Leveraging Social Network Concepts for Efficient Peer-to-Peer Live Streaming Systems

Haiying Shen, Ze Li and Hailang Wang
Dept. of Electrical and Computer Engineering
Clemson University, Clemson, SC 29634
{shenh, zel, hailanw}@clemson.edu

Jin Li
Microsoft Research
Redmond, WA 98052
jinl@microsoft.com

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. The increase in the number of channels in today's live streaming applications 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, which if widely used, poses heavy burden on the centralized 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 multi-channel watching as interactions between channels. By collecting the information of channel interactions and 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. Extensive experimental results from the PeerSim simulator and PlanetLab verify that SAVE outperforms other popular protocols in system efficiency and server load reduction.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; H.3.5 [**Online Information Services**]: Data sharing

General Terms

Algorithms, Design, Performance

Keywords

P2P live streaming, Social networks, P2P networks

MM'12, October 29–November 2, 2012, Nara, Japan.

1. INTRODUCTION

Peer-to-Peer (P2P) live streaming applications [1, 2, 3] such as PPLive and UUSee are attracting millions of viewers every day. 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. Nowadays, the wide coverage of broadband Internet enables users to enjoy live streaming programs effortlessly, and the increase of channels triggers users' desire of watching multiple channels successively or simultaneously (i.e., multi-channel watching mode). A typical multi-channel interface contains one main view and one or more secondary views (i.e., Picture in Picture (PIP)), 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. 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 take part in maintaining multiple overlays. As a node opens more channels, its maintenance cost for overlay connections increases dramatically. Also, in order to join in an overlay, a node needs to ask the server which nodes it can connect to. Thus, the successive-channel or multi-channel watching of millions of users poses heavy burden on the centralized server and delayed response leads to inefficiency in P2P live streaming systems [4].

In this paper, we aim to improve the efficiency and scalability of P2P live streaming systems with many users engaging many successive-channel watching or multi-channel 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 multi-channel 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 ac-

tivity is driven by its interests [5, 6]. 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 is mostly limited to a small number of channels that it is mostly interested in [7, 8, 9]. 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 multi-channel watching, nodes can stay in the same overlay or take the inter-channel 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. Following the small-world property [10, 9, 11] that a node can always find another node within a limited number of hops, a node in a channel can find a node in another channel in a few steps via social connections between friends. Therefore, each node in SAVE maintains a friendlist recording nodes sharing common-interest channels and watching time periods. When a node wants to switch to a channel which is not in its current cluster or when a node returns to the system, it refers to its friendlist to find nodes in the desired channel to join 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. The two schemes contribute to three main features of SAVE listed below, and hence enhancing the system efficiency and scalability and the QoS in terms of 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 time cost of joining a channel [4]. Switching channels in SAVE in most cases does not cause users to leave 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 sever 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. Survey results confirm the social network properties in P2P live streaming systems and demonstrate the feasibility and necessity of SAVE to a certain extent. We conducted simulation in PeerSim [12] and deployed SAVE on PlanetLab [13]. Extensive experimental results prove that SAVE outperforms other popular protocols in terms of system efficiency and server load reduction.

2. RELATED WORK

P2P live streaming channel overlays. P2P live streaming overlays fall into four categories: tree [14, 15, 16, 17, 18, 19], mesh [20, 21, 22, 23, 24, 25, 26, 27], hybrid structure [28, 29, 30, 31, 32, 33] and Distributed Hash Table (DHT) [34]. In tree-based methods, parent nodes push all received chunks to their children. Mesh-based methods [21, 22, 23] connect nodes in a random manner. Each node usually serves a number of nodes while also receives chunks from others. Mesh-based methods are naturally resilient to churn, but they generate higher overhead. Hybrid methods synergistically combine tree-based and mesh-based structures. Wang *et al.* [30, 31] proposed a hybrid structure consisting of two tiers. PRIME [28] is a hybrid system featuring segmented and two-phase chunk delivery. It builds a tree in which nodes' topological distances to the server depend on their hops away from the server. The DHT-based structure [34] builds a hierarchical structure, in which the upper tier consists of stable DHT nodes and the lower tier consists of normal nodes. A video chunk requester relies on DHT functions to find a chunk owner for subsequent chunk retrieval. SAVE can help this approach to join multiple DHT overlays of different channels based on user preference to enhance efficiency.

Multi-channel P2P live streaming techniques. Current works on multi-channel P2P live streaming are mainly focused on bandwidth allocation optimization. In order to optimize the allocation of each node's upload capacity to each channel it is watching, Wu *et al.* [35, 36, 37] used game theory to resolve the conflicts in allocating bandwidth. By noticing that overlays are overlapped in multi-channel applications, DAC [38] divides the nodes in an overlap to several virtual logical nodes, 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 [22] uses the mesh overlay structure and maps logical overlay to the physical location topology. Path selection is based on physical location topology. Wang *et al.* [39] established linear programming models to answer a question: under what circumstances, should a particular design be used to achieve the desired streaming quality with the lowest implementation complexity?

Social network aided video-on-demand systems. There are few works on video streaming with the aid of social networks. In the video-on-demand system area, Cheng *et al.* [40] investigated the social networks in YouTube videos. They further proposed NetTube [5] 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.

