A Social-Network-Aided Efficient Peer-to-Peer Live Streaming System

Haiying Shen, *Senior Member, IEEE, Member, ACM*, Yuhua Lin, and Jin Li, *Fellow, IEEE*

***Abstract*—In current peer-to-peer (P2P) live streaming systems, nodes in a channel form a P2P overlay for video sharing. To watch a new channel, a node depends on the centralized server to join in the overlay of the channel. In today's live streaming applications, the increase in the number of channels triggers users' desire of watching multiple channels successively or simultaneously. However, the support of such watching modes in current applications is no better than joining in different channel overlays successively or simultaneously through the centralized server, which if widely used, poses a heavy burden on the server. In order to achieve higher efficiency and scalability, we propose a Social-network-Aided efficient liVe strEaming system (SAVE). SAVE regards users' channel switching or multichannel watching as interactions between channels. By collecting the information of channel interactions, nodes' interests, and watching times, SAVE forms nodes in multiple channels with frequent interactions into an overlay, constructs bridges between overlays of channels with less frequent interactions, and enables nodes to identify friends sharing similar interests and watching times. Thus, a node can connect to a new channel while staying in its current overlay, using bridges or relying on its friends, reducing the need to contact the centralized server. We further propose the channel-closeness-based chunk-pushing strategy and capacity-based chunk provider selection strategy to enhance the system performance. Extensive experimental results from the PeerSim simulator and PlanetLab verify that SAVE outperforms other systems in system efficiency and server load reduction, as well as the effectiveness of the two proposed strategies.**

***Index Terms*—Peer-to-peer (P2P) live streaming, P2P networks, social networks.**

I. Introduction

PEER-TO-PEER (P2P) live streaming applications [1]–[3] such as PPLive and UUSee are attracting millions of viewers every day. The success of these applications is rooted in the decentralized nature of P2P networks, which relieves the load on the centralized server by utilizing the upload capacity of participating users. Nowadays, in a live streaming system, hundreds of media channels are broadcast to millions of users at the same time [4]. For example, UUSee simultaneously sustains 500 live streaming channels [2] and routinely serves millions of users each day [5].

Manuscript received March 02, 2013; revised August 18, 2013 and February 09, 2014; accepted March 10, 2014; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Meo. This work was supported in part by the US NSF under Grants IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056, and CNS-1025652 and Microsoft Research Faculty Fellowship 8300751.

H. Shen and Y. Lin are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634 USA (e-mail: shenh@clemson.edu; yuhual@clemson.edu).

J. Li is with Microsoft Research, Redmond, WA 9805 USA (e-mail: jinl@microsoft.com).

This paper has supplementary downloadable material available at http://ieeexplore.ieee.org.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2014.2311431

Fig. 1. Screenshot from PPStream.

In current P2P live streaming systems, all nodes watching a channel form into a P2P overlay for streaming video sharing between each other. To watch a new channel, a node needs to contact the centralized server for the nodes in the channel in order to join in the channel's overlay. For example, nodes in UUSee need to contact the server to obtain tens of nodes, which incurs a large amount of communication overhead on the server. The current wide coverage of broadband Internet enables users to enjoy live streaming programs smoothly, and the increase of channels triggers users' desire of watching multiple channels successively or simultaneously (i.e., multichannel watching mode). A typical multichannel interface contains one main view and one or more secondary views [i.e., picture in picture (PIP)] as shown in Fig. 1, so that users can switch freely between main view and PIPs.

However, since most current P2P live streaming systems only allow users to share the stream in one channel, the support of successive and simultaneous watching modes in current applications is no better than joining in different channel overlays successively or simultaneously through the centralized server. Although today's PPStream [3] application can support PIP, it also uses this strategy. A node watching multiple channels stays in multiple P2P overlays, and thus needs to maintain multiple overlays. As a node opens more channels, its maintenance cost for overlay connections increases dramatically. Also, the server receives more requests from nodes to join in new channels. Thus, the successive-channel or multichannel watching of millions of users poses heavy burden on the centralized server, and delayed response leads to inefficiency in P2P live streaming systems [6].

In this paper, we aim to improve the efficiency and scalability of P2P live streaming systems with many users engaging in many successive-channel watching or multichannel watching by releasing the load on the centralized server. We propose a Social-network-Aided efficient liVe strEaming system (SAVE).

The key of its design is the utilization of social network concepts. By considering channels as nodes in a social network, SAVE regards users' channel switching or multichannel watching as interactions between channels. By considering users as nodes in a social network, SAVE identifies users with the same interests and watching times as friends with social connections. Note that we leverage social network behavior properties rather than online social networks. Specifically, SAVE incorporates two main schemes: channel clustering and friendlist.

Channel Clustering Scheme: A node's watching activity is driven by its interests [7], [8]. Thus, nodes with similar interests tend to routinely watch the same channels and may watch them in the same time periods. Also, the channel watching activities of each node are mostly limited to a small number of channels that it is mostly interested in [9]–[11]. Therefore, SAVE clusters channels with frequent interactions. It merges channels with high frequent interactions into one overlay and builds bridges between the channels with less frequent interactions. Thus, in successive- or multichannel watching, nodes can stay in the same overlay or take the interchannel bridges to join in a new overlay without relying on the server with high probability. We propose a centralized algorithm and a decentralized algorithm for the channel clustering.

Friendlist Scheme: The navigability property in a small-world network shows that a node is able to find a path to a destination node within a short number of steps [11]–[13], which indicates that a node in a channel can find a node in another channel in a few steps via friend connections. Therefore, each node in SAVE maintains a friendlist that records nodes sharing common-interest channels and watching time periods. When a node wants to switch to a channel that is not in its current cluster or when a node returns to the system after departure, it refers to its friendlist to find nodes in the desired channel to join in the overlay. We propose an algorithm for identifying friends for the friendlist construction.

From the perspective of the entire system, for the individual nodes' skewed interests, some interests are shared by a large portion of the nodes in the system, while others are shared by a small portion of the nodes. The former interests are handled by the channel clustering scheme, and the latter interests are handled by the friendlist scheme. We further propose channel-closeness-based chunk-pushing strategy and capacity-based chunk provider selection strategy to enhance the system performance. The two schemes with the strategies contribute to the following three main features of SAVE, and hence enhance the system efficiency and scalability as well as satisfactory user experience.

- Low overhead. In SAVE, nodes can stay in the same overlay when they switch channels or watch multiple channels in most cases, which greatly reduces the overhead caused by frequent join and leave operations and overlay maintenance.
- Quick response. When switching channels, users experience delay, which is decided by both the buffering speed and the latency of joining a channel [6]. Switching channels in SAVE in most cases does not need users to leave their current overlay and join in a new overlay, leading to low delay and better user experience.
- Light server load. Light server load can greatly reduce the bandwidth and hardware cost and improve system scalability. In SAVE, nodes can join in a new channel overlay without the participation of the server most of the time, reducing the server load.

We conducted a survey on user streaming video watching activities. The survey result shows that: 1) users tend to watch different channels successively; 2) the distribution of a user's interests is skewed; and 3) many users share the same interests and watching times. The survey results are consistent with the study results in the previous studies [14]–[16]. They confirm the social network properties in P2P live streaming systems and demonstrate the feasibility and necessity of SAVE to a certain extent. We conducted simulations in PeerSim [17] and deployed SAVE on PlanetLab [18]. Extensive experimental results prove that SAVE outperforms other systems in terms of system efficiency and server load reduction.

The problem handled in this paper is channel switching and multichannel watching in users' routine video watching activities. Note that Zapping [19] is a common behavior in multicast-based IPTV when users surf channels and pick their favorite one. Though zapping is not the main focus of our work, the SAVE's design actually does not exclude the zapping behaviors. When there a large number of zapping activities between two channels, SAVE can be effective in reducing startup delay in zapping.

The remainder of the paper is organized as follows. Section II gives an overview on the related work. Section III describes the design of SAVE. The performance evaluation is presented in Section IV. Section V concludes the paper with remarks on future work. Sections VI and VII in the supplemental file present our survey results on live streaming video watching activities and additional experimental results of the performance of SAVE built on the DHT structure of DCO [20].

II. Related Work

P2P Live Streaming Channel Overlays: P2P live streaming protocols fall into four categories: tree-based [21]–[24], mesh-based [25]–[32], hybrid structure combining both mesh and tree structures [15], [33]–[37], and distributed hash table (DHT)-based structure [20].

Tree-based methods deliver video content via push mechanism, in which parent nodes forward received chunks to their children. Early proposed tree-based methods such as Narada [23] and multicast [22] rely on a single tree structure, which is vulnerable to churn. Recent works [24], [34] build multiple trees, in which leaf nodes join in several trees to improve the system resilience to churn.

Mesh-based methods [26]–[28] connect nodes in a random manner to form a mesh structure. Each node usually serves a number of nodes while also receiving chunks from other nodes. eQuus [29] further facilitates clustering and locality-aware mechanisms, while other works [30]–[32] introduce improved packet scheduling protocols. Mesh-based methods are resilient to churn, but generate high overhead by frequent content publishing.

Hybrid methods synergistically combine tree-based and mesh-based structures. Wang *et al.* proposed a hybrid structure consisting of two tiers [15], [35]. It utilizes stable nodes to constitute a tree structure to push down video chunks, while all nodes also form a mesh overlay for chunk exchanges. PRIME [33] is a hybrid system featured in segmented and two-phase chunk delivery. Shen *et al.* [20] proposed to use DHT structure to index video chunks for P2P video streaming systems. SAVE can be implemented on top of these different overlay-based systems to improve their performance.

Multichannel P2P Live Streaming Techniques: One group of works is for bandwidth allocation optimization. In order to optimize the allocation of each node's upload capacity to each channel it is watching, Wu *et al.* [38] used game theory to resolve the conflicts in allocating bandwidth. By noticing that overlays are overlapped in multichannel applications, DAC [39] divides the nodes in the overlap to several virtual logical nodes with each in a single channel overlay. Then, by mapping the service relationship among the independent overlays, DAC models the bandwidth allocation problem to a solvable global optimization problem. AnySee [27] uses the mesh overlay structure and incorporates a location matching algorithm to map logical overlay to the physical location topology. Path selection is based on physical location topology.

