A Peer-to-Peer Framework for Cost-Effective On-Demand Media Streaming

Xin Liu, Jun Wang, Son T. Vuong
Department of Computer Science, University of British Columbia
Vancouver, BC, Canada, V6T 1Z4
Email: {liu, jwang, vuong}@cs.ubc.ca

Abstract—This paper presents a novel peer-to-peer (P2P) framework for cost-effective on-demand media streaming, named BitVampire. BitVampire’s primary design goal is to aggregate peers’ storage and upstream bandwidths to facilitate on-demand media streaming. To achieve this goal, BitVampire splits published videos into segments and distributes them to different peers. When a peer (or a receiver) wants to watch a video, it first searches the corresponding segments, then selfishly determines the best subset of supplying peers and parallel downloads the desired media content from these peers in real-time mode. In BitVampire, participating peers help each other to get the desired content, thus powerful servers/proxies are not necessary, which makes it a cost-effective approach. To demonstrate the feasibility of BitVampire, We implemented a prototype using Java and JMF (Java Media Framework), and conducted some preliminary experiments.

I. INTRODUCTION

On-demand media streaming has recently gained intensive consideration due to its promising usage in a rich set of Internet-based services such as video on demand, distance learning, media distribution, etc. However, there are still many challenges towards building efficient, scalable on-demand streaming systems due to the high bandwidth and delay requirements for media streaming.

A majority of existing on-demand media streaming systems follows the Client-Server design, in which videos are stored in a set of central servers. All the requests are served by servers and the streaming content is directly delivered from servers. Obviously, this architecture is not scalable since the servers will become bottleneck as the requests increase. To save servers’ resources and alleviate servers’ traffic load, multicast has been applied and different solutions have been proposed. Batching [15] aggregates multiple client requests into one multicast session. Patching [6] allows a client to catch up with an on-going multicast session and patch the missing starting portion through server unicast. In merging [4], the client can repeatedly merge into a larger and larger multicast session. In periodic broadcasting [14], server separates a media stream into segments and periodically broadcasts them through different multicast channels, from which a client can choose to join in.

Besides the server-based solutions mentioned above, another technique to reduce servers’ traffic load is the cooperative proxy caching. Existing work in this area include prefix-based caching [13][11], and segment-based caching [1][3]. In prefix-based caching, proxies store the initial frames of popular clips. Upon receiving a request for the stream, proxy initiates transmission to the client and simultaneously requests the remaining frames from the server. In segment-based caching, parts of media content are cached on different proxies in the network and the stream is coordinated playback from these independent caches.

However, in both server-based solutions and proxy-based solutions, the servers and proxies are expected to deliver high-quality streaming service to a large number of clients. Therefore, the servers and proxies should be very powerful, which makes the deployment and maintenance cost very expensive. On the other hand, recent research and experiments reveal that the current Internet has enough resources to support large-scale media streaming in a peer-to-peer manner [12]. If these resources can be aggregated in a systematic way, an on-demand media streaming system could be constructed without powerful servers/proxies, which makes the system cost-effective. This is the primary design goal of BitVampire.

To achieve this goal, BitVampire splits the published videos into segments and distribute them to different peers. When a peer (or a receiver) wants to watch a video, it first searches the corresponding segments, then selfishly determines the best subset of supplying peers and parallel downloads the desired media content from these peers in real-time mode. In this paper, we present the system design of BitVampire, with the emphasis on the approach to distribute segments and the scheduling algorithm to aggregate bandwidths from multiple peers.

The rest of paper is organized as follows. Section II gives an overview of the system, identifying the system entities and operations. Section III briefly introduces Category Overlay [8], which is chosen as the underlying search infrastructure. The approaches to distribute and search segments are detailed in Section IV. In Section V, we discuss our scheduling algorithm, which coordinates different peers to deliver media content to receiver in real-time mode. We present the demonstration prototype in Section VI, and conclude this paper in Section VII.