3. DESIGN OF THE SAVE SYSTEM

3.1 An Overview of SAVE

Figure 1 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. When a node wants to join an overlay (i.e., to watch a channel), it asks for nodes in the overlay from n_s and then joins in the overlay through the recommended nodes. 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. We assume that the cluster heads will not be overloaded in this paper, and leave the study of the additional load for functioning as cluster heads as our future work. SAVE has two main schemes: channel clustering and friendlist.

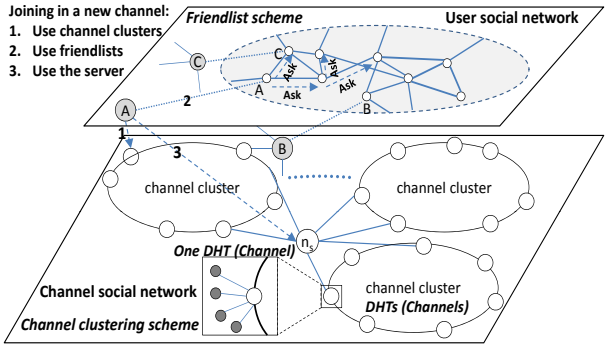


Figure 1: A high-level view of the SAVE structure.

Channel clustering scheme (Section 3.2). This scheme considers the interactions between channels, and connects the frequently-interacted channels (i.e., connects a group of nodes with similar successive- and multi-channel watching activities). As time proceeds, 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. Each cluster also has a cluster head, denoted by h_{cr}, connecting with all h_cs 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 3.3). The friendlist scheme enables a node to maintain a friendlist recording nodes with similar individual channel watching patterns (i.e., interest channels and watching time). When a node wants to join a channel which is not in its current cluster, it can quickly join in the channel relying on the friendlist. 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 the 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 a bridge connecting to the new channel. If it fails, it uses its friendlist. If it cannot find a friend in its desired channel within certain hops, it resorts to the server finally.

3.2 Channel Clustering

We define $s(x, y)$ as a switching activity from channel x to channel y and define $m(x, y)$ as a multi-channel watching activity on both channels x and y. For a node's activity of $s(x, y)$, we define *age* as the time interval between when the switching occurred and the current time, which is used to represent the freshness of the switching activity. We define *duration(x)* as the time interval that the node stays in channel x before the switching of $s(x, y)$, and define *duration(y)* as the time interval that the node stays in channel y before its next switching. Similarly, for a node's activity of $m(x, y)$, we define *age* as the time interval between when the multi-channel watching occurred and the current time. We define *duration(x)* and *duration(y)* as the time interval that the node stays in both channels x and y before the multi-channel watching stops. Parameter $I(x) = 1$ when $duration(x) \geq T_s$; otherwise $I(x) = 0.1$, where T_s is a predefined threshold. In SAVE, each node has a profile that lists the node's interested channels specified by the node. If both x and y are in the node's interested channels, we consider the switching non-accidental, and set the value of parameter $inTags$ to 1. Otherwise, we set $inTags = 0.1$ in order to minimize the influence of accidental switching activities.

The *duration* and *inTags* reflect, to a certain extent, whether both x and y are the node's favorite channels, and the switching or multi-channel watching is the node's routine activity. We use *channel closeness* of two channels to reflect the frequency of interactions between the channels. If a node's switching or multi-channel watching activity occurred more recently, it spent a certain time period in both channels, and $inTags = 1$, it means the node tends to watch both channels routinely. Such activities of many nodes indicate the higher closeness between the two channels. The channel head of channel y keeps a record of channel watching and switching activities of nodes in channel y, and calculates the *channel closeness* between channel x and y by

$$C(x, y) = \sum_{\Omega} \frac{I(x) \cdot I(y)}{\omega^{age}} \cdot inTags, \quad (1)$$

where Ω stands for all the activities of $s(x, y)$ and $m(x, y)$, ω is the scaling parameter which exponentially reduces the freshness of switching and watching activities.

In fact, the closeness of two channels can be regarded as an undirected weighted link between the two channels in the social network graph, and the channel clustering is the process of grouping channels with high-weight links. SAVE aims to generate clusters that maximize the number of intra-cluster interactions and minimize the inter-cluster interactions. Driven by this objective, we first develop a centralized method using the server to collect global inter-channel activities for channel clustering. Then, we further develop a decentralized method to cluster channels by utilizing the local inter-channel activity information.

3.2.1 Centralized Channel Clustering

In the centralized clustering method, the centralized server node n_s needs the information of closeness between channels for channel clustering. To fulfill this information collection task, we use a two-tier information aggregation method, in which the activities are reported from nodes to their channel heads which calculate channel closeness and report the information to the server. For the channel clustering, the server relies on the minimum cut tree based algorithm, which generates sub-optimal clustering result but has low computation complexity. First, the server 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 algorithm to divide the vertices in the entire graph to subsets. Consequently, all channels are clustered into different channel clusters. As shown in Figure 2, a *cut* is a divide of all the nodes V in graph G that separates G into two node 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 inter-cluster cuts values and relatively large sum of intra-cluster cut values. Algorithm 1 shows the pseudo-code of the clustering algorithm. First, we insert an artificial sink t into the graph $G(V, E)$ (step 1). The sink is connected with all nodes in the graph with weight α ($0 \leq \alpha \leq 1$) (steps 2-4). α is used to control the number of generated clusters. If $\alpha = 0$, all channels will be in one giant cluster, while $\alpha = 1$ will make all nodes become singletons. Then, we use the maximum flow algorithm [41] which involves a recursive process to construct a minimum cut tree with the minimum sum of the values of cuts (step 5). After that, we remove the sink and graph G is divided into several clusters (steps 6-7). The intra-cluster cut value can be used to measure the tightness of channels in each cluster. If a cluster has tightness higher than a pre-defined threshold, its channels are merged to one overlay. Otherwise, its channels build bridges between each other.