Liang *et al.* [40] envisioned a framework of future video streaming, in which users can freely choose to watch one or more channels simultaneously. Wang *et al.* [41] proposed selecting neighboring nodes based on the channel subscription and residual bandwidth. To maintain system stability with high churn, Wu *et al.* [42] developed queuing models to analytically study the performance of multichannel P2P live video systems and derived near-optimal provisioning rules for assigning peers to channels. SAVE is different from the above works in that it aims to achieve efficient node joining in channels in channel switching or multichannel watching. Ramos *et al.* [43] proposed a method to send channels adjacent to the requested one to the Set Top Box (STB) during zapping periods in IPTV networks, so that STB can turn the source signal into content to be displayed on the screen during user channel switching. Kermarrec *et al.* [44] proposed a method in which each peer maintains a predefined metric (e.g., proximity, latency, or bandwidth) and chooses some peers based on the metric in other channels as contacts upon channel switching. Chen *et al.* [19] proposed OAZE, where each peer maintains connections to other physically close peers in a certain number of channels, which its associated user is likely to watch. When a node wants to switch to a channel, it tries to find neighbors connecting to the target channel for the switch. However, a node's neighbors may not connect to the target channel, so OAZE cannot provide a high probability for a node to find a neighbor to switch to its target channel. SAVE differs from the above systems in that it leverages the social network in channels and the social network in users to facilitate the channel switching. SAVE's channel clusters and friendlists help nodes to join in their target channels without relying on the server with a very high probability. Also, SAVE produces less overhead than OAZE since a node needs to contact fewer nodes for a channel switching.

Social-Network Aided Video-on-Demand Systems: Several social-network-based video-on-demand systems have been proposed. Cheng *et al.* [45] investigated the social networks in YouTube videos. They found that the links to related videos specified by uploaders have small-world characteristics. They further proposed NetTube [7] that uses the links between related videos to generate a social network of nodes, and then uses a social-network-assisted prefetching strategy to achieve smooth transition between video playbacks. Salvador *et al.* [46] proposed to use node characteristics including geographical location, node availability, and distance for video file sharing.

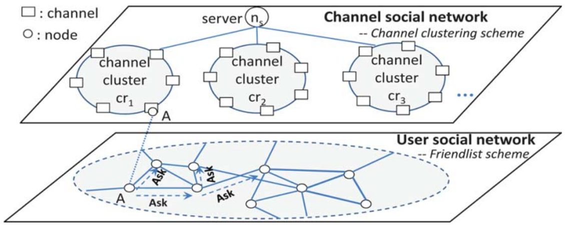


Fig. 2. High-level view of the SAVE structure.

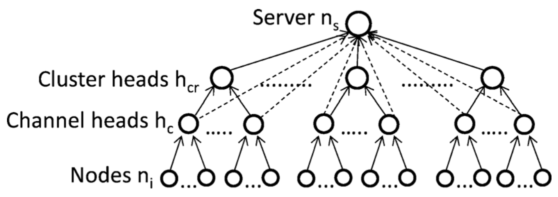


Fig. 3. Hierarchical tree structure in SAVE.

III. Design of the SAVE System

A. Overview of SAVE

Fig. 2 shows a high-level view of the structure of SAVE. The server node (denoted by n_s) is the center of the entire network. Initially, all nodes in each channel form an overlay. In SAVE, each channel overlay has a channel head denoted by h_c, which is a stable node with the highest capacity and longest lifetime staying in the channel. SAVE has two main schemes: channel clustering and friendlist.

Channel Clustering Scheme (Section III-B): This scheme considers the interactions between channels and connects the frequently interacted channels (i.e., connects a group of nodes with similar successive- and multichannel watching activities). As time goes on, the information of watching activities of the nodes is collected, and the single channels are gradually grouped into channel clusters. Channel overlays in one *channel cluster*, denoted by cr, are merged into one overlay or are bridged. By bridged, we mean the head (h_c) of each channel overlay is connected with the heads of other channel overlays in the cluster. As shown in Fig. 3, each cluster also has a cluster head, denoted by h_{cr}, connecting with all h_c in its cluster. The nodes in each channel cluster can, with great probability, quickly switch between or simultaneously watch their favorite channels without the involvement of the central server.

Friendlist Scheme (Section III-C). The friendlist scheme enables a node to maintain a friendlist that records nodes with similar individual channel watching patterns (i.e., interest channels and watching time). By relying on the friendlist, a node can quickly join in a channel that is not in its current cluster. Also, when a returning node (non-first-time user) starts to watch a channel, it can rely on its friendlist rather than the server to join its desired overlay.

As a result, when a node connects to a new channel, it first attempts to take advantage of the channel cluster. The node can directly request chunks in its current overlay if the overlay owns the channel. Otherwise, the node tries to take the bridge connecting to the new channel. If it fails, it uses its friendlist. If it

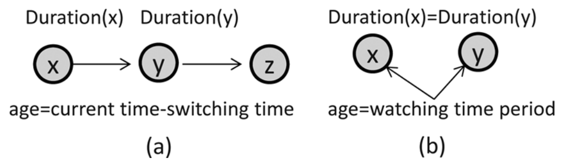


Fig. 4. Illustration of the parameter definitions. (a) Channel switching. (b) Multi!channel watching.

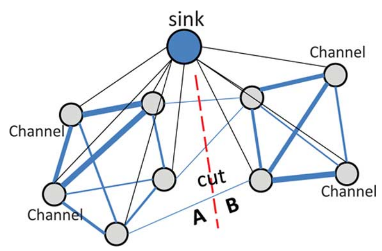


Fig. 5. Minimum-cut tree-based clustering algorithm.

cannot find a friend in its desired channel within certain hops, it resorts to the server finally.

B. Channel Clustering

We use *channel closeness* of two channels to reflect the frequency of interactions between these channels, i.e., the recent tendency of nodes to switch between or watch both channels. Such a tendency can be evaluated by three factors: 1) the age (i.e., freshness) of the node's switching or multichannel watching activity on both channels; 2) the time period that the node stays in both channels; and 3) if both the channels are in the node's interested channel list. We introduce how to consider these factors to calculate the channel closeness.

As shown in Fig. 4(a), for a node's switching activity from channel x to channel y, we define *age* as the time elapsed since the switching. We define $t(x)$ as the time interval that the node stays in channel x before switching. Similarly, as shown in Fig. 4(b), for a node's multichannel watching activity on both channels, we define *age* as the time elapsed since the multichannel watching is started, and define $t(x)$ and $t(y)$ as the time interval that the node stays in both channels; note $t(x) = t(y)$. When $t(x)$ reaches a certain value, we consider the node is truly interested in watching channel x. Accordingly, we predefine a threshold T_s, and define parameter $I(x) = 1$ when $t(x) \geq T_s$; otherwise $I(x) = 0.1$. In SAVE, each node has a profile that lists its interested channels specified by the user. If both channels x and y are in the node's interested channels, we consider the switching nonaccidental, and set the value of parameter γ to 1. Otherwise, we set $\gamma = 0.1$ to minimize the influence of accidental switching activities. The scale for parameter I and γ is adjustable. They reflect, to a certain extent, whether both x and y are the node's interested channels, and the switching or multichannel watching is the node's routine activity.

The channel head of channel y keeps a record of channel watching and switching activities of nodes in channel y (denoted by Ω) and calculates the *channel closeness* between channels x and y by

$$C(x,y) = \sum_{\Omega} \frac{I(x) \cdot I(y)}{\varpi^{\text{age}}} \cdot \gamma \tag{1}$$

where $\varpi > 1$ is a scaling parameter, which exponentially reduces the freshness of switching and multichannel watching activities. Thus, the value of C(x,y) is in the range of (0, 1].

The closeness of two channels can be regarded as the weight of a link connecting them in the social network graph. The channel clustering is the process of grouping channels with high-weight links. SAVE aims to generate clusters so that the number of intracluster interactions is maximized and the number of intercluster interactions is minimized. To this end, we first propose a centralized method using the server to collect global interchannel activities for channel clustering (in Section III-B.1). Then, we further develop a decentralized method to cluster channels by utilizing the local interchannel activity information (in Section III-B.2).

1) Centralized Channel Clustering: In the centralized clustering method, nodes report their activities to their channel heads, which calculate channel closeness and report the information to the server. Then, the server conducts channel clustering. The server first generates an undirected graph $G(V,E)$, where vertices (V) represent channels and edges (E) represent the interactions between channels. The weight of the edge connecting channels x and y is the sum of channel closeness $C(x,y)$ and $C(y,x)$. The server then uses the minimum-cut tree-based algorithm to divide the vertices in the entire graph to subsets, i.e., to create channel clusters. As shown in Fig. 5, a *cut* separates all channels V in graph G into two channel subsets A and B. The value of a cut equals the sum of the weights of the edges crossing the cut. The minimum-cut tree algorithm creates clusters that have small sum of intercluster cut values and relatively large sum of intracluster cut values. Algorithm 1 shows the pseudocode of the clustering algorithm corresponding to the process in Fig. 5. First, we insert an artificial sink s into the graph $G(V,E)$ (step 1). The sink is connected with all channels in the graph with weight w $(0 \leq w \leq 2)$ (steps 2–4). α is used to control the number of generated clusters. If $w = 0$, all channels will be in one giant cluster, while $w = 2$ will make all channels become singletons. Then, we use the maximum flow algorithm [47] that involves a recursive process to construct a minimum-cut tree with the minimum sum of the values of cuts (step 5). Next, we remove the sink, and graph G consequently is divided into several clusters (steps 6 and 7). The intracluster cut value can be used to measure the tightness of channels in each cluster. If a cluster has tightness higher than a predefined threshold, its channels are merged to one overlay. Otherwise, its channels build bridges between each other. The minimum cut algorithm requires a computation complexity of $O(|V| \cdot |E| \log(|V|^2/|E|))$ [48]. Other clustering approaches (e.g., [49]) can also be adopted for the channel clustering in SAVE.

2) Decentralized Channel Clustering: For ease of presentation, we use *channel cluster* to denote both individual channels and a cluster of multiple channels. A cluster head is the most stable node with the highest capacity and longest lifetime staying in the cluster. The decentralized method aims to generate and maintain a stable state for the created clusters, i.e., they have small sum of intercluster closeness and relatively large

Algorithm 1: Centralized channel clustering algorithm executed by n_s

```
1 $V' = V \cup s$; //$s$ is the sink;
2 //Connect $s$ to $V$ to generate an expanded graph $G'(V', E')$;
3 for all nodes $v \in V$ do
4     Connect $v$ to $s$ with an edge of weight $w$;
5 Calculate the minimum cut tree $T'$ of $G'$;
6 Remove $s$ from $T'$;
7 Divide $G$ to clusters with small sum of intercluster cut values and large sum of intracluster cut values;
8 Return all connected subgraphs as the clusters of $G$;
```

sum of intracluster channel closeness. For this purpose, we introduce a concept: *stability* of cluster cr_i to cluster cr_j, denoted by $V(cr_i, cr_j)$

$$V(cr_i, cr_j) = \left\{ \sum_{x,y \in cr_i} C(x,y) - \sum_{x \in cr_i, y' \in cr_j} C(x,y') \right\} \Big/ |cr_i| \tag{2}$$

where $|cr_i|$ is the number of channels in cr_i. A higher $V(cr_i, cr_j)$ means more channel interactions within cr_i than between cr_i and cr_j. We use Θ and $|\Theta|$ to denote the set and the number of all channel clusters in the system, respetively. Cluster head h_{cr_i} builds a *cluster stability vector*, denoted by

$$\mathcal{V}_i = [V(cr_i, cr_1), \ldots, V(cr_i, cr_{i-1}), V(cr_i, cr_{i+1}), \ldots, V(cr_i, cr_{|\Theta|})].$$

We use the number of users for cluster cr_j (denoted by p_j) to represent the cluster's popularity. If the channels in cr_j are more popular, nodes in cr_i have a higher tendency to watch the channels in cr_j. Thus, we define *inverse popularity vector* of $cr_i(\mathcal{P}_i)$, which includes $1/p_j$ of each channel cluster in $(\Theta - cr_i)$, and define cr_i's *cluster stability degree* as

$$S_{cr_i} = \mathcal{P}_i \cdot \mathcal{V}_i^{\top}. \tag{3}$$

Channel clusters with larger S_{cr} values are more stable, i.e., more interactions occur within the cluster than outside the cluster.

