if the variable is out of its feasible range. These cache updates are regulated by auxiliary variables that represent shadow prices of the resources. For example, if during an update phase, a cache exhausts all of its resources, these auxiliary variables increase, forcing the other variables to reduce their values in response to this. The pseudo-code of this part is presented in Algorithm 1.

All variables updated by the algorithm are in units of chunks. For example, the algorithm converts upload rates, which are real numbers, into units of ‘number of chunks per frame’. For example, consider a movie with a streaming rate of 2Mbps. Further, assume we use $(n, k) = (40, 20)$ codes to encode the video per frame. Recall that this implies that each frame consists of 20 chunks, which are encoded into 40 chunks with the property that any 20 of these 40 chunks suffice to decode the frame. If an upload rate variable indicates 1.5Mbps, the corresponding number of chunks of the frame to be sent to the user is $\frac{1.5\text{Mbps}}{2\text{Mbps}} \times 20\text{chunks} = 15\text{chunks}$. Similarly, the cache storage variables are also converted to physically meaningful units. $W_m = 0.25$ is equivalent to storing one quarter of video $m$, or storing 5 chunks of each frame. If the updated count on the number of chunks for a cache is greater than the actual number of stored chunks, the cache will download the missing number of chunks from the server. On the other hand, if the count becomes smaller, the cache will delete the appropriate number of stored chunks.

Updating (virtual) variables is repeated every $T_{update}$ time units. In our system, we set the default setting as $T_{update} = 0.01$ second. Applying these frequently changing variables is periodically done with longer periods. The upload rate variables are updated according to the corresponding variables every $T_{rate}$ seconds. Similarly, the cache variables are applied every $T_{storage}$ seconds. In the following performance evaluation section, we observe that $T_{storage}$ controls the tradeoff between the server to cache traffic and the update frequency of the caches.

If the resulting number is not an integer, we round it down the nearest integer.
Algorithm 1 Cache’s Resource Allocation Algorithm

1: \( c = \) Available upload bandwidth of the cache
2: \( s = \) Available storage of the cache
3: Wait \( T_{\text{update}}\).
4: for each connected user \( u \) do
5: \( m = \) ID of a video being watched by user \( u \).
6: \( y_u = \) “Satisfaction-level” binary signal of user \( u \).
7: Update rate \( x_u; \Delta x_u = \epsilon(g_u - q - \lambda_u) \).
8: Update availability price \( \lambda_u; \Delta \lambda_u = \epsilon(x_u - W_m \gamma_m) \).
9: end for
10: Update bandwidth price \( q; \Delta q = \epsilon(S - s) \).
11: for each video \( m \) do
12: \( U_m = \) Set of connected users watching video \( m \)
13: \( \Lambda_m = \sum_{u \in U_m} \lambda_u = \) Sum of availability prices
14: Update stored fraction \( \omega_m; \Delta \omega_m = \epsilon(\Lambda_m - \beta_m \omega_m) \)
15: end for
16: \( S = \sum_m W_m \beta_m = \) Sum of used storage.
17: Update storage price \( \omega; \Delta \omega = \epsilon(S - s) \).
18: Repeat.

D. User Algorithms

As described above, when a user starts watching a video, it downloads the frame from the connected server. After the download completes, it starts watching the first frame, while simultaneously downloading the next frame from the connected caches and the server. Upon a user’s request of a video, each cache sends 1) a list of cached chunks to the user, and 2) a suppliable upload rate, in units of number of chunks per frame the cache can offer the user. Then, the user decides which chunks to download from each cache. If the user is not able to download all its needed chunks from the caches, it sends an ‘UNSATISFIED’ signal to the caches, and downloads the missing number of chunks from the server. If the connected caches can cumulatively provide the required number of chunks, the user sends a ‘SATISFIED’ signal to the connected caches. As described in the cache’s algorithm section, these satisfaction-level binary signals will be collected by the connected caches and used as a way to measure the ‘local’ demand of a movie. Thus, we see that even before the current frame ends, the next frame is decoded and ready to be watched seamlessly, as needed to keep the streaming rate going. While repeating the procedure described above to have uninterrupted streaming, users also run the topology update algorithm to search for a better matching set of caches to connect to. The chunk selection algorithm and the topology update algorithm will now be described.