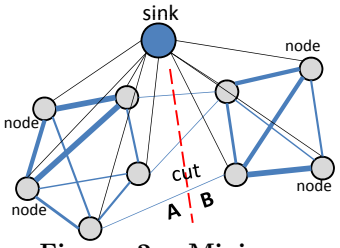


Figure 2: Minimum-cut tree-based clustering algorithm.

Algorithm 1: Centralized channel clustering algorithm executed by n_s.

1 $V' = V \cup t$; *//t is the sink;*
2 *//Connect t to V to generate an expanded graph* $G'(V', E')$;
3 **for** *all nodes* $v \in V$ **do**
4 $\quad$ *Connect v to t with an edge of weight* α;
5 *Calculate the minimum cut tree* T' *of* G';
6 *Remove t from* T';
7 *Divide G to clusters with small sum of inter-cluster cut values and large sum of intra-cluster cut values;*
8 **Return** *all connected sub-graphs as the clusters of G;*

3.2.2 Decentralized Channel Clustering

For ease of presentation, we use *channel cluster* to denote both individual channels and a cluster of multiple channels. Each cluster has a head, which is the most stable node with the highest capacity and longest lifetime staying in the cluster in all channel heads in the cluster. In the decentralized method, each cluster head collects both the intra-cluster and inter-cluster communications in a distributed manner in order to find the clusters with frequent interactions to group with. Consequently, the created clusters converge to a stable status, i.e., they have small sum of inter-cluster closeness and relatively large sum of intra-cluster channel closeness.

The channel head of channel y belonging to channel cluster cr_i collects the information of $I(x)$, $I(y)$, age and $inTags$ for activities of $s(x, y)$ and $m(x, y)$, and reports the information to its cluster head h_{cr_i}. Then, h_{cr_i} builds a *cluster stability vector* denoted by $\mathcal{V}_d$ with length d, which is the number of channel clusters that its cluster has interacted with. We define an interaction from cluster cr_i to cluster cr_j as a channel switching from a channel in cr_i to a channel in cr_j. Each element in the vector $V(cr_i, cr_j)$ ($1 \leq j \leq d$), called *stability value*, is defined as:

$$V(cr_i, cr_j) = \sum_{\Psi} \frac{I(x) \cdot I(y)}{\omega^{age}} \cdot \nabla \cdot inTags, \quad (2)$$

where ($x \in cr_i, y \in cr_i$ *or* cr_j), Ψ stands for all the activities of $s(x, y)$ and $m(x, y)$ of channel x and channel y; channel x must be a channel in cr_i, while channel y can be a channel in cr_i or cr_j. This means the interactions can be within cr_i or between cr_i and cr_j. When $y \in c_{r_i}$, $\nabla = 1$ and when $y \in c_{r_j}$, $\nabla = -1$. Thus, the interactions between channels in cr_i increase the stability value, and the interactions between cr_i and cr_j decrease the stability value. $V(cr_i, cr_j)$ indicates the stability of cr_i relative to cr_j, and higher $V(cr_i, cr_j)$ means more channel interactions within cr_i than between cr_i and cr_j. This metric is used to form clusters so that the number of intra-cluster interactions is maximized while the number of inter-cluster interactions is minimized.

We use $\mathcal{V}_d^i$ and $\mathcal{V}_d^k$ to denote the $\mathcal{V}_d$ of cr_i and cr_k, respectively. If we group cr_i and cr_k into one cluster (i.e., $cr_{(i,k)} = cr_i \cup cr_k$), the inter-cluster interactions between cr_i and cr_k become intra-cluster interactions. Then, each element in the cluster stability vector of $cr_{(i,k)}$, $V(cr_{(i,k)}, cr_a)$ ($cr_a \in \mathcal{V}_d^i \cup \mathcal{V}_d^k$), equals:

$$V(cr_i, cr_a) + V(cr_k, cr_a) + V(cr_i, cr_k) + V(cr_k, cr_i), \quad (3)$$

where $V(cr_i, cr_a) = 0$ if $cr_a \notin \mathcal{V}_d^i$, and $V(cr_k, cr_a) = 0$ if $cr_a \notin \mathcal{V}_d^k$.

A channel cluster with higher popularity would attract more viewers, which shows the future trend in channel switching. Thus, we further consider the popularity of each channel cluster, measured by the number of users in the cluster, denoted by p, in calculating the cluster stability. If the channels in cr_j are very popular, nodes in cr_i have a higher tendency to watch the channels in cr_j and vice versa. We define *inverse popularity vector* ($\mathcal{P}_d$), which includes $1/p$ of each channel cluster of cr_i's interacted clusters, and calculate the *cluster stability degree* D by

$$D = \mathcal{P}_d \cdot {\mathcal{V}_d}^{\top}. \quad (4)$$

Channel clusters with larger D are more stable, i.e., more interactions occur within the cluster than outside the cluster.