We use $cr_{(i,j)}$ to denote the cluster that groups cr_i and cr_j (i.e., $cr_{(i,j)} = cr_i \cup cr_j$). To create stable clusters, for each interacted cluster cr_j, cluster head h_{cr_i} calculates $S_{cr_{(i,j)}}$ based on (3). If $S_{cr_i} < S_{cr_{(i,k)}}$, that is, the stability degree of the grouped cluster is higher, cr_i intends to group with cr_j.

Algorithm 2 shows the decentralized channel clustering algorithm. It is conducted by each channel cluster head h_{cr_i} to identify channel clusters in Θ that it can group with to maximize its S_{cr}. Each cluster head h_{cr_i} periodically collects the information of intracluster and intercluster interactions between its own cluster (cr_i) and other clusters, and calculates $\mathcal{V}_i$ and S_{cr_i} (step 1). For each interacted cluster cr_k (step 2), h_{cr_i} calculates the grouped cluster $cr_{(i,k)} = cr_i \cup cr_k$ (step 3), asks for information from h_{cr_k} (step 3), and then calculates $S_{cr_{(i,k)}}$ (steps 5–8). If $cr_{(i,k)}$ is more stable than cr_i (i.e., $S_{cr_i} < S_{cr_{(i,k)}}$), h_{cr_i} sends an invitation to cr_k. cr_k then decides whether adding cr_i will cause an increase of its S_{cr}. If yes, it accepts the invitation.

Algorithm 2: Decentralized channel clustering procedure executed by cluster head h_{cr_i}

```
1  Calculate $\mathcal{V}_i$ and $S_{cr_i}$ [(2) and (3)];
2  for each interacted channel cluster $cr_k \in (\Theta - cr_i)$ do
3      $cr_{(i,k)} = cr_i \cup cr_k$;
4      Ask for information from $h_{cr_k}$;
5      for each channel cluster $cr_a \in (\Theta - cr_i - cr_k)$ do
6          Calculate $V(cr_{(i,k)}, cr_a)$ in $\mathcal{V}_{(i,k)}$ [(2)];
7      Calculate $\mathcal{P}_{(i,k)}$;
8      Calculate $S_{cr_{(i,k)}} = \mathcal{P}_{(i,k)} \cdot \mathcal{V}_{(i,k)}{}^{\top}$ [(3)]
9      if $S_{cr_i} < S_{cr_{(i,k)}}$ then
10         //$cr_{(i,k)}$ is more stable than $cr_i$;
11         Return $cr_k$; //return the selected $cr_k$
```

If $S_{cr_{(i,k)}}$ is greater than a predefined threshold, two clusters merge to one overlay. Otherwise, they build bridges between each other. The interactions between channels in one cluster may diminish over time.

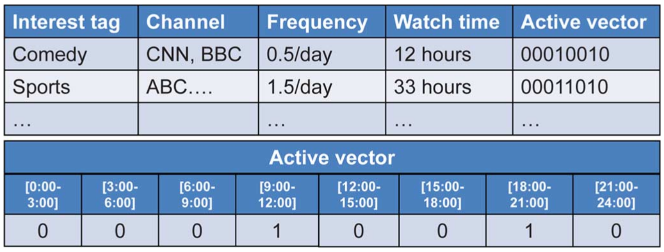
Each cluster head periodically uses the minimum-cut tree-based clustering algorithm to partition its cluster. If it finds that the partition leads to a higher S_{cr} for a new partitioned cluster, this partitioned cluster is split from the cluster.

Suppose $|\Omega|$ is the maximum number of activities in Ω, n is the maximum number of channels in a cluster, and m is the maximum number of channel clusters. The complexity of the calculation in steps 1, 6, and 8 in Algorithm 2 is $O(dn^2|\Omega|)$, and the loops in steps 2 and 5 have m iterations. Therefore, the algorithm has a complexity of $O(dm^2n^2|\Omega|)$.

3) Cluster Combination and Partition: We use "cluster combination and partition" to represent both "bridge construction and removal" and "overlay combination and separation." As shown in Fig. 3, SAVE has a hierarchical tree structure composed of the nodes, channel heads, cluster heads (only in the decentralized clustering method), and the server in the bottom–up direction. In a channel overlay, each node has a connection with the channel head. In a cluster, each channel head has a connection (i.e., bridge) with the cluster head. Also, the server has connections with all channel heads and the cluster head in each cluster.

In the centralized clustering method, the server notifies the channel heads of frequently interacted channels to build or remove bridges between them. In the decentralized clustering method, the cluster heads communicate with each other for the bridge construction and elimination. In cluster combining, to build a bridge between two channel clusters (including individual channels), each channel head in one cluster builds connection with each channel head in the other cluster. The more stable and higher-capacity cluster head becomes the cluster head of the new bridged cluster. In both centralized and decentralized methods, after the bridge is built, the channel heads notify the nodes in their channels about the bridge establishment. Each channel head and nodes in a cluster maintain a record of all channels in its cluster.

When a cluster head finds that splitting a cluster (or a channel) from its cluster can increase its cluster's S_{cr}, it notifies the head of this cluster (or channel). The notified head removes the bridges to other channels and becomes the cluster head of its cluster. The cluster head also notifies all other channel heads



Interest tag	Channel	Frequency	Watch time	Active vector
Comedy	CNN, BBC	0.5/day	12 hours	00010010
Sports	ABC….	1.5/day	33 hours	00011010
…		…	…	…

Active vector							
[0:00-3:00]	[3:00-6:00]	[6:00-9:00]	[9:00-12:00]	[12:00-15:00]	[15:00-18:00]	[18:00-21:00]	[21:00-24:00]
0	0	0	1	0	0	1	0

Fig. 6. Profile of a node.

TABLE I
RECORD FOR A FRIEND IN THE FRIENDLIST

IP address	Similarity	Profile	Entry creation time

in the cluster, which remove the corresponding bridges. After a channel head removes the bridges, it notifies its nodes in the channel, which update their records of bridged channels.

C. Friendlist Construction

Today's live streaming applications usually list a number of interest tags for channels. SAVE requests users to fill their interest tags manually when they initially join in the system and to periodically update their tags. Fig. 6 shows a node's profile based on its own channel watching activities in SAVE. "Interest tag" is a channel category such as comedy, sports, and news that a node likes to watch. "Channel" lists the channels that the node frequently watches in an interest tag. "Frequency" and "Watch time" stand for the frequency and time of watching the channels in an interest tag during a certain period. "Active vector" represents the daily watching routine of a node. By dividing the 24 h of a day to T time-slots, we can use a binary string to represent the activity of a node during a day. For example, 00010010 means that the user usually watches video from 9:00 am to 12:00 pm and 6:00–9:00 pm. The time is unified into a standard time zone. A fine-grained time partition can be used to improve the accuracy.

To determine the *similarity* of two nodes, n_i and n_j, we consider their common interests, common frequently watched channels, and the overlap of their watching time periods. Accordingly

$$S(n_i,n_j)=\sum_{tag_i\cap tag_j} S_{cl}(n_i,n_j)\cdot S_{av}(n_i,n_j) \tag{4}$$

where $tag_i \cap tag_j$ includes the common tags between n_i and n_j; $S_{cl}(n_i,n_j)=\frac{|cl_i\cap cl_j|}{|cl_i\cup cl_j|}$ is the similarity between their channel lists cl_i and cl_j; $S_{\text{av}}(n_i,n_j)=\frac{|v_i\cap v_j|}{T}$ is the similarity of their active vectors v_i and v_j. The *similarity* of two nodes represents the probability that they watch the same channel at the same time. Unlike other social-network-based methods [7] that only consider common interests, SAVE also considers watching time for friend clustering, which leverages user routine behavior for efficient video sharing.

Each node in SAVE maintains a friendlist that records a certain number of friends sharing high similarity with itself. The record of a friend consists of the items listed in Table I. When node n_i communicates with the heads or the server during video watching, it piggybacks its profile on the messages. The heads or server then send back a list of friends sharing high similarity with n_i for friendlist creation and update. Note that there may be overlaps between a node's neighbors in its channel (or cluster) overlays and the recommended friends. To avoid sending unnecessary requests to friends that cannot help with the channel switching, a node excludes its overlay neighbors from the recommended friend list when creating or updating its friendlist. To keep the friendlist updated, node n_i periodically updates its friendlist by calculating $\frac{S(n_i,n_j)}{Age(n_j)}$, where $Age(n_j)$ is the time interval between the creation/update time of the friend n_j in the friendlist and current time. The profiles with the similarity values less than a predefined threshold will be discarded.

On weekends and holidays, users' video watching time should be different from weekdays. We can add the "active vector" for weekends and for holidays in each node's profile to increase the accuracy of user watching time pattern, and consider these active vectors when determining the watching pattern similarity of two nodes. Then, when searching for video chunk providers, we also consider weekends and holidays. We will conduct this nontrivial task of considering weekends and holidays to enhance the accuracy in our future work.

D. Efficient Multichannel Video Streaming

When node n_i initially joins in SAVE, it requests n_s to recommend nodes in its desired channel. Node n_i then joins in the channel overlay by connecting to the recommended nodes for chunk sharing. In a cluster, channels could be merged into one overlay or bridged. Because the channels in a merged overlay are very close to each other (i.e., nodes frequently conduct successive- or multichannel watching activities on these channels), when a node in a merged overlay wants to switch channel or watch multichannels, it has a high probability to find its requested chunks from its overlay using the original chunk search algorithm [15], [20]–[37]. Because the bridged channels are relatively close to each other, a node is very likely to find the bridges to join in the overlay of its desired channel. If the node fails to find such a bridge, it then uses its friendlist, and finally resorts to the server.

We explain the details of the channel switching or joining. To use bridges to switch to channel c_j, node n_i in channel c_i directly sends a request with *chunkName* to its channel head h_{c_i}, which checks whether it connects to h_{c_j}. If yes, it forwards the request to h_{c_j}. Then, h_{c_j} responds to n_i with a few nodes in its channel overlay and also forwards the chunk request to its neighbors. The chunk owner is searched by using the original chunk search algorithm in the P2P live streaming systems [15], [20]–[37]. The chunk owner sends the requested chunk to n_i. n_i connects to the returned nodes, and hence has joined in c_j. If h_{c_i} is not bridged with h_{c_j}, then n_i tries to use its friendlist to find a bootstrap node to join in c_j. n_i sends a request with time to live (TTL) to all of its friends. The TTL denotes the number of hops a request will be forwarded. After receiving the request, a node checks whether it is in the overlay of c_j. If not, it decreases the TTL by one and further forwards the request to its friends. Otherwise, it responds to the requester with a few nodes in c_j, and also finds a chunk provider, which returns the requested chunk to n_i. Then, n_i connects to the returned nodes and has joined in c_j. The node with $\text{TTL}=0$ notifies n_i that the search has failed. Then, n_i resorts to the server to join in c_j.