II. SYSTEM OVERVIEW

This section provides a system overview of the proposed framework. We first identify all entities in the system. Then, we explain how the system works.
A. System Entities

Following are the entities in our proposed framework:

- **Peers.** This is a set of nodes currently participating in the system. Each participating peer contributes some of its upstream bandwidth and storage. We call the upstream bandwidth and storage peer \( P_i \) is willing to contribute as \( Bw_i \) and \( St_i \), and the available upstream bandwidth and storage peer \( P_i \) can contribute at a specific time as \( Bw_{avail} \) and \( St_{avail} \). Initially, \( Bw_{avail} = Bw_i \), \( St_{avail} = St_i \) and \( Bw_{avail} \leq Bw_i, St_{avail} \leq St_i \) hold at any time.

- **Seed Peers.** To handle the situation in which all the hosting peers of a specific segment leave the system, we introduce seed peers into the system. Seed peers always stay in the system, and each segment of published videos has a replica stored in seed peers. Seed peers serve the streaming request only when the request cannot be satisfied by regular peers, and if it is the case, seed peers will re-distribute a replica of that segment to peers, thus decreasing the future demand on seed peers. Seed peers are almost the same as regular peers, except that they are stable and have large storage, which is very cheap nowadays.

- **Media Files.** This is a set of videos currently available in the system. Every video is assigned a unique ID, called videoID, which is generated when the video is published. Every video belongs to a predefined type, such as Action video, Sports video, Comedy video, etc, and is associated with a list of keywords provided by the publishers. We assume that each video is encoded with a constant bit rate \( Br \) (in kbps). A video is split into equal sized segments, and segment is the minimum unit that a peer can cache.

B. System Operations

In BitVampire, when a peer publishes a video, the video will be split into equal sized segments, and these segments will be distributed to peers according to our segments distributing algorithm. Once peers receive segments, they will publish the received segments to the Category Overlay. Note that during the segments distributing process, every segment will have a replica distributed to one of the seed peers.

When a peer (called a receiver) wants to watch a video, it first searches the 1st segment. Then it determines if the streaming request can be satisfied by the peers contained in the search results (including seed peers). If the answer is yes, it selfishly determines the best subset of supplying peers and applies the proposed scheduling algorithm to aggregate bandwidths from the selected supplying peers and coordinate them to stream the 1st segment; otherwise, the request is rejected. When the streaming of the 1st segment is almost over, the receiver will do the same thing with the 2nd segment, the 3rd segment, and so on. Figure 1 shows an example. Suppose peer \( P_6 \) wants to watch a video whose playback bit rate is 500kbps. It searches for segment #0 and finds that \( P_1, P_2, P_3 \) have segment #0; it then selects \( P_1, P_2, P_3 \) as the supplying peers and aggregates bandwidths from them to stream segment #0. Segment #1 and #2 are streamed in the same way.

After the streaming session of a segment is over, the receiver will cache the segment in its contributed storage. We use LRU (Least Recently Used) algorithm to select the victim segment to replace if there is not enough available storage for the new cached segment.

III. Category Overlay

In this section, we briefly introduce Category Overlay, which is chosen as the underlying search infrastructure in BitVampire.

The basic idea of Category Overlay is to construct multiple category specific overlays on the unstructured P2P system and restrict a specific search within the corresponding overlay. In more detail, we first cluster the peer group into clusters. Then in each cluster, nodes (called Agent Nodes) are selected to take charge of predefined categories. The Agent Node is responsible for maintaining a keyword list table (called Content Index Table) for all the contents belonging to the categories it is in charge of. For a specific category, all of its Agent Nodes (in different clusters) are connected to form a category overlay. Thus, multiple category overlays can be constructed over the clusters.

Figure 2 shows an example of Category Overlay. As the figure shows, peers are clustered into three clusters: \( C_1, C_2 \) and \( C_3 \). In each cluster, nodes are selected to take charge of three predefined categories: \( Ca_1, Ca_2 \) and \( Ca_3 \). For example, in cluster \( C_1 \), node \( N_1 \) is in charge of category \( Ca_1 \); in cluster \( C_2 \), node \( N_2 \) is in charge of category \( Ca_2 \); and in cluster \( C_3 \), node \( N_3 \) is in charge of category \( Ca_3 \). Since \( N_1, N_2 \) and \( N_3 \) are Agent Nodes for category \( Ca_1 \), they are connected to form the category overlay \( O_1 \). Category overlay \( O_1 \) and \( O_2 \) can be formed in the same way.