Thus, in the decentralized channel clustering method, each cluster head periodically collects the information of intra-cluster and inter-cluster interactions between its own cluster and other clusters. It also asks for $\mathcal{V}_d$ from the cluster head of every other channel clusters it knows, and calculates the D of the cluster that combines its channel cluster and that cluster. Two clusters are combined if their combination leads to higher D than their individual Ds. Specifically, the head of each channel cluster cr_i conducts Algorithm 2 to identify channel clusters in all clusters Θ known to itself that it can group with so that its D is maximized. h_{cr_i} selects each cluster cr_k (step 3) and generates a virtual cluster $cr_{(i,k)} = cr_i \cup cr_k$ (step 4). Then, h_{cr_i} calculates this virtual cluster's D (step 6-10). If $cr_{(i,k)}$ is more stable than cr_i ($D_{cr_i} < D$), cr_i and cr_k will be combined. Finally, cr_i finds the cluster to group with that can make the cluster more stable, i.e., the number of inter-cluster interactions is minimized and the number of intra-cluster interactions is maximized (step 15).

If channel cluster cr_i selects cr_k to combine with, it sends an invitation to cr_k along with its vector $\mathcal{V}_d^i$. cr_k then decides whether adding cr_i will cause an increase of its D based on its own knowledge and $\mathcal{V}_d^i$. If yes, it accepts the invitation. If $D_{cr_{(i,k)}}$ is greater than a pre-defined threshold, two

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

Calculate $\mathcal{V}_d^i$ *and* D_{cr_i}*;*
for *each channel cluster* $cr_k \in (\Theta - cr_i)$ **do**
  $cr_{(i,k)} = cr_i \cup cr_k$ *;*
  Ask for $\mathcal{V}_d^k$ *from* h_{cr_k}*;*
  for *each current channel cluster* cr_a *in* $\mathcal{V}_d^i \cup \mathcal{V}_d^k$ **do**
    Calculate $V(cr_{(i,k)}, cr_a)$ *in* $\mathcal{V}_d^{(i,k)}$ *(Equ.(3));*
  Prepare $\mathcal{P}_d$*;*
  Calculate $D_{cr_{(i,k)}} = \mathcal{P}_d \times {\mathcal{V}_d}^{(i,k)^\top}$*;*
  if $D_{cr_i} < D_{cr_{(i,k)}}$ **then**
    //$cr_{(i,k)}$ *is more stable than* cr_i*;*
    Return cr_k*; //return the selected* cr_k

Table 1: The record for a friend in the friendlist.

IP address	Similarity	Profile	Entry creation time

clusters merge to one overlay. Otherwise, they build bridges between each other. In order to make sure that the channels in one cluster have frequent interactions, each cluster head also periodically calculates the degree of its cluster excluding each channel. If excluding one channel leads to higher D, then the cluster head splits the channel from its cluster.

3.2.3 Cluster Combination and Partition

We use "cluster combination and partition" to represent both "bridge construction and removal" and "overlay integration and separation". SAVE has a hierarchical tree structure composed of the server, cluster heads (only in the decentralized clustering method), channel heads and nodes in each level from the top to the bottom. In the centralized clustering method, the server notifies channel heads to combine or partition. In the decentralized clustering method, the cluster heads communicate between each other for the cluster combination and partition. After a bridge is built between several channel clusters, the more stable and higher-capacity cluster head becomes the cluster head of the new cluster. The cluster head notifies the channel heads in other clusters, which update their connections with the new cluster head. The method for the overlay combination is based on the overlay construction method in different overlay structures. The newly selected head maintains a record of the head in each overlay before combination, which is used for possible subsequent overlay partition.

For bridge removal, the notified channel head removes the bridges to other channels in the cluster and becomes the head of the separated channel. The cluster head also notifies the channel heads in other channels in the cluster, which remove corresponding bridges. For overlay partition, the cluster head or server notifies the previous head of the identified cluster to separate its overlay from the cluster, and the remaining nodes in the cluster conduct corresponding updates.

3.3 Friendlist Construction

In SAVE, each node maintains a profile based on its own channel watching activities as shown in Figure 3. The "Interest tag" is a channel category such as comedy, sports and news that a node likes to watch. 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 along with their increasing watching activities. In the figure, the "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 hours 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 each digit stands for 3 hours and the user usually watches video from 10:00am to 1:00pm and 6:00pm to 9:00pm.

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
...		...	...	...

Figure 3: The profile of a node.

To determine the similarity of two nodes' watching patterns, we consider not only their common interests but also their common frequently watched channels and the overlap of their watching time periods. We define the *similarity* of two nodes, n_i and n_j, as the product of the proportion of overlapping interest tags and the active vector,

$$S(n_i, n_j) = \sum_{tag_i \cap tag_j} S_{cl}(n_i, n_j) \times S_{av}(n_i, n_j), \qquad (5)$$

where $tag_i \cap tag_j$ is the common tags between n_i and n_j; $S_{cl}(n_i, n_j)$ is the similarity between their two channel lists cl_i and cl_j, defined as $S_{cl}(n_i, n_j) = \frac{|cl_i \cap cl_j|}{|cl_i \cup cl_j|}$; $S_{av}(n_i, n_j)$ is the similarity of active vectors v_i and v_j of node n_i and n_j, define as $S_{av}(n_i, n_j) = \frac{|v_i \cap v_j|}{T}$. The *similarity* of two nodes represents the probability that they watch the same channel at the same time. Unlike other social network based methods [5] that only consider common interests, SAVE considers both interests and watching time for friend clustering, which leverages user routine behavior for efficient video sharing.