E. Capacity-Based Chunk Provider Selection

A node's capacity represents the number of chunk requests it can concurrently serve. When there are several potential chunk providers in the network, selecting a high-capacity node improves user watching experience. Section III-D explained how node n_i requests the chunks of its desired channel when it wants to switch channel or watch multichannels. When n_i in channel c_i wants to switch to channel c_j, if c_j and c_i are bridged, the channel head of c_j, h_{c_j} will recommend a few nodes in its channel overlay to n_i; otherwise, if node n_j within TTL hops of n_i's friend network is in the overlay of c_j, n_j recommends a few nodes in the overlay of c_j. Before h_{c_j} or n_j recommends nodes to n_i, they can ask several candidates about their available capacities, and then pick the ones with the highest available capacity to recommend. To avoid the latency for the available capacity querying, the nodes in each channel overlay can periodically report their available capacities to their channel head, which further broadcasts this information to all nodes in the channel overlay. Thus, a channel head or a friend can directly recommend the nodes with the highest available capacity. By connecting to high-capacity nodes as chunk providers, the chunk requester can have a better watching experience.

F. Structure Maintenance in Node Churn

Node churn is mainly about the node joins and departures from the system, namely the nodes are getting offline or online. SAVE needs to maintain its structure in node churn. To ensure there is always a head node in each channel, before a channel head departs, it selects a new head node and transfers all of its information to the new head. Also, it notifies all related nodes including all nodes in its channel and the channel heads of other channels in its cluster about the new channel head. It also notifies the server in the centralized method and notifies the server and its cluster head in the decentralized method. The notified nodes update their connections accordingly. The joins and departures of normal nodes are handled by the original protocol in the P2P live streaming system [15], [20]–[37]. A head may fail without warning. If node n_i has not received a response from its connected node n_j after a certain period of time, it assumes that n_j is dead or has left the system without warning. If n_j is a head, n_i notifies the server, which selects the node with the highest stability and capacity and longest lifetime from the nodes in the channel and notifies all nodes that originally connected to the old head. The departures and failures of cluster heads are handled in the same way as the channel heads. SAVE can use multiple channel heads and cluster heads to enhance system reliability, but at the cost of higher maintenance overhead.

G. Channel-Closeness-Based Chunk-Pushing

How to fully utilize the limited cache of each node to reduce channel viewing startup delay is a challenge. Xu *et al.* [50] showed that when the cache used for a channel reaches 660 kB, the cache hit rate nearly reaches 100%. The chunk unit size is the fixed data piece size (1 kB) in data transmission in P2P live streaming [50], [51]. Then, the number of cached chunks needed for one channel being watched is 660. Thus, reserving $660m$ kB (i.e., $660m$-chunk) cache for the m channels that a node is simultaneously watching is sufficient for smooth watching activity. The remaining cache can be used for prefetching chunks of channels the node is very likely to watch.

Recall that the channel closeness of two channels indicates the probability that nodes in one channel will switch to the other or nodes simultaneously watch them. If channels c_i and c_j are close channels (in a merged overlay or two bridged overlays), nodes in channel c_i tend to watch channel c_j. Then, if the chunks of c_j can be cached in the nodes in c_i, when c_i viewers want to watch c_j, they can immediately watch c_j through these cached chunks before connecting to chunk providers in c_j. Based on this rationale, we propose a channel-closeness-based chunk-pushing strategy, in which the chunks of the close channels of each channel c_i are pushed to the caches of the nodes in c_i to maximally reduce the startup delay of channel watching. As a channel being watched needs 660-chunk cache size, $660/k$-chunk cache can be used for a channel not being watched. The value of k is determined so that the time for viewing $660/k$ chunks can cover the time for finding chunk providers to which to connect. Fig. 7 depicts the cache allocation of different channels in a node, and only the extra cache space is used for pushed chunks.

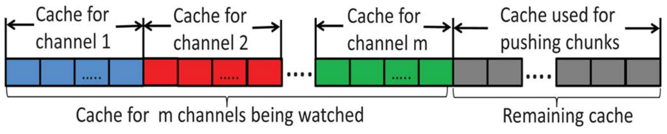


Fig. 7. Cache allocation for different channels in a node.

Recall that the server keeps track of the switching and multichannel watching activities in the entire system and calculates the closeness between each pair of channels periodically. Based on this information, the server then identifies a list of q top closest channels for each channel c_i represented by $\vec{L}_i = \langle c_1, c_2, \ldots c_q \rangle$, which is ordered in the descending order of their closeness with channel c_i. The server then pushes $660/k$ chunks of each channel in $\vec{L}_i$ to randomly selected nodes in the overlay of c_i so that each chunk exists in K caches. This process is completed in a K-round iteration. In each round, the server randomly selects a node from the overlay of c_i with extra cache, and pushes the chunks of each c_j $(j = 1, 2 \ldots q)$ to it until the node has no extra cache or the chunks of c_q are pushed. In the former case, the server randomly selects another node and continues to push the remaining chunks. These cached chunks will then be further pushed to nodes throughout the overlay of c_i. Specifically, overlay neighboring nodes piggyback the information of their extra cache sizes and cached channels in $\vec{L}_i$ on their periodical exchanged messages for overlay maintenance. If node n_j notices that its neighbor node n_i has extra cache and does not have its cached chunks, n_j pushes these chunks to n_i. Since chunks from higher-closeness channels to channel c_i have a higher probability to be requested by the nodes in c_i, a node gives a higher priority to chunks of higher-closeness channels to be stored in its cache. Whenever a node needs to cache new channel chunks and its cache is full, it selects the oldest chunks of the least-close channel to replace.

With our proposed chunk-pushing strategy, when node n_i in c_i wants to watch channel c_j in $\vec{L}_i$, if n_i has cached chunks for c_j, it can immediately start watching c_j before it finds and connects to chunk providers of c_j. Otherwise, node n_i searches cached chunks in its nearby nodes in the overlay of c_i when it searches the chunk providers as introduced in Section 3.4. That is, node n_i first broadcasts a chunk request containing the *chunkName* with TTL to nodes within its current overlay. If request receiver n_j has pushed chunks of channel c_j, n_j sends

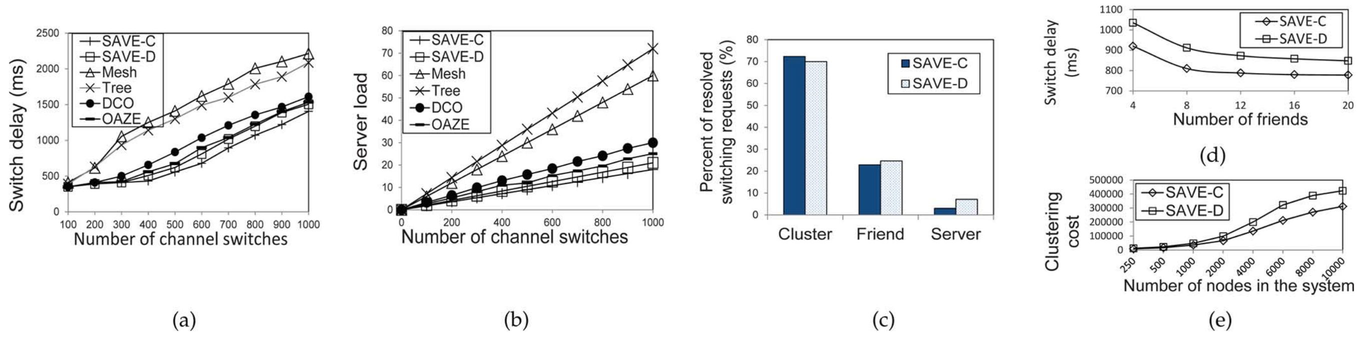


Fig. 8. Experimental results from simulation on PeerSim. (a) Channel switch delay. (b) Server load. (c) Distribution of resolved switching requests. (d) Effectiveness of friendlists. (e) Clustering cost.

TABLE II
EXPERIMENT DEFAULT PARAMETERS

Parameter	Default value
Simulation duration	24 hours
Number of nodes	10,000 (simulation), 300 (PlanetLab)
Number of channels	100 (simulation), 30 (PlanetLab)
Video bit rate	600kbps
Online time for nodes	distribution from Figure 27
Warm-up period	1 hour

these chunks back to n_i. Then, n_i is able to begin watching channel c_j before it finds and connects to the chunk providers of c_j, thus decreasing the watching startup delay. Note that this cached chunk searching step can be integrated with the original chunk search algorithm in node n_i's merged overlay. The chunks of close channels to channel c_i are frequently demanded by nodes in c_i. By pushing these chunks to nodes in c_i, our proposed strategy reduces the watching startup time.

IV. Performance Evaluation

To evaluate the performance of SAVE, we used the event-driven simulator PeerSim [17]. In order to further investigate the performance of SAVE in the real-world environment, we also built SAVE prototypes on the PlanetLab [18] real-world testbed. PlanetLab can provide real delay measurement due to the heterogeneous and time-varying bandwidths and instability of nodes. The data from our survey in Section VI of the supplementary file is used as the foundation to simulate user activities to make the experiment more realistic. Each test lasts for 24 h. In our simulation, the P2P live streaming system consists of 10 000 nodes and 100 channels. We set the default video bit rate, which is the size of a video segment per second, to 600 kb/s [52]. In (1), ϖ was set to 1.02 and T_s was set to 10 min. In the PlanetLab experiment, we selected 300 online nodes and chose the computer with IP address 128.112.139.26 in Princeton University as the server. Considering that the PlanetLab test have much fewer nodes than the simulation, we reduced the number of channels to 30.

The TTL for friend lookup through friendlists was set to 2. The number of interests of each node is distributed in [1], [7] according to the survey results shown in Fig. 24 of the supplementary file. We regarded one interest as an interested channel. We evenly divided the 30 or 100 channels to five groups. Each node chose 90% of its interested channels from one group randomly chosen from the five groups, and chose the remaining 10% of its interested channels randomly from the channels in other groups. When a node switches channels, it has 90% probability to watch a channel in its interested channels, and 10% probability to watch other channels. The channel switching time is distributed according to the survey result in Fig. 22 of the supplementary file (i.e., a certain percent of nodes have a certain channel switching interval time). A node periodically switches channel after its switching interval time has elapsed. The statistics in [53] and [7] are used for the distribution of the download bandwidth in the simulation. A node's upload bandwidth is set to 1/3 of its download bandwidth [54]. The default settings are summarized in Table II.