In Category Overlay, every cluster member node maintains a Category Table, which stores the Category-to-Agent mappings. It looks like a hash table where the key is a category and the value is that category’s Agent Node. When a node publishes a content belonging to a specific category, it first looks up its Category Table to find that category’s Agent Node, then it sends a “publish content” message to the found Agent Node, along with the keyword list (while the content is still in owner’s storage). Upon receiving this message, the Agent Node will store the keyword list in its Content Index Table.

When a node issues a query, it specifies a category, as well as a list of keywords. The query will go to the Agent Node which is in charge of that specified category. Then the corresponding Agent Node looks up its Content Index Table to find the contents which have the matched keywords, and returns the results to the

---

**Fig. 1. Example of watching a video**
query initiator. In addition, that Agent Node also needs to propagate the query within the corresponding overlay. Each Agent Node in this overlay will look up its Content Index Table and return the results to the query initiator. Compared to Gnutella [5], in which queries need to go through all the nodes, a query in Category Overlay just needs to be propagated within the corresponding overlay, which is much more efficient.

Note that in Category Overlay, each cluster is tree-based (the root of the tree is called Core Node). Two cluster members are connected with Cluster Links (tree branches), and two neighbour clusters are connected through Inter-Cluster Links. More information about Category Overlay and its maintenance mechanisms, as well as the simulation-based performance evaluation can be found in [8].

IV. SEGMENTS DISTRIBUTING AND SEARCHING

A. Segments Distributing

In BitVampire, each participating peer Pi estimates its stay time in the system by computing the smoothed weighted average as follows and uses this value to represent its stability.

\[
\text{EstimatedStay}_i = \alpha \times \text{EstimatedStay}_i + \beta \times \text{CurrentStay}_i, \quad (1)
\]

where EstimatedStayi is the estimated stay time of peer Pi, taking into account all the stay history of Pi, and CurrentStayi is the time period Pi participated in the system since its last leave or failure. \(\alpha + \beta = 1\), \(\alpha\) is between 0.8 and 0.9, and \(\beta\) is between 0.1 and 0.2. Besides, peer Pi also maintains the average usage ratio of its contributed bandwidth since it participated in the system, called \(R^{usage}_i\), and the frequency it serves streaming requests in the recent period, called \(Freq_{serve}_i\).

We define \(G_{Sti}\), the goodness of candidate peer Pi to store a segment as a function of its \(EstimatedStay_i\), \(Bw_i\), \(R^{usage}_i\), and \(Freq_{serve}_i\). Suppose there are m candidate peers: \(\{P_1, P_2, \ldots, P_m\}\), \(G_{Sti}\) has the following form:

\[
G_{Sti} = \alpha_0 \times \frac{\text{EstimatedStay}_i}{\max_{m} \text{EstimatedStay}_j} + \beta_0 \times \frac{Bw_i \times (1-R^{usage}_i)}{\max_{m} [Bw_j \times (1-R^{usage}_j)]} - \gamma_0 \times \frac{Freq_{serve}_i}{\max_{m} [Freq_{serve}_j]} \quad (2)
\]

where \(\alpha_0\), \(\beta_0\), \(\gamma_0\), are the factors to give \(EstimatedStay_i\), \(Bw_i\times(1-R^{usage}_i)\), \(Freq_{serve}_i\) different weights. Given this formulation, the candidate peer that is more stable, has higher available bandwidth and lower serve frequency will have a greater \(G_{Sti}\).

To detect node failure, every peer in the Category Overlay periodically sends “alive” messages to its parent. We let peer send its \(EstimatedStay\), \(Bw\), \(R^{usage}\), and \(Freq_{serve}\) along with the “alive” message. The parent collects information contained in the received “alive” messages and periodically sends an aggregate report to its parent, along with the “alive” message. Thus, eventually Core Node will have recent information of every cluster member. Core Node sorts the cluster members by their \(G_{St}\) in descending order and stores the result in a sorted candidates list. Core Node periodically maintains this list based on the renewed information of cluster members.

When a peer wants to publish a video, it splits the video into equal-sized segments. The workload analysis of today’s enterprise media server [2] found that most of clients only watch the first several minutes portion of media files. To benefit from this fact, we let the first segment have several replicas. Suppose the video is split into \(N_s\) segments, and the first segment has \(N_f\) replicas, then totally \(N_s+N_f\) segments need to be distributed to peers.