Each node in SAVE maintains a friendlist recording a certain number of friends sharing high similarity with itself. The record of a friend consists of the items listed in Table 1. One question is how a node can find friends to build and update its friendlist. Recall that for channel clustering, nodes report their channel switching activities to their channel heads. Then, the channel heads periodically report the information to the server in the centralized channel clustering method, and to their cluster heads which further forward the information to the server in the decentralized channel clustering method. Thus, nodes can piggyback their profiles on the reports. Consequently, the channel heads, and (or) cluster heads and (or) the server have profiles of many nodes. When node n_i communicates with the heads or the server during video watching, it can piggyback its profile on the message. The heads or server then send back a list of friends sharing high similarity with n_i, which updates 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 pre-defined threshold are discarded.

3.4 Efficient Multi-Channel Video Streaming

When node n_i initially joins in SAVE, it requests n_s to recommend nodes in its desired channel. The node then joins in the channel overlay by connecting to the recommended

nodes and retrieve and share chunks with other nodes in the overlay. 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 multi-channel watchings on these channels), when a node in a merged overlay wants to switch channel or watch multi-channels, it can find its requested chunks from its current overlay using the original chunk search algorithm with high probability. Because the bridged channels are relatively close to each other, a node can easily use the bridges to join in the overlay of its desired channel with high probability. If there is no bridge connecting to the desired channel, the node then uses its friendlist, and finally resorts to the server. The bridges in a channel cluster help a node join a channel frequently watched by itself and other nodes. The friendlist helps a node join a channel frequently watched by itself and its friends. The centralized server handles the case that a node occasionally watches a channel outside of its interests.

To use bridges for switching 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 finds the owner of the requested chunk. 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 for joining c_j. n_i sends a request with TTL to all of its friends and 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 its channel overlay. It 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 TTL=0 will notify n_i the failure of search. Then, n_i resorts to the server.

3.5 Structure Maintenance in Node Churn

The 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, when a current channel head is departing, 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 its cluster head in the decentralized method. In the decentralized clustering method, before a cluster head leaves, it notifies all channel heads in its cluster and the server about the new cluster head. The notified nodes update their connections accordingly. The new channel head or cluster head is a newly selected node. The joins and departures of normal nodes are handled by the original protocol in the P2P live streaming system. 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, a new head will be elected. SAVE can use multiple channel heads and cluster heads to enhance system reliability, but at the cost of higher maintenance overhead.

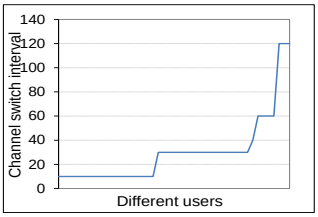


Figure 4: Channel switch interval.

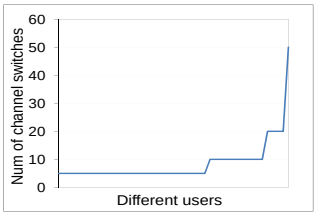


Figure 5: Number of channel switches.

4. SURVEY RESULTS

We conducted an online survey[1] on SurveyMonkey [42] on live streaming video watching activities. Though the survey is not as strong as real trace data, the survey results confirm the social network properties in online live streaming systems to a certain extent. We collected 60 responses in the community of undergraduate and graduate students in Clemson University. We will study more characteristics of successive and simultaneous multiple channel accessing in our future work.

4.1 Channel Switching Activities

One question in the survey is "how long on average do you watch a channel before you switch to another?" Figure 4 shows the channel switch interval in minutes for each user while watching streaming videos. We can see that most users have an interval between 10-30 minutes. 43% users have an interval of 20 minutes, 18% have an interval of more than 40 minutes and 7% have an interval of 120 minutes. The average interval between channel switches for all users is 30 minutes. The result shows a relatively frequent channel switching pattern of most users. We use *session* to represent the process that a user logs in the system, watches the streaming videos and logs out the system. Figure 5 plots the average number of channel switches per session of each user. We see 67% users switch channel 5 times, 22% switch channel 10 times, 9% and 2% switch channel 20 and 50 times, respectively. We then reach the following observation:

Observation(O) 1. *On average, a significant percent of users switch channels relatively frequently. A small percent of users stay in a channel for a relatively long time. Most users switch a certain number of channels per session, and a small percent of users switch channels many times per session.*

The frequent channel switch requests from millions of users would overload the centralized server and increase the latency of responses. This shows the necessity of SAVE. The nodes staying in a channel for a relatively long time can function as cluster heads that build bridges between each other in order to link the channels in a cluster. This reduces the maintenance overhead of bridges.