Since mesh structure is used in most current P2P live streaming systems, we first built SAVE on the mesh structure and compared it to mesh-based system [2] (Mesh), tree-based system [23] (Tree), and DHT-based system (DCO) [20] in order to show the effectiveness of SAVE on improving the efficiency of these systems, where a node needs to contact the central server for switching channels. We also compared SAVE to the OAZE distributed channel switching approach [19]. For OAZE, we used its practical algorithm, in which every peer connects to a channel that is randomly picked among the channel group of the peer's current channel. The node connects to c number of nodes in its connected channel, and c was randomly chosen from 4 and 5. For SAVE, we have two variations using the centralized (Section III-B.1) and decentralized (Section III-B.2) channel clustering methods, represented by "SAVE-C" and "SAVE-D," respectively. We also built SAVE upon DCO for comparison.

A. Switching Delay and Server Load

We measured the switch delay by the time interval between the timestamp that a node sends a request for switching to a new channel and the timestamp that the node receives the first chunk of the new channel. Figs. 8(a) and 9(a) show the switch delays in the simulation and PlanetLab experiments, respectively. We randomly chose 1000 switchings from all switchings. We then ordered the switch delays in an ascending order, calculated the average value of every 100 values, and finally got 10 average values. We see that both SAVE-C and SAVE-D achieve the fastest switching, which confirms their highly efficient video streaming due to the channel clustering and friendlist schemes. In other systems, a node needs to contact the centralized server in order to join in another overlay, thus generating a longer delay. SAVE-D has a slightly larger startup delay than SAVE-C. This is because SAVE-C has the global information of all channel switching activities of all nodes in the system

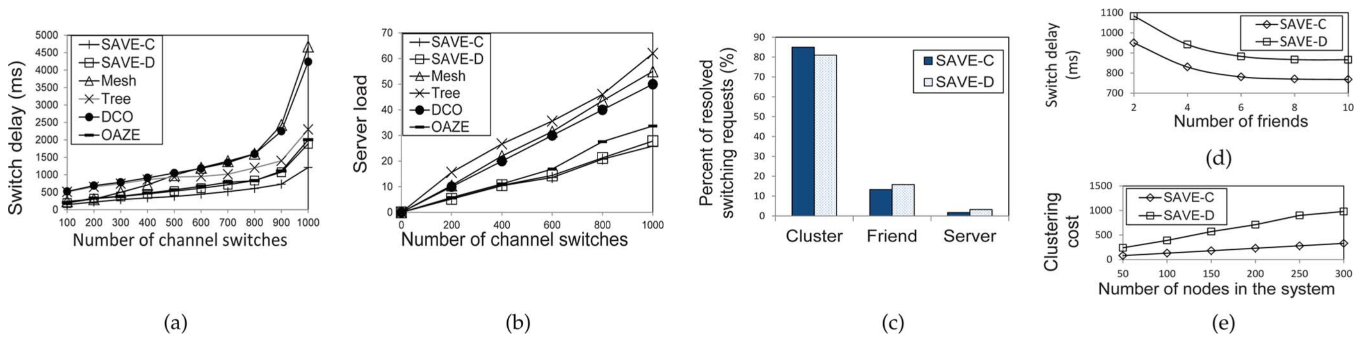


Fig. 9. Experimental results from the PlanetLab testbed. (a) Channel switch delay. (b) Server load. (c) Distribution of resolved switching requests. (d) Effectiveness of friendlists. (e) Clustering cost.

for more accurate clustering channels, while SAVE-D relies on local cluster information exchanges. We also see that Tree produces slightly shorter delay than Mesh because a node in Mesh needs to pull a chunk from its neighbors, while Tree uses push. DCO generates lower delay than Mesh and Tree in simulation and generates lower delay than Mesh but higher delay than Tree in the PlanetLab results. DCO uses stable nodes as DHT nodes for locating chunk owners. In the simulation with a stable environment, nodes in DCO can always find chunk owners relying on DHT, leading to lower delay. However, DHT nodes may fail in a less stable environment in the PlanetLab testbed, which leads to chunk owner location failures and longer delay. OAZE yields a lower switch delay than DCO. This is because each node in OAZE functions as a channel switcher, so a channel switching node may find a switcher of the corresponding channel in its neighbors to join in the channel overly. OAZE yields higher delay than SAVE-D and SAVE-C. As SAVE clusters channels with frequent interactions and a node's friends have the same interests and watching time as the node, a node has a high probability to find nodes in the target channel. We also observe that the delay on PlanetLab exhibits an exponential growth, which is different from the simulation results. This is because in the simulation test, the bandwidths of nodes are constant, while in PlanetLab, nodes' real bandwidth varies and some nodes have very low bandwidth, resulting in long delay in communication.

We define *server load* as the total number of channel joining requests served by the server in order to measure the effectiveness of SAVE in achieving its objective. Figs. 8(b) and 9(b) show the server loads over time in the simulation and PlanetLab experiments, respectively. We observe that SAVE incurs significantly lower server load than other systems. SAVE releases the load on the server by clustering channels with frequent interactions and building friendlists. SAVE-C generates slightly lower server load than SAVE-D due to its more accurate clustering of channels. We see that Mesh generates slightly lower server load than Tree because nodes in Tree need to contact the server more frequently since chunks often fail to transmit due to the vulnerability of the tree structure to churn. DCO produces lower server load than Mesh and Tree due to two reasons. First, nodes can easily find chunk owners relying on DHT. Second, nodes can subsequently receive chunks from the located chunk owners. We see that OAZE produces lower server load than DCO because all nodes in OAZE function as channel switchers and help the server resolve the switching requests. OAZE produces higher server load than SAVE because the probability for a node to find a switcher for the corresponding channel within its neighbors is not high in OAZE, while SAVE can provide a high probability for a node to find nodes in a corresponding channel.

B. *Effectiveness of the Social Network in Save*

Figs. 8(c) and 9(c) show the percent of resolved switch requests using clusters, friends, and the server in SAVE-C and SAVE-D in the simulation and PlanetLab experiments, respectively. In both systems, a significantly higher percentage of switch requests is resolved by clusters, a moderate percentage of requests is resolved by friends, and a very small percentage of requests is resolved by the server. The results demonstrate the effectiveness of the channel clustering and friendlist schemes. We also see that SAVE-C has a higher percentage in using clusters than SAVE-D because SAVE-C has higher accuracy in clustering channels than SAVE-D with global information. Figs. 8(d) and 9(d) show the effect of the number of friends in a friendlist on the channel switch delay. We see that as the number of friends increases, switch delay decreases on PeerSim and PlanetLab. When a node has more friends, it has a higher probability to find a friend watching the channel it wants to switch to. We also see that when the number of friends increases further, the reduction in switch delay becomes smaller. Thus, a relatively small number of friends is sufficient to make the friendlist scheme effective.

Friendlist is an indispensable component in SAVE. As shown in Figs. 8(c) and 9(c), a moderate percentage of channel switch requests is resolved by friends. We tested the impact of friendlist on the channel switch delay and server load. We used "Cluster" to denote SAVE with only channel cluster strategy, "Friend" to denote SAVE with only friendlist strategy, and "Combine" to denote SAVE with both strategies. In "Cluster," when a node's channel switching request cannot be resolved by the channel cluster, the node will contact the server directly; meanwhile in "Friend", a node will first contact its friends when switching to a new channel, and it requests the server if the friends cannot help it join the corresponding channel overlay. Figs. 10(a) and 11(a) show the switch delay for different approaches on PeerSim and PlanetLab, respectively. We see that "Cluster" yields the least switch delay due to the effectiveness of channel clustering, and nodes have a high probability to join a new channel overlay with the help of its channel head. "Friend" yields larger delay than "Cluster" due to the limited number of friends, and around 30% percent of switch requests are solved by friends. "Combine" generates the largest switch delay, as a node will try "Cluster" and "Friend" before resorting to the server. Figs. 10(b) and 11(b) show the sever load status for

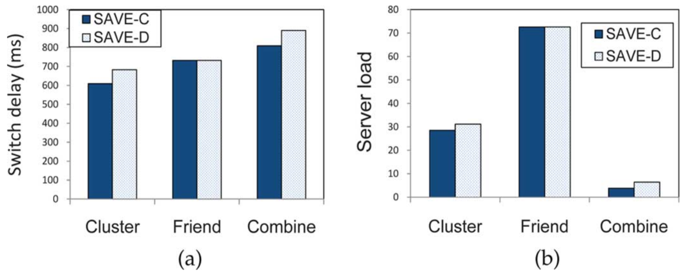


Fig. 10. Impact of friendlist on PeerSim. (a) Channel switch delay. (b) Server load.

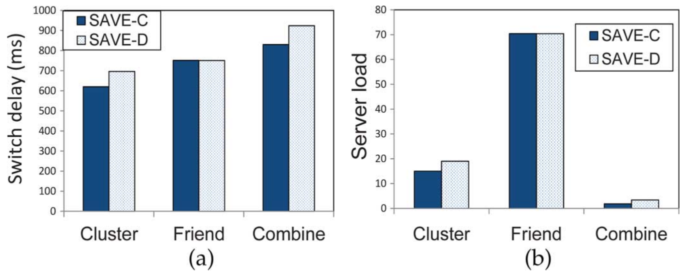


Fig. 11. Impact of friendlist on PlanetLab. (a) Channel switch delay. (b) Server load.

different approaches on PeerSim and PlanetLab, respectively. We see that by combining "Cluster" and "Friend" strategies, the sever load will be dramatically reduced. Thus, "Friend" is effective in reducing server load in SAVE.

The experimental results show that SAVE is effective in clustering channels. In PeerSim, on average, SAVE is able to group all 100 channels into nine channel clusters. Within each cluster, one channel overlay merging occurs, and other channel overlays are bridged. On PlanetLab, all channels are grouped into five clusters. Within each cluster, an average of 0.8 overlay merging occurs, and all other channel overlays are bridged.

C. Cost of Save

Figs. 8(e) and 9(e) show the clustering cost of SAVE-C and SAVE-D measured by the total number of messages to cluster channels in the simulation and PlanetLab experiments, respectively. The clustering cost increases as the number of nodes in the system increases. Also, SAVE-D requires more communications than SAVE-C in building the clusters. This is because in SAVE-C, channel heads report to the server about node channel switching activities. In SAVE-D, channel heads report to their cluster heads about node channel switchings, and all cluster heads need to communicate with each other for cluster combining and splitting. However, the communication messages are distributed among many cluster head nodes, which does not increase server load. We can see that SAVE's clustering cost is acceptable.

We use Y to denote the total number of downloaded chunks in the system, and use X to denote the total number of communication messages including those that handle node joins and departures. Then, we measured the maintenance overhead by $X/(X+Y)$. The lifetime of each node is chosen from $[x-0.2x, x+0.2x]$ min, where the average lifetime x was varied from 10 to 30 with an increment of 5 in each step. Fig. 12(a) and (b) shows the maintenance overhead as a function of node churn rate on PeerSim and PlanetLab, respectively. As the average lifetime increases, the maintenance overhead decreases.