Chunk Selection Algorithm is used to determine which chunks to request from each cache with two major objectives. One is to maximize the number of unique chunks we get from the caches per frame, and a second is to maximize the usage of each cache according to its provided rate. We take a greedy approach to solving this problem and describe it briefly. We first sort the chunks by rarest first, where rarity is determined by that chunk’s occurrence across all caches for that frame. We then assign each chunk, from rarest to the most common, to the cache that currently has the lowest ratio of assigned upload rates to provable upload rates. The greedy algorithms achieve the maximum number of chunks to download while minimizing the relative gap between the provided rates by the caches and the actual rates assigned by the user.

Topology Update Algorithm is used to explore a better match between users and caches in terms of supply and demand. \( T_{\text{normal}} \) seconds after the initial connection with the initial set of caches is established, a user will randomly connect to a cache from its list of unconnected caches. Then it waits for \( T_{\text{transit}} \) seconds to give enough time for the new cache to adjust its assigned upload rates and stream the cached video chunks to the user. When this timer expires, the user drops one of the connected caches. Which cache is dropped depends on the quality of the match between the user and the cache, as measured by the current upload rate being supplied by that cache to the user. In our algorithm, we employ what we call a “soft choking” rule. While a hard choking rule would involve deterministically dropping the worst-performing cache, we use a softer rule where the probability of dropping a cache depends on its supplied upload rate (specifically, the choking probability is proportional to the negative exponent of the supplied upload rate for each connected cache). Under this rule, every connected cache can be dropped, even the best-performing one, but the worst-performing cache is the most likely to be dropped. This way of choking is theoretically justified by a recently proposed Markov Approximation Method [18]. The user repeats this process and the pseudo-code of it is presented in Algorithm 2.

Algorithm 2 User’s Topology Update Algorithm

1: Wait \( T_{\text{normal}} \).
2: Randomly choose and connect to a new cache.
3: Wait \( T_{\text{transit}} \).
4: for each connected cache \( c \) do
5: \( x_c = \) Upload rate provided by cache \( c \)
6: \( \rho_c = \exp(-x_c) = \) Probability of dropping cache \( c \)
7: end for
8: Normalize \( p_c \).
9: Pick and disconnect a random cache \( c \) with probability \( \rho_c \).
10: Repeat.

V. IMPLEMENTATION DETAILS

The Tracker is based on a stripped-down webserver based on \texttt{web.py}. That way, \texttt{http} commands can be used for updates and accesses. An important aspect of the implementation is the gathering of logs from the various caches, and this is also managed by the tracker.

Customized FTP Protocol: The rest of the algorithm is implemented by modifying the FTP protocol to support coded video streaming. Mainly, we added two FTP commands, \texttt{LIST\_CNKS} and \texttt{RETR\_CNKS} to \texttt{pyftpdlib}, an existing open
source FTP server library. When LIST_CNSK is called with the id of a video by a user, a cache responds with the list of cached chunk indices of the specified video. The RETR_CNSK command is used with the id of a video, index of a frame, and list of chunk indices to retrieve the set of chunks specified by the arguments. On the client side, the commands were implemented on ftplib, an open source ftp client.

Thus, the clients, caches and server(s) run instances of the modified server, and the user runs several instances of the modified client. All of the algorithms described in the previous section are fully implemented as protocols.

VI. SIMULATION RESULTS
A. Packet-level Simulator
The mathematical formulation, described in sec.III applies to a given demand profile. While we can prove that the distributed system we derive from it will work well in this static case, this is hardly good enough. It is important to study its performance under dynamic conditions such as user and cache churn, varying content popularity and network conditions.

As a stepping stone to bridging theory and practice, we have implemented a large-scale packet-level simulator in MATLAB, and equipped it with full functionality. The simulator allows us to test the robustness of the system against various dynamics by varying system environments such as number of users and caches, video popularity, and network conditions. The distributed nature of the system, allowed us to run relatively large scale simulations in parallel using multi-core processors. This section describes some of our findings.

We first run a small scale simulation to validate the simulator. The simulation setup is as following. There are 100 users in the system and each is watching one of 20 videos. Video popularity follows the Zipf’s law, which is a heavy-tail distribution. There are 50 caches, each of which can store up to 2 videos. The cache upload bandwidth is at most two times the video streaming rate. The server is connected to all users. Figure 6 shows how the system running algorithms evolves. After 2000 iterations, the system evolves such that the caches provide 95.4% of the overall traffic and the server needs to provide only 4.6% of the overall traffic.