4.2 Feasibility of Channel Clustering

The result in Figure 5 implies that most users may have a certain number of favorite channels. From Figure 6, we see that the number of interests of each user varies from 1 to 10, with 50% of all users having no more than 3 interests and 96% of all users having no more than 7 interests. Based on Figures 5 and 6, we can reach:

[1]Survey link: http://www.surveymonkey.com/s/CQ2SZ2G

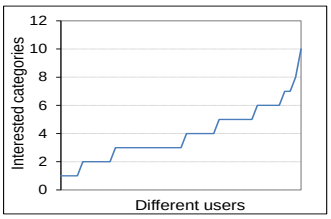


Figure 6: Number of interests.

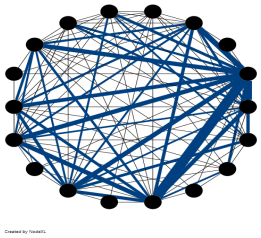


Figure 7: Interest correlation.

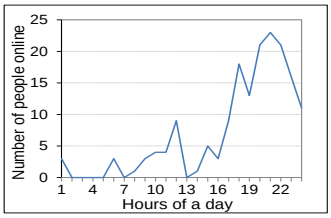


Figure 8: Online periods.

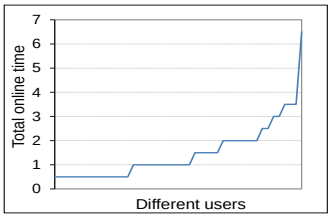


Figure 9: Online time.

O 2. *On average, most users usually watch channels in a few interests. A very small percentage of users switch between many channels.*

SAVE merges or bridges the channels that most users always watch, these users can share chunks in multiple channels or directly jump between the channels without relying on the centralize server. For the very small percent of users that switch between many channels, they can use friendlists for the switching. The server is needed for random channel switches.

We drew a graph for the 18 interests in the questionnaire, shown in Figure 7, in which a vertex represents an interest and a link connecting two interest vertices means one user has both interests. The link between two interest nodes has higher weight when these two interests are shared by more users. We observe that a certain number of interests have high-weight links between each other. This means that a relatively large percent of users share certain interests. In SAVE, clustering the channels in these interests by merging or bridging can facilitate fast switching between these channels. Also, the benefits of clustering would be more than the cost of cluster maintenances since many nodes would benefit from the merged overlay and bridges. We also find that some vertices have links with median weights. The nodes with these interests can use friendlists for switching channels. We arrive at an observation:

O 3. *Many nodes sharing common interests (i.e., watching and switching between the same channels) makes channel clustering and friendlist schemes viable and effective approaches for high video steaming efficiency.*

4.3 Video Watching Behavior Pattern

The user watching behaviors have a large impact on the performance of SAVE in terms of peer online time and channel switching behavior. Figure 8 shows the number of online users at specific hours within the day. We can see that most people watch videos between 18:00-23:00, which we called the "active period" (five hours in total). If we consider a node that has been online for more than half of this period as stable node, then from Figure 9, we can see that about 25% nodes qualify. Actually, according to [31], in video streaming applications, nodes generally have around 1.5 hours online time.

O 4. *Users are online for a relatively long time, with a considerable part of them being stable nodes.*

This observation is consistent with that in [43]. The stable nodes can serve as cluster heads to connect channels in a cluster together by building bridges between each other. Then, the adverse effect of churn on the clusters of SAVE is mitigated.

5. PERFORMANCE EVALUATION

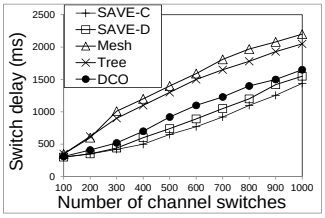
We conducted experiments on the event-driven simulator PeerSim and PlanetLab which is a high-performance computing cluster in Clemson University. Each test lasts for 24 hours. In our simulation, the system consists of 10000 nodes and 100 channels. We set the video bit rate, which is the size of a video segment per second, to 600kbps. The ω in Equ.(2) was set to 1.02. 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 in these two tests.

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 Figure 6. We regarded one interest as an interested channel. We divided the 30 or 100 channels to 5 groups, and each group has 6 or 20 channels. Each node chose 90% of its interested channels from one group randomly chosen from the 5 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. The warm-up period lasted for 2 hours. The online time for nodes is distributed according to the survey result in Figure 9. The channel switching time is distributed according to the survey result in Figure 4. A node periodically switches channel after its switching interval time has elapsed. In order to keep the same number of nodes in the test while simulating node churn, after a node leaves the system, another node joins in the system. The statistics in [44, 5] is for the distribution of the download bandwidth in the simulation. A node's upload bandwidth is set to 1/3 of its download bandwidth [45].

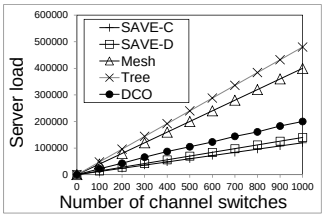
In all previous P2P live streaming systems, a node needs to contact the central server for changing channels. Since mesh structure is used in most current P2P live streaming systems, we first built SAVE on the mesh structure and compared it with mesh-based system [2] (Mesh), tree-based system [16] (Tree) and DHT-based system (DCO) [34]. For SAVE, we have two variations using the centralized (Section 3.2.1) and decentralized (Section 3.2.2) channel clustering methods, represented by "SAVE-C" and "SAVE-D", respectively. We also built SAVE upon DCO for comparison.