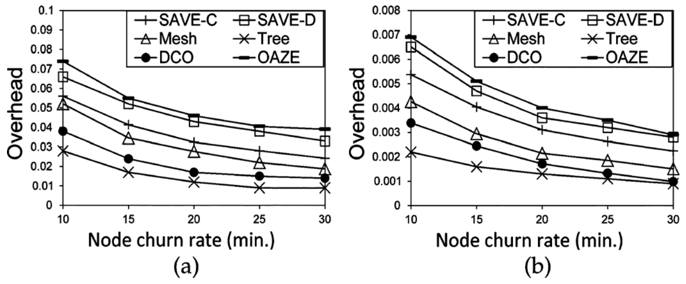


Fig. 12. Overhead vs. node churn rate. (a) The PeerSim simulator. (b) The PlanetLab testbed.

This is because a slower node join and departure rate generates fewer messages for joining and leaving the overlay structure. OAZE and SAVE are built on Mesh, and they have additional methods to avoid relying on the central server for channel switching, which produce additional overhead, so they produce higher overhead than Mesh.

OAZE generates a higher overhead than SAVE-C and SAVE-D. This is because a node probes neighbors within two hops to switch to a new channel in OAZE. In SAVE-C and SAVE-D, due to the effectiveness of both the channel clustering scheme and friendlist scheme, a node can join a new channel overlay with fewer node communications, leading to less overhead than OAZE. Also, SAVE-D generates a higher overhead than SAVE-C since cluster heads need to communicate with each other in SAVE-D. We can see that SAVE's overheads are acceptable. DCO produces a lower overhead than Mesh because nodes in DCO do not need to constantly contact each other for chunk information. Tree generates the least overhead because each node maintains fewer neighbors than other methods; each node has one parent and two children. However, Tree is vulnerable to churn as mentioned before.

D. Performance With Flash Crowds and User Churn

In order to test the performance of SAVE under flash crowds, we made a large number of users switch to a specific channel (e.g., a football game or a concert) according to [14]. Specifically, 15% of users switched to this channel every 5 min during the first 30 min, and 8% of users changed to this channel every 5 min during the next 60 min. Other settings are the same as the default settings. Figs. 13(a) and 14(a) show the switch delays, and Figs. 14(b) and 13(b) show the server loads in the simulation and PlanetLab experiments, respectively. We see that the experimental results are consistent with those in Figs. 8(a), 9(a), 8(b), and 9(b) due to the same reasons. These experimental results show that SAVE outperforms other systems with lower switch delay and lower server load even when the system is under flash crowds.

Next, we test the performance of SAVE under different node churn rates. The node churn rate setting is the same as that in Fig. 12(a) and (b). Figs. 13(c) and 14(c) show the switch delays, and Figs. 13(d) and 14(d) show the server loads with different churn rates in simulation and PlanetLab experiments, respectively. We see that the relative performance differences between different systems are consistent with those in the previous experimental results due to the same reasons. Also, we see that the churn rate does not greatly affect the performance of the systems due to their structure maintenance mechanisms. These results verify that SAVE is resilient to node churn.

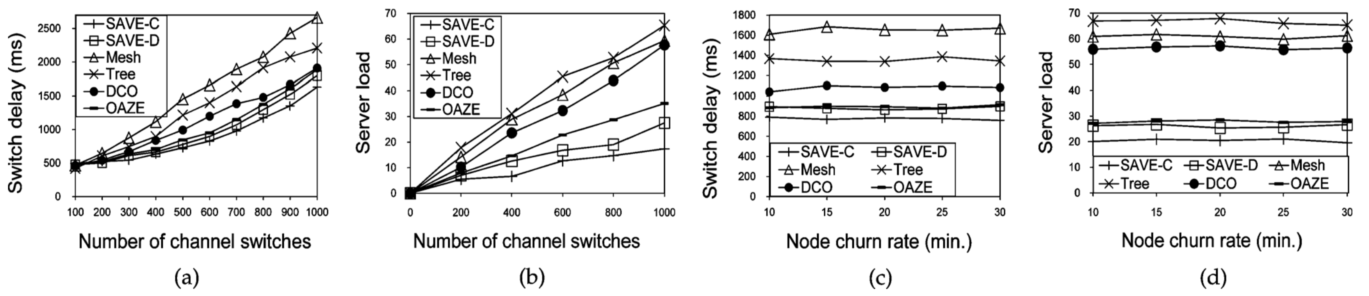


Fig. 13. Experimental results from simulation on PeerSim under flash crowds and churn. (a) Channel switch delay under flash crowds. (b) Server load under flash crowds. (c) Channel switch delay under different user churn rates. (d) Server load under different user churn rates.

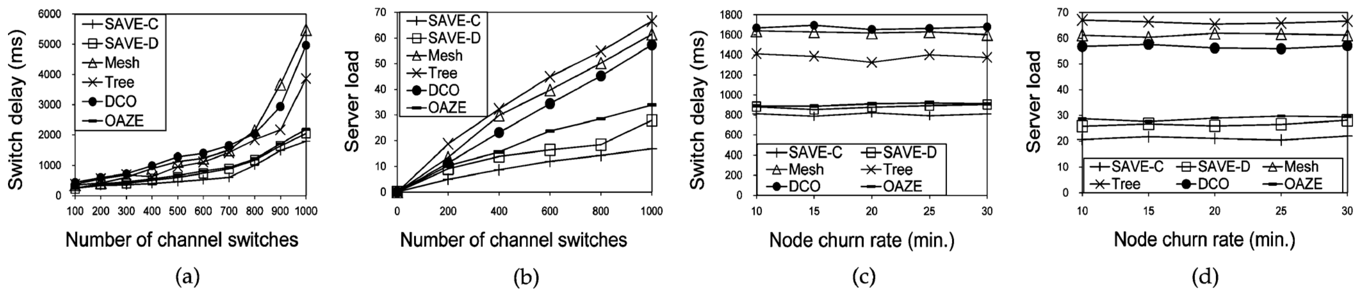


Fig. 14. Experimental results from the PlanetLab testbed under flash crowds and churn. (a) Channel switch delay under flash crowds. (b) Server load under flash crowds. (c) Channel switch delay under different user churn rates. (d) Server load under different user churn rates.

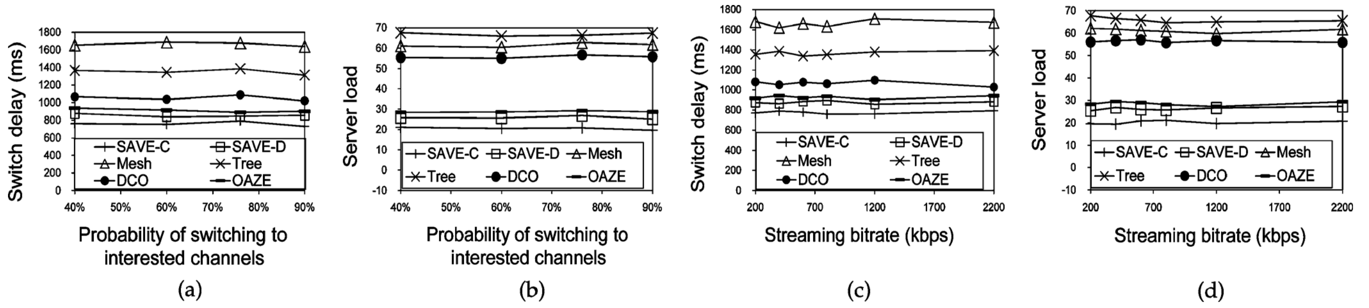


Fig. 15. Experimental results from simulation on PeerSim with different workloads and streaming bit rates. (a) Channel switch delay with different workloads. (b) Server load with different workloads. (c) Channel switch delay with different streaming bit rates. (d) Server load with different streaming bit rates.

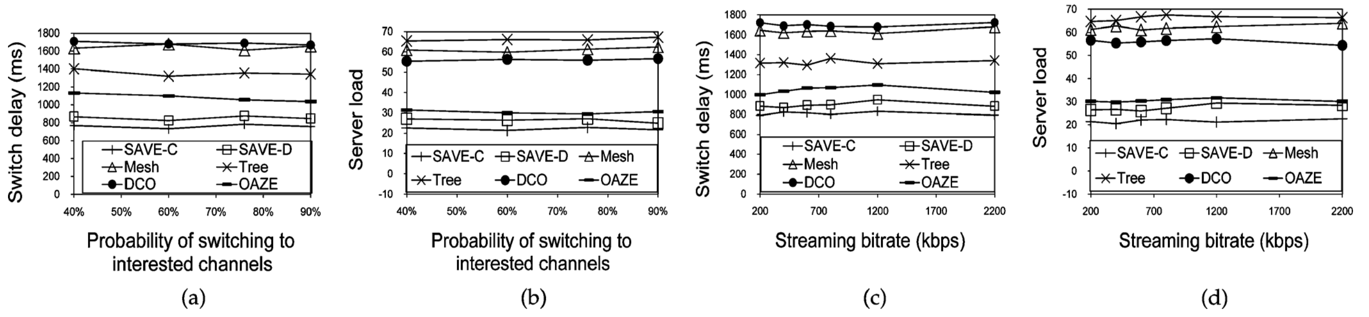


Fig. 16. Experimental results from the PlanetLab testbed with different workloads and streaming bit rates. (a) Channel switch delay with different workloads. (b) Server load with different workloads. (c) Channel switch delay with different streaming bit rates. (d) Server load with different streaming bit rates.

E. Performance With Different Workloads

In this experiment, we set the probability that a node will switch to a channel in its interested channels to 40%, 60%, 76%, and 90% according to [19], [55]. Figs. 15(a) and 16(a) show the switch delays of different systems in the simulation and PlanetLab experiments, respectively. We see that the switch delays remain nearly constant under different workload scenarios, and SAVE outperforms other systems in reducing the switch delay. Figs. 15(b) and 16(b) show the server loads of different systems in the simulation and PlanetLab experiments, respectively. We can see that the experimental results are consistent with those in Figs. 8(b) and 9(b) due to the same reasons. These results indicate that SAVE is effective in reducing the switch delay and server load under different workloads.

F. Performance With Different Streaming Bit Rates

Based on the real streaming bit rates on current video streaming services [56], we varied the streaming bit rate from 200 to 2200 kb/s and tested the performance of different systems. Figs. 15(c) and 16(c) show the switch delays, and

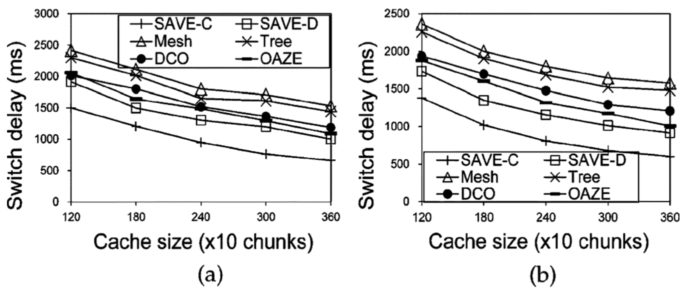


Fig. 17. Effectiveness of the chunk pushing strategies on PeerSim. (a) Random chunk pushing. (b) Channel-closeness-based chunk pushing.