The publisher peer sends a “publish video” message to its cluster’s Core Node, and this message will be propagated to the Core Nodes of other clusters. After receiving this message, the Core Node selects the first \(N_s(N_s>N_f+N_f)\) peers from the sorted candidates list and sends the information of these peers back to the publisher. The publisher waits for \(Timeout_p\) to receive the messages sent back by the Core Nodes and collects information of the candidate peers. After \(Timeout_p\), the publisher will assign segments to candidates.

Figure 3 illustrates the pseudo code of our segments distributing algorithm. Note that the algorithm tends to assign more segments to the candidate peers which have higher \(G_{St}\), which means these peers will take more responsibility to serve streaming requests. However, their \(Freq_{serve}\) will increase as the streaming requests come in, thus decreasing their \(G_{St}\). When another video is published, it is likely that their \(G_{St}\) will be
exceeded by others, so that video’s segments will be distributed to other peers. In a long term, this could result in load balance in peers to some extent.

B. Segments Publishing and Searching

After the segments are assigned to peers, the publisher will send segments to peers. When a peer receives a segment, it first decreases its $S_{i}^{avail}$ by $\text{seg\_size}$, where $\text{seg\_size}$ is the size of a segment. Then it publishes the received segment to the Category Overlay.

As mentioned in Section III, to publish content in Category Overlay, a node should specify the category the content belongs to, as well as a keyword list. In order to use Category Overlay, we define the categories as follows: (1) we first predefine the video types, such as Action video, Sports video, etc; (2) then we combine the video type and segment number as the category, such as Action-0, Action-1, Sports-0, etc. So all the first segments of Action videos belong to Action-0 category; all the second segments of Action videos belong to Action-1 category; and so on.

When a peer publishes a video, it will specify the video type and provide a list of keywords. When the publisher distributes segments to peers, the specified video type and keyword list will be sent to peers as well. When a peer publishes the received segment, it will use the combination of video type and segment number as the segment’s category, and use the received keyword list as the publishing keywords. In addition, each published segment has the video’s $\text{videoID}$ to specify which video it comes from, thus when searching, we can use this $\text{videoID}$ to ensure that the found segments come from the same video. After segments have been published, we can search the desired segments in the same way described in Section III.

V. SEGMENTS STREAMING

A. Supplying Peers Selection

When a peer (receiver) searches the desired segments for watching, the size of the results could be large. Thus we need a scheme to select supplying peers from the search results. We let the receiver selfishly determines the best subset of supplying peers. The detail of our scheme is presented below.

After receiving the search results, the receiver will send an enquiry message to each peer contained in the results. Upon receiving this message, a peer will send a reply message back to the receiver, along with its $B_{i}^{avail}$ and $\text{EstimatedRTT}$, where $\text{EstimatedRTT}$ is the estimated round trip time between the peer and the receiver. The receiver waits for $\text{Timeout}_r$ to get the reply messages and collect information contained in the messages. After $\text{Timeout}_r$, the receiver will select the subset of supplying peers based on their $G^S_i$, the goodness of peer to become supplier. Suppose there are $m$ candidate peers: $\{P_1, P_2, \ldots, P_m\}$, the $G^S_{i}$ for a peer $P_i$ is defined as follows:

$$G^S_{i} = \alpha_{Sp} \times \frac{B_{i}^{avail}}{\max_{i \in \text{candidate}}\{B_{i}^{avail}\}} - \beta_{Sp} \times \frac{\text{EstimatedRTT}_{i}}{\max_{i \in \text{candidate}}\{\text{EstimatedRTT}_{i}\}}$$  (3)

where $\alpha_{Sp}$, $\beta_{Sp}$ are the factors to give $B_{i}^{avail}$, $\text{EstimatedRTT}_{i}$ different weights. Given this formulation, the candidate peer which is nearer to the receiver, has higher available bandwidth will have a greater $G^S_i$.

The receiver will select $M$ (in our prototype implementation, $M = 3$) candidate peers that have the greatest $G^S_i$ as the suppliers, as long as the aggregated available bandwidth from these peers is bigger than or equal to the video playback bit rate. Otherwise, more than $M$ peers will be selected to meet the playback bit rate requirement. The unselected candidate peers will be kept in a standby set, from which substitute peers can be selected in case of suppliers leave or failure. If the aggregated available bandwidth from all of the candidate peers is less than the playback bit rate, the segment watching request will be rejected.