5.1 Switching Delay and Server Load

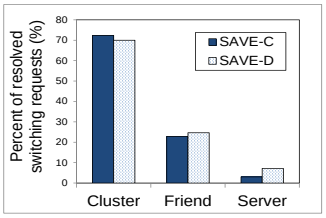
Figure 10(a) and Figure 11(a) show the switch delays in the simulation and PlanetLab experiments, respectively. We randomly chose 1000 switchings from all switchings. We then ordered the switch delays in an increasing order, calculated the average value of every 100 values, and finally got



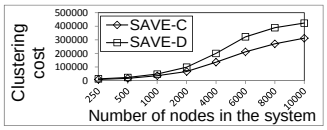
(a) Channel switching delay.



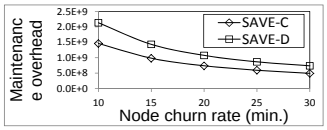
(b) Server load.



(c) Distribution of resolved switching requests.

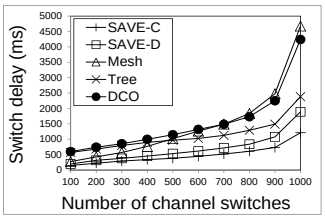


(d) Clustering cost.

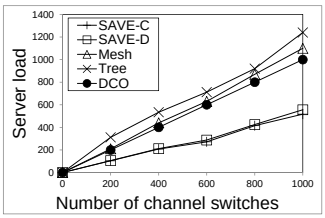


(e) Maintenance overhead.

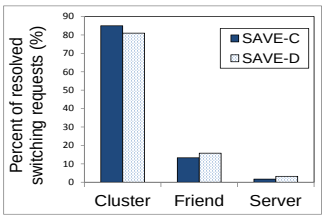
Figure 10: Experimental results from simulation on PeerSim.



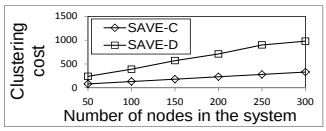
(a) Channel switching delay.



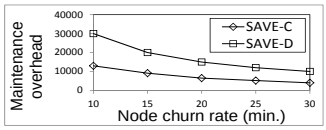
(b) Server load.



(c) Distribution of resolved switching requests.



(d) Clustering cost.



(e) Maintenance overhead.

Figure 11: Experimental results from the PlanetLab testbed.

10 average values. We see that both SAVE-C and SAVE-D achieve the fastest switching, which confirms their highly efficient video streaming. In other systems, a node needs to contact the centralized server in order to join in another overlay, 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 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. This is 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, in a less stable environment in the PlanetLab testbed, DHT nodes may fail, leading to chunk owner location failures and longer delay. We also observe that the delay on PlanetLab exhibits 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.

Figure 10(b) and Figure 11(b) show the server load (total number of requests served by the server) 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. This is because nodes in Tree needs to contact the server node more frequently because 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 owners.

5.2 Effectiveness of the Social Network in SAVE

Figures 10(c) and 11(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 percent of switch requests are resolved by clusters. A moderate percent of requests are resolved by friends and a very small percent of requests are resolved by the server. The results demonstrate the effectiveness of the channel clustering and friendlist schemes. We also see that SAVE-C has higher percent in using clusters than SAVE-D because SAVE-C has higher accuracy in clustering channels than SAVE-D with global information.

5.3 Cost of SAVE

Figures 10(d) and 11(d) 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.

Next, we test the maintenance overhead of SAVE in node churn measured by the total number of messages to handle node joins and departures. 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. Figures 10(e) and 11(e) show the maintenance

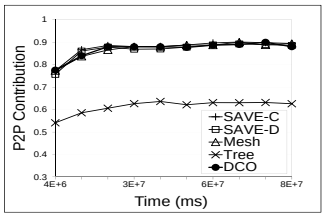


Figure 12: P2P contribution.

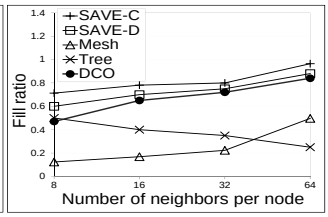


Figure 13: Fill ratio.

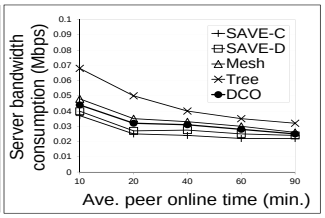


Figure 14: Server bandwidth consumption.

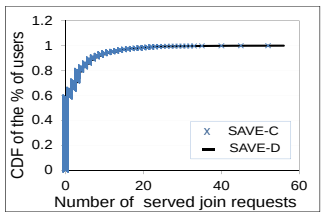


Figure 15: Load balance status.

overhead of SAVE-C and SAVE-D in the simulation and PlanetLab experiments, respectively. As the average lifetime increases, the maintenance overhead decreases. This is because slower node join and departure rate generates less messages for maintaining the SAVE structure. This is because in SAVE-C, channel heads connect with the server, while in SAVE-D, channel heads connect with cluster heads which connect with the server. We can see that SAVE's clustering cost and maintenance overhead is acceptable.

5.4 Overall P2P Performance

Below, we present simulation results for SAVE built on the DHT structure. Figure 12 shows the P2P contribution percentages over time measured by the number of chunk requests that are resolved by peers rather than the server. We see that all the percentages exhibit an increase at the beginning and then become stable. As time goes on, more and more chunks are disseminated to nodes, and chunks can be retrieved from nodes. Tree has a lower P2P contribution because it is more vulnerable to churn.