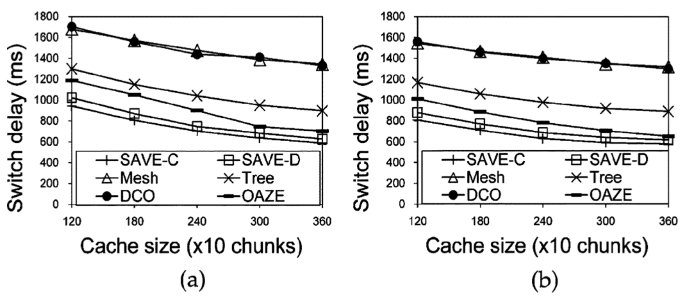


Fig. 18. Effectiveness of the chunk pushing strategies on PlanetLab. (a) Random chunk pushing. (b) Channel-closeness-based chunk pushing.

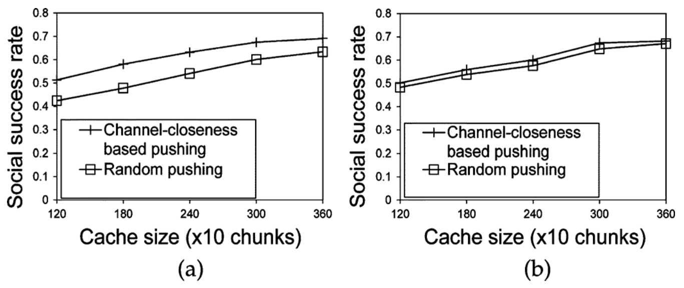


Fig. 19. Social success rate versus cache size. (a) PeerSim simulator. (b) PlanetLab testbed.

Figs. 15(d) and 16(d) show the server loads of different systems in the simulation and PlanetLab experiments, respectively. We see that the switch delays and server loads of different systems remain nearly stable with different streaming bit rates. Also, the relative performance differences between different systems are consistent with those in Figs. 8(c) and 9(a), and Figs. 8(b) and 9(b), due to the same reasons. These experimental results indicate that SAVE outperforms other systems in switch delay and server load with different streaming bit rates.

G. Channel-Closeness-Based Chunk Pushing

In this experiment, we compare our proposed chunk pushing strategy to the random pushing strategy. In the random pushing strategy, the server randomly chooses q channels and pushes the chunks of these channels using the same way as in our proposed pushing strategy; q was set to 10 and 4 in the PeerSim and PlanetLab experiments, respectively. The cache size of nodes in the system follows a Pareto distribution [57] with parameter $\alpha = 2$. We varied the average value of the Pareto distribution in our experiments. The number of cached chunks for each channel was set to 660.

Fig. 17(a) and (b) shows the average channel switch delay versus the average cache size (in the Pareto distribution) for the random pushing strategy and our proposed pushing strategy on PeerSim, respectively. Comparing these two figures, we see that our proposed strategy yields switch delay reduction ranging from 60 to 170 ms for all five methods. The reason for this improvement is that our strategy pushes the chunks of channels that are likely to be viewed into node caches, so that a node has a higher probability to find the cached chunks of its desired channel from itself or nearby nodes. Both figures show that the switch delay decreases as the average cache size increases since a larger cache size allows a node to store the chunks of more channels, hence increasing the probability to find cached chunks for a desired channel. The two figures show that the switch delay follows SAVE-C < SAVE-D < OAZE < DCO < Tree < Mesh due to the same reasons as in Fig. 8(a).

Fig. 18(a) and (b) shows the average channel switch delay versus the average cache size (in the Pareto distribution) for the random pushing strategy and our proposed pushing strategy on PlanetLab, respectively. Comparing these two figures, we see that our strategy produces switch delay reduction ranging from 10 to 160 ms for all five methods. We also see that as the cache size increases, the switch delay decreases in both figures. These results are consistent with the simulation results in Fig. 17(a) and (b) for the same reasons. We also notice that the switch delay follows SAVE-C < SAVE-D < OAZE < Tree < DCO < Mesh, which is consistent with that in Fig. 9(a) due to the same reasons.

Fig. 19(a) and 19 show the social success rate for different average size of cache (in the Pareto distribution) in simulation and on PlanetLab, respectively. We see that our proposed pushing strategy yields improvement in social success rate over the random pushing strategy. Also, when the cache size increases, the social success rate increases because a larger cache can store more channel chunks, thus chunk requests are more likely to be resolved by peers rather than the server.

H. Capacity-Based Chunk Provider Selection

In this experiment, we assumed that the capacity of nodes in the system follows a Pareto distribution [57]–[59] with parameter $\alpha = 2$. When a node receives requests more than its capacity, it puts the extra requests into a waiting queue until it has spare capacity. If a request in a queue cannot be served within 2000 ms, it will be forwarded to the server. We compared our proposed chunk provider selection strategy to the random selection strategy, in which randomly selected nodes are recommended to the chunk requester as chunk providers.

Fig. 20(a) and (b) shows the average channel switch delay versus the average node capacity (in the Pareto distribution) for the two provider selection strategies on PeerSim, respectively. From Fig. 20(a), we see that the switch delay decreases as the capacity increases for all five methods. This is because when nodes in the system have more capacity to serve chunk requests, the chunk requests are less likely to be delayed. Comparing Fig. 20(a) and (b), we see that our chunk provider selection strategy generates the channel switch delay reduction ranging from 10 to 220 ms for all five methods. The random selection strategy may forward a chunk request to an overloaded node, which increases switch delay. Our strategy selects nodes with high available capacities as the chunk providers, thus avoiding

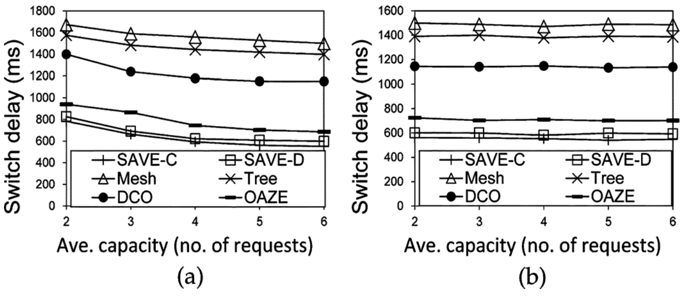


Fig. 20. Effectiveness of the chunk provider selection strategies on PeerSim. (a) Random selection. (b) Capacity-based selection.

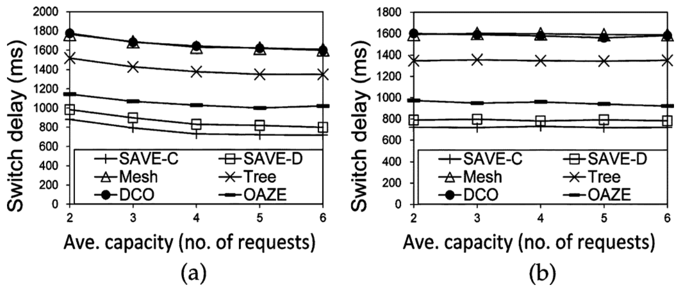


Fig. 21. Effectiveness of the chunk provider selection strategies on PlanetLab. (a) Random selection. (b) Capacity-based selection.

request serving delay. Therefore, our strategy is effective in reducing the average channel switch delay. When the average capacity increases, the improvement made by our proposed strategy decreases because when most nodes in the system have high capacities, the effect of selecting high-capacity nodes as chunk providers is not very obvious.

Fig. 21(a) and (b) shows the average channel switch delay versus the average node capacity (in the Pareto distribution) when using the random selection strategy and using our proposed selection strategy on the PlanetLab, respectively. We can make the same observations as in Fig. 20(a) and (b) for the same reasons. Our proposed strategy leads to about 3 to 200 ms reduction in switch delay for all five methods. When the average capacity is 2, the latency reduction for SAVE-C and SAVE-D is 160 and 200 ms, respectively. These results further confirm the effectiveness of our capacity-based chunk provider selection strategy.

V. Conclusion

In this paper, we propose SAVE, a social-network-aided efficient P2P live streaming system. SAVE supports successive- and multiple-channel viewing with low switch delay and low server overhead by enhancing the operations of joining and switching channels. SAVE considers the historical channel switching activities as the social relationships among channels and clusters the frequently interacted channels together by merging overlays or building bridges between the overlays. It maximizes the probability that existing users can locate their desired channels within its channel cluster and can take the bridges for channel switches. In addition, each node has a friendlist that records nodes with similar watching patterns, which is used to join a new channel overlay. SAVE also has the channel-closeness-based chunk pushing strategy and capacity-based chunk provider selection strategy to enhance its system performance. Our survey on user video streaming watching activities confirms the necessity and feasibility of SAVE. Through the experiments on the PeerSim simulator and PlanetLab testbeds, we prove that SAVE outperforms other representative systems in terms of overhead, video streaming efficiency and server load reduction, and the effectiveness of SAVE's two strategies. Our future work lies in further reducing the cost of SAVE in structure maintenance and node communication. Also, we will design algorithms for cluster separation and decentralized cluster head election.

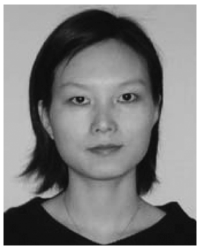
Acknowledgment

The authors would like to thank Mr. Z. Li for his help in the PlanetLab experiment and Mr. H. Wang for proofreading the conference version of this paper. An early version of this work was presented in the Proceedings of the ACM International Conference on Multimedia (MM), Nara, Japan, October 9–November 2, 2012 [60].