After supplying peers have been selected, the receiver will reserve bandwidths from them. Suppose $M$ supplying peers ($P_1, P_2, \ldots, P_M$) are selected, and the video playback bit rate is $Br$. The receiver will reserve bandwidth $B_{i}^{r}$ (the reserved bandwidth should be in multiple of bandwidth reservation unit $B_{i}^{w}$) from supplier $P_i$, in proportion to their $G^S_i$, and satisfy following condition:

$$\sum_{i=1}^{M} B_{i}^{r} = Br$$  (4)

Then the receiver sends a “reserve bandwidth” message to each supplier. Upon receiving this message, supplying peer $P_i$ decreases its $B_{i}^{avail}$ by $B_{i}^{r}$. Once the streaming session supplied by peer $P_i$ is over, $P_i$ increases its $B_{i}^{avail}$ by $B_{i}^{r}$. Note that by “reserve bandwidth”, we do not mean that the bandwidth $B_{i}^{r}$ from peer $P_i$ is actually reserved and cannot be used by other applications. The current Internet does not provide resource reservation service, thus the bandwidth contributed by supplying peer $P_i$ may fluctuate during the streaming session.

B. Scheduling Algorithm

To fully utilize the aggregate bandwidths from multiple supplying peers, we want different suppliers to send different portion of a segment to the receiver at the same time. So we further divide each segment into equal-sized blocks. Thus the receiver can parallel download different blocks from different supplying peers in real-time model.

Figure 4 illustrates the pseudo code of our scheduling algorithm. This algorithm will be executed by the receiver to generate the schedule. Note that this algorithm assigns blocks to suppliers in a roughly round robin manner, and also in proportion to the bandwidths contributed by the suppliers.

After schedule is generated, the receiver will send it to suppliers. After receiving schedule, a supplying peer will send the assigned blocks to the receiver in order using UDP, and perform TCP-friendly congestion control over the UDP connection (e.g., RAP [10] or TFRCP [9]). The receiver maintains a ring buffer. Once receiving a block, the receiver writes this block to the ring buffer at the right position. After receiver gets the first $S_{\text{min\_buf}}$ blocks, it will continuously read data from the ring buffer and render video frames on the player.
window.

During the streaming session, the receiver monitors the incoming rate from each supplier. If the receiver detects that the incoming rate from a supplier is decreasing for an enough long period $T_{\text{dec}}$, or it is notified or detects the leave or failure of a supplier, it will select substitute supplying peers from the standby set and reserves bandwidths from them (the total new reserved bandwidth should be bigger than or equal to the bandwidth provided by the supplier that is substituted). Then it will generate a new schedule to assign the rest not-received blocks to the new set of suppliers, and sends the schedule to the suppliers.

VI. DEMONSTRATION PROTOTYPE AND EXPERIMENTS

To demonstrate the feasibility of BitVampire, a prototype was implemented using Java and JMF (Java Media Framework) [7]. In the prototype, control packets are sent using TCP and streaming packets are sent using UDP. To ensure TCP-friendly congestion control over UDP connection, RAP protocol [10] was used to adjust UDP packets sending rate. Figure 5 is the snapshot of the prototype, where three nodes are running and two of them are watching videos.

We also conducted some preliminary experiments in a local network environment, which consists of 12 Desktop PCs residing in different labs of our department (which has two buildings). The experiment results show that based on these common low-cost PCs, BitVampire can achieve smooth video playbacks in the receivers.

VII. CONCLUSION

In this paper, we propose BitVampire, a novel peer-to-peer framework for cost-effective on-demand media streaming. In this framework, published videos are split into segments and distributed to peers, thus upstream bandwidths form multiple peers can be aggregated to serve a single streaming request. We propose a segments distributing algorithm to distribute segments, and rely on the Category Overlay to search the desired segments. To parallel download streaming content from multiple supplying peers in real-time mode, we further divide segment into blocks and propose a scheduling algorithm to assign blocks to different supplying peers to send. To demonstrate the feasibility of BitVampire, we implemented a prototype using Java and JMF and conducted some preliminary experiments in a local network environment.

REFERENCES