Figure 6 shows the snapshots of the system at different times. The 100 circles on the bottom of each figure represent users. A color associated with a user visualizes the index of a video. 50 circles on the top represent caches. Above each cache circle, a box represents its disk. A small colored box within each big box represents stored amount of the corresponding video. A cache can fill up the storage up to the height of the box by caching several videos. An overlay topology between users and caches are represented as a bipartite graph between them. A red bar represents the server load. Via the visualized outputs, we were also able to check the validity of the algorithm and moreover find the interesting observations. The following sections will cover extensive simulation results.

B. A toy example
Another small scale simulation illustrates the complexity of the problem being solved by the system. Figure 7 shows a simple VoD system with three caches and six users. Let us first consider the resource allocation algorithm under a fixed overlay topology depicted in Figure 7a. Caches have to maximize the the joint upload rates of them so as to minimize the server load. The optimal caching and rate allocation can be easily found by finding the optimal solution via any optimization solver. The optimal solution turns out to be as following and depicted in 7a.

1) Cache 1: Store video A. Upload the full stream to user 1 and user 2.
2) Cache 2: Store half of video A and half of video B. Upload the fractional stream of video A with rate half to user 3. Upload the fractional stream of video B with rate half to user 4 and 5.
3) Cache 3: Store video B. Upload the fractional stream of video B to users 4 and 5. Upload the full stream of video B to user 6.
4) Server: Upload the fractional stream of video A with rate half to user 3.

Figure 7b shows the convergence of non-cache traffic if we run only the distributed resource algorithm under the same configuration. It is observed that the distributed algorithm also converges to the same optimal point where the non-cache traffic is 0.5Mbps.

Now, what is the optimal topology selection and resource allocation? Assume that degree bound of each cache is 3, 4, and 3 respectively. By considering all possible \( \binom{4}{0} \times \binom{4}{4} = 600 \) overlay topologies, one can find out the optimal topology and resource allocation scheme as following and it is depicted in figure 7c.

1) Cache 1: Connect to User 1, User 2, and User 3. Store video A. Upload the full stream to user 1 and the half stream to user 2 and user 3.
2) Cache 2: Connect to User 2, User 3, User 4, and User 5. Store half of video A and half of video B. Upload the half stream of video A to user 2 and user 3. Upload the half stream of video B to user 4 and user 5.
4) Server: Do nothing.

Here, the server traffic is 0Mbps.

Figure 7d shows the convergence of non-cache traffic if we run the distributed resource allocation algorithm and topology update algorithm together. It is observed that the distributed algorithm also converges to the optimal point where the non-cache traffic is 0Mbps quickly.

This confirms that the simulated distributed algorithm can indeed converge to the theoretical optimal operation point quickly. In the following sections, we now show results from extensive experiments at large scales having interesting consequences.
C. Robustness

Figure 8a shows the robustness of the system against changes in video demand. First we simultaneously add several videos and make them the most popular videos in the system. This results in a spike in server traffic. However, the distributed caches quickly adapt to the change in demand by altering the set of stored movies and in 200 iterations the system has converged to the new optimum. In fact, the same rapid adjustment is observed in Figure 8b when we invert the demand for all movies, i.e., the most popular movie becomes the least popular and so on. This is an extreme change in demand, especially given that the distribution is Zipf.

Figure 8c shows what happens a randomly chosen subset of half the caches in the system become simultaneously inoperative. Remarkably, the surviving caches are able to detect the sudden increase in demand of users, adapt their caches, and update their upload rates to the users quickly. Figure 8d shows how quickly the surviving caches adjust their storages to adapt.

D. Scalability

Finally, Figure 9a provides evidence of scalability. With 10,000 users, 5,000 caches, and 1,000 videos, the system is able to achieve offload more than 98% of the demand from the server to caches. However, more interestingly, we observe that the total amount of caching of a video in the system is roughly proportional to its popularity in Figure 9b. This is interesting because without any global knowledge of video popularity, the caches jointly figure it out in a fully distributed manner.