References

[1] "PPLive," [Online]. Available: http://www.pplive.com
[2] "UUSee," [Online]. Available: http://www.uusee.com
[3] "PPStream," [Online]. Available: http://www.ppstream.com
[4] J. Liu, S. G. Rao, B. Li, and H. Zhang, "Opportunities and challenges of peer-to-peer internet video broadcast," *Proc. IEEE*, vol. 96, no. 1, pp. 11–24, Jan. 2008.
[5] C. Wu and B. Li, "Exploring large-scale peer-to-peer live streaming topologies," *Trans. Multimedia Comput.*, vol. 4, no. 3, 2008, Art. no. 19.
[6] F. Dobrian, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," in *Proc. ACM SIGCOMM*, 2011, pp. 362–373.
[7] X. Cheng and J. Liu, "NetTube: Exploring social networks for peer-to-peer short video sharing," in *Proc. IEEE INFOCOM*, 2009, pp. 1152–1160.
[8] M. Mcpherson, "Birds of a feather: Homophily in social networks," *Annu. Rev. Sociol.*, vol. 27, no. 1, pp. 415–444, 2001.
[9] C. Wilson, B. Boe, A. Sala, K. Puttaswamy, and B. Zhao, "User interactions in social networks and their implications," in *Proc. EuroSys*, 2009, pp. 205–218.
[10] A. Fast, D. Jensen, and B. Levine, "Creating social networks to improve peer-to-peer networking," in *Proc. ACM SIGKDD*, 2005, pp. 568–573.
[11] A. Iamnitchi, M. Ripeanu, and I. Foster, "Small-world file-sharing communities," in *Proc. IEEE INFOCOM*, 2004, pp. 952–963.
[12] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proc. STOC*, 2000, pp. 163–170.
[13] S. Milgram, "The small world problem," *Psychol. Today*, vol. 2, no. 1, pp. 60–67, 1967.
[14] I. Bermudez, M. Mellia, and M. Meo, "Investigating overlay topologies and dynamics of P2P-TV systems: The case of SopCast," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1863–1871, Oct. 2011.
[15] F. Wang, J. Liu, and Y. Xiong, "Stable peers: existence, importance, and application in peer-to-peer live video streaming," in *Proc. IEEE INFOCOM*, 2008, pp. 2038–2046.
[16] J. Mendes, P. Salvador, and A. Nogueira, "P2P-TV service and user characterization," in *Proc. IEEE CIT*, 2010, pp. 2612–2620.
[17] "The PeerSim simulator," 2013 [Online]. Available: http://peersim.sf.net
[18] "PlanetLab," [Online]. Available: http://www.planet-lab.org/
[19] Y. Chen, E. Merrer, Z. Li, Y. Liu, and G. Simon, "OAZE: A network-friendly distributed zapping system for peer-to-peer IPTV," *Comput. Netw.*, vol. 56, no. 1, pp. 365–377, 2012.
[20] H. Shen, Z. Li, and J. Li, "A DHT-aided chunk-driven overlay for scalable and efficient peer-to-peer live streaming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 11, pp. 2125–2137, Nov. 2012.
[21] Y. Chu, A. Ganjam, T. Ng, S. Rao, K. Sripanidkulchai, J. Zhang, and H. Zhang, "Early experience with an internet broadcast system based on overlay multicast," in *Proc. USENIX*, 2004, p. 12.
[22] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proc. SIGCOMM*, 2002, pp. 205–217.

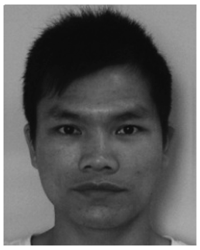
[23] Y. Chu, S. Rao, and H. Zhang, "A case for end system multicast," in *Proc. ACM SIGMETRICS*, 2000, pp. 1–12.
[24] R. Tian, Q. Zhang, Z. Xiang, Y. Xiong, X. Li, and W. Zhu, "Robust and efficient path diversity in application-layer multicast for video streaming," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 8, pp. 961–972, Aug. 2005.
[25] S. Asaduzzaman, Y. Qiao, and G. Bochmann, "CliqueStream: An efficient and fault-resilient live streaming network on a clustered peer-to-peer overlay," in *Proc. P2P*, 2008, pp. 269–278.
[26] X. Zhang, J. Liu, B. Li, and T. P. Yum, "CoolStreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming," in *Proc. IEEE INFOCOM*, 2005, pp. 2102–2111.
[27] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng, "AnySee: Peer-to-peer live streaming," in *Proc. IEEE INFOCOM*, 2006, pp. 1–10.
[28] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr, "Chainsaw: Eliminating trees from overlay multicast," in *Proc. IPTPS*, 2005, pp. 127–140.
[29] T. Locher, S. Schmid, and R. Wattenhofer, "eQuus: A provably robust and locality-aware peer-to-peer system," in *Proc. P2P*, 2006, pp. 3–11.
[30] Y. Guo, C. Liang, and Y. Liu, "AQCS: Adaptive queue-based chunk scheduling for P2P live streaming," in *Proc. IFIP Netw.*, 2008, pp. 433–444.
[31] L. Massoulie, A. Twig, C. Gkantsidis, and P. Rodriguez, "Randomized decentralized broadcasting algorithms," in *Proc. IEEE INFOCOM*, 2007, pp. 1073–1081.
[32] F. Picconi and L. Massoulie, "Is there a future for mesh-based live video streaming?," in *Proc. P2P*, 2008, pp. 289–298.
[33] N. Magharei and R. Rejaie, "PRIME: Peer-to-peer receiver-driven mesh-based streaming," in *Proc. IEEE INFOCOM*, 2007, pp. 1415–1423.
[34] J. Venkataraman and P. Francis, "Chunkyspread: Multi-tree unstructured peer-to-peer multicast," in *Proc. IPTPS*, 2006, pp. 1–10.
[35] F. Wang, Y. Xiong, and J. Liu, "mTreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *Proc. ICDCS*, 2007, p. 49.
[36] J. Mol, A. Bakker, J. Pouwelse, D. Epema, and H. Sips, "The design and deployment of a bittorrent live video streaming solution," in *Proc. ISM*, 2009, pp. 342–349.
[37] Y. Liu, "Delay bounds of chunk-based peer-to-peer video streaming," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1195–1206, Aug. 2010.
[38] C. Wu, B. Li, and S. Zhao, "Multi-channel live P2P streaming: Refocusing on servers," in *Proc. IEEE INFOCOM*, 2008, pp. 2029–2037.
[39] M. , L. Xu, and B. Ramamurthy, "A flexible divide-and-conquer protocol for multi-view peer-to-peer live streaming," in *Proc. P2P*, 2009, pp. 291–300.
[40] J. Liang, B. Yu, Z. Yang, and K. Nahrstedt, "A framework for future internet-based TV broadcasting," in *Proc. IPTV*, 2006, pp. 1–6.
[41] M. Wang, L. Xu, and B. Ramamurthy, "Channel-aware peer selection in multi-view peer-to-peer multimedia streaming," in *Proc. ICCCN*, 2008, pp. 1–6.
[42] D. Wu, Y. Liu, and K. Ross, "Modeling and analysis of multichannel P2P live video systems," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1248–1260, Aug. 2010.
[43] F. Ramos, J. Crowcroft, R. Gibbens, P. Rodriguez, and I. White, "Channel smurfing: Minimising channel switching delay in IPTV distribution networks," in *Proc. ICME*, 2010, pp. 1327–1332.
[44] A. Kermarrec, E. Merrer, Y. Liu, and G. Simon, "Surfing peer-to-peer IPTV: Distributed channel switching," in *Proc. Euro-Par*, 2009, pp. 574–586.
[45] X. Cheng, C. Dale, and J. Liu, "Statistics and social network of YouTube videos," in *Proc. IWQoS*, 2008, pp. 229–238.
[46] P. Salvador and A. Nogueira, "Study on geographical distribution and availability of BitTorrent peers sharing video files," in *Proc. ISCE*, 2009, pp. 1–4.
[47] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," *J. SIAM*, vol. 9, no. 4, pp. 551–570, 1961.
[48] J. Hao and J. Orlin, "A faster algorithm for finding the minimum cut in a directed graph," *J. Algor.*, vol. 17, no. 3, pp. 424–446, 1994.
[49] M. Newman, "Modularity and community structure in networks," in *Proc. PNAS*, 2006, pp. 8577–8582.
[50] K. Xu, M. Zhang, J. Liu, Z. Qin, and M. Ye, "Proxy caching for peer-to-peer live streaming," *Comput. Netw.*, vol. 54, no. 7, pp. 1229–1241, 2010.
[51] Y. Huang, T. Fu, D. Chiu, J. Lui, and C. Huang, "Challenges, design and analysis of a large-scale p2p-vod system," in *Proc. SIGCOMM*, 2008, pp. 375–388.
[52] Z. Liu, C. Wu, B. Li, and S. Zhao, "UUSee: Large-scale operational on-demand streaming with random network coding," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.
[53] C. Huang, J. Li, and K. W. Ross, "Can internet video-on-demand be profitable?," in *Proc. SIGCOMM*, 2007, pp. 133–144.
[54] "The difference between upload and download speed for broadband DSL," [Online]. Available: http://www.broadbandinfo.com/cable/speed-test
[55] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatrianin, "Watching television over an IP network," in *Proc. IMC*, 2008, pp. 71–84.
[56] Move Networks, American Fork, UT, USA, "Move Networks," [Online]. Available: http://www.movenetworkshd.com/
[57] X. Zhang, Y. Qu, and L. Xiao, "Improving distributed workload performance by sharing both CPU and memory resources," in *Proc. ICDCS*, 2000, pp. 233–241.
[58] K. Psounis, P. M. Fernandez, B. Prabhakar, and F. Papadopoulos, "Systems with multiple servers under heavy-tailed workloads," *Perform. Eval.*, vol. 62, no. 1–4, pp. 456–474, 2005.
[59] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson, "Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level," *IEEE/ACM Trans. Netw.*, vol. 5, no. 1, pp. 71–86, Feb. 1997.
[60] H. Shen, Z. Li, H. Wang, and J. Li, "Leveraging social network concepts for efficient peer-to-peer live streaming systems," in *Proc. ACM Multimedia*, 2012, pp. 249–258.
[61] "Surveymonkey: Free online survey software and questionnaire tool," [Online]. Available: http://www.surveymonkey.com/



Haiying Shen (M'07–SM'13) received the B.S. degree in computer science and engineering from Tongji University, Shanghai, China, in 2000, and the M.S. and Ph.D. degrees in computer engineering from Wayne State University, Detroit, MI, USA, in 2004 and 2006, respectively.

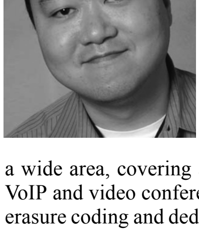
She is currently an Associate Professor with the Electrical and Computer Engineering Department, Clemson University, Clemson, SC, USA. Her research interests include distributed computer systems and computer networks with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and cloud computing.

Dr. Shen is a Microsoft Faculty Fellow of 2010 and a member of the Association for Computing Machinery (ACM).



Yuhua Lin received the B.S. and M.S. degrees in computer engineering from Sun Yat-sen University, Guangzhou, China, in 2009 and 2012, respectively, and is currently pursuing the Ph.D. degree in electrical and computer engineering at Clemson University, Clemson, SC, USA.

His research interests include social networks and reputation systems.



Jin Li (S'94–A'95–M'96–SM'99–F'12) received the Ph.D. degree in electrical engineering from Tsinghua University, Beijing, China, in 1994.

He is currently a Principal Researcher managing the Multimedia Communication and Storage Team with Microsoft Research, Redmond, WA, USA. His invention has been integrated into many Microsoft products, such as Microsoft Office Communicator/Lync, Live Messenger, Live Mesh, Windows 7, Windows 8, etc. He holds 45 issued US patents. He has published in top conferences and journals in a wide area, covering audio/image/video compression, multimedia streaming, VoIP and video conferencing, P2P networking, distributed storage system with erasure coding and deduplication, and high-performance storage system design.