VII. TESTBED RESULTS

In this section, in addition to corroborating our analytical and simulation-based findings, we study the system’s
performance in practice and examine various system-level tradeoffs. We first observe that the server traffic to caches is not negligible in practice, and study the split between server to cache and cache to device traffic. Second, we study the tradeoff between the cache update intervals and the time taken for the system to adjust to changes in demand. Lastly, we test the topology update algorithm’s performance in practice. Even with the several practical constraints, the topology algorithm is able to reduce the server traffic. These insights give us the confidence to claim that a much larger system on the scale of hundreds of thousands of highly heterogeneous nodes would perform as well as our current implementation.

A. Experimental Setup

We deploy our full implementation of the system on 20 Amazon EC2 medium instances located in Northern California. We run 60 caches and 120 users on those instances. There are 20 high-definition (3.2Mbps) videos registered in the system, and users randomly choose a video following a Zipf’s distribution. This implies that the most popular video is watched by more than 25 users while the least popular videos are watched by only 1 or 2 users. When a user node finishes watching a video, it randomly chooses a new video to watch and connects to newly chosen caches.

Each cache can store up to 2 videos and has an upload bandwidth of four times of video streaming rate. We used FTP bandwidth throttles to simulate caches’ upload bandwidth in Amazon EC2. Each user can connect to up to 5 caches.

B. Traffic split between server to cache and cache to user

![Fig. 10: Server traffic to users and caches](image)

First, we look at how much traffic occurs from the server to the users and caches. In this experiment, each user repeats choosing and watching a video. Figure 10 shows how much traffic occurs from the server to the caches, compared to the traffic from the server to the users. At the beginning, the server traffic to the caches is 29.0% of the server traffic to users. This is because the server has to fill up the empty caches initially. However, as time goes on, the server traffic to the caches becomes negligible; it is less than 5.2% of the server traffic to the users. This observation supports our theoretical assumption that the server traffic to the caches is non-negligible only in the initial phase, unless the video demand distribution varies too rapidly.

C. Tradeoff between cache update intervals and the time taken for the system to adjust to changes in demand

Second, we answer how frequently the caches should update their storage. If the server to cache traffic is not considered, the caches can freely update their storage as frequently as they want without the system being penalized. However, in practice, if the caches update their storage too frequently, the server will be burdened due to its constantly needing to update the caches. A natural question that arises is: how frequently should the caches update their storage to be appropriately responsive to changes in demand without overburdening the server too much? We take a look at this trade-off by changing video popularity abruptly in the middle of the experiment. We gave 50% more resources to the caches so that the system can achieve near-zero server traffic. After the system reaches this state, we suddenly “invert” the video popularity histogram.

Figure 11a, 11b shows results with $T_{storage} = 0.5s$ and $T_{storage} = 5s$ respectively. With the shorter cache update period, caches are able to adjust their storages so quickly that the server traffic to the users is quite low. On the other hand, with the longer cache update interval, the caches do not adjust their storages frequently, and more users download videos directly from the server, resulting in high peaks. We believe the tradeoff observed here must be considered jointly with how quickly the video demand distribution changes.

D. Topology update algorithm in Practice

In practice, the topology update might hurt the system because a fast-varying topology can incur a higher server traffic to the caches because of the periodic soft-choking algorithm described earlier. Although it is true that eventually, upon convergence, the topology update algorithm is guaranteed to achieve a lower amount of server traffic to the users, what is not clear is whether the transient server traffic to the caches needed to attain this topology convergence is negligible. For testing the performance of the topology update in practice, we consider the case where the users watch the same video in a continuous loop.

Figure 12a, 12b shows the server traffic without the topology update algorithm and with topology update algorithm, respectively. It shows that, upon convergence, the topology update algorithm actually achieves 19.3% lower server traffic to users and 37.9% lower server traffic to caches. This is because users who are watching the same video are more likely to connect to the same caches, thereby reducing the traffic to the caches also. The results here imply that the topology update algorithm is still important even with the consideration of the server traffic to caches in practice. More extensive tests under more dynamic settings of demand churn are part of ongoing work.

VIII. Conclusion

In this paper, we proposed an architecture for distributed VoD when the components are potentially very unreliable,
and when storage, bandwidth, and node degree bound constraints may be severe. We started from a theoretical formulation from which we derived a set of distributed algorithms that are highly robust to changes in demand, user churn and device failures. In addition to exploring the behavior of the system via a packet level simulator, we also related our experience with a full implementation. Experimental results from the test-bed also provide valuable insights into design of a much larger practical system, which we argued could scale to large number of users and caches. We continue to gain experience from our test-bed with a view such larger deployments.

REFERENCES