Fill ratio is defined as the ratio of nodes holding a chunk at a certain time. Figure 13 is the average fill ratio of different systems two seconds after a chunk is distributed by the server. We see that SAVE achieves much higher fill ratio than DCO. Because of SAVE's channel clustering and friendlist schemes, a node can join or switch channels without relying on the server. In DCO, a node needs to contact the centralized server in order to join in another overlay, generating a certain delay in chunk dissemination. SAVE-D has a slightly lower fill ratio than SAVE-C because the decentralized method is not as accurate as the centralized method in channel clustering, increasing the probability that a node's desired channel is not in its current cluster.

We also see that SAVE and DCO generate higher fill ratio than Mesh. In SAVE and DCO, the chunk discovery in each channel is based on DHT, which can guarantee chunk provider discovery. After a provider is located, the client directly asks the provider for subsequent chunks. In Mesh, a node pulls a chunk from their neighbors. As the number of neighbors per node increases, the fill ratio of Mesh increases significantly because more neighbors means chunks can be more quickly delivered to nodes. Tree's fill ratio is better than Mesh's when the number of neighbors is small, but becomes worse as the number increases. This is because each node has a bandwidth constraint. The chunk dissemination speed slows down as the number of children of a node increases.

Figure 14 shows the server bandwidth consumption as a function of the average peer online time. The results follow Tree>Mesh>DCO>SAVE-D>SAVE-C. SAVE based on DHT achieves the least sever bandwidth consumption due to its cluster clustering and friendlist schemes. SAVE-D>SAVE-C is because the centralized clustering method is more accurate. DCO provides high chunk availability with the aid of DHT, generating less sever bandwidth consumption than Mesh. Tree exhibits the highest server side bandwidth consumption because nodes have a higher probability of failing to obtain chunks from its parents under high network churn.

Figure 15 shows the CDF of nodes versus the load (number of channel or switch requests served) of different nodes in SAVE-C and SAVE-D. We see that up to 90% of all nodes have a load less than 20 requests, while more than 20% of all nodes have the opportunity to server other nodes. Hence, SAVE can utilize a significant portion of all nodes in a relatively even manner. Balance load distribution will not overload a node and thus user experience is not greatly degraded.

6. CONCLUSIONS

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 optimizing 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. This 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 recording nodes with similar watching patterns, which is used to join a new channel overlay. 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. 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 cluster head re-election.

Acknowledgements

This research was supported in part by U.S. NSF grants OCI-1064230, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, CSR-2008826, CSR-2008827, Microsoft Research Faculty Fellowship 8300751, and Oak Ridge Award 4000111689.

7. REFERENCES

[1] PPLive. http://www.pplive.com.
[2] UUSee. http://www.uusee.com.
[3] PPStream. http://www.ppstream.com.
[4] F. Dobrian, V. Sekar, I. Stoica, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. of SIGCOMM*, 2011.

[5] X. Cheng and J. Liu. Nettube: Exploring social networks for peer-to-peer short video sharing. In *Proc. of INFOCOM*, 2009.

[6] M. Mcpherson. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27(1):415–444, 2001.

[7] C. Wilson, B. Boe, A. Sala, K. P.N. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *Proc. of EuroSys*, pages 205–218, 2009.

[8] A. Fast, D. Jensen, and B. Levine. Creating social networks to improve peer-to-peer networking. In *Proc. of SIGKDD*, 2005.

[9] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-World File-Sharing Communities. In *Proc. of INFOCOM*, 2004.

[10] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proc. of STOC*, pages 163–170, 2000.

[11] S. Milgram. The small world problem. *Psychology Today*, 1967.

[12] The PeerSim simulator. http://peersim.sf.net.

[13] PlanetLab. http://www.planet-lab.org/.

[14] 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. of USENIX*, 2004.

[15] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of SIGCOMM*, Pittsburgh, PA, USA, 2002.

[16] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS*, 2000.

[17] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Proc. of INFOCOM*, 2003.

[18] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanid-kulchai. Distributed streaming media content using cooperative networking. In *Proc. of ACM NOSSDAV*, 2002.

[19] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *Proc. of SOSP*, 2003.

[20] 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. of P2P*, 2008.

[21] 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. of INFOCOM*, 2005.

[22] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng. AnySee: Peer-to-Peer Live Streaming. In *Proc. of IEEE INFOCOM*, 2006.

[23] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: eliminating trees from overlay multicast. In *Proc. of IPTPS*, 2005.

[24] T. Locher, S. Schmid, and R. Wattenhofer. eQuus: a provably robust and locality-aware peer-to-peer system. In *Proc. of P2P*, 2006.

[25] Y. Guo, C. Liang, and Y. Liu. Adaptive queue-based chunk scheduling for P2P live streaming. In *Proc. of IFIP Networking*, 2008.

[26] L. Massoulie, A. Twig, C. Gkantsidis, and P. Rodriguez. Randomized decentralized broadcasting algorithms. In *Proc. of INFOCOM*, 2007.

[27] F. Picconi and L. Massoulie. Is there a future for mesh-based live video streaming? In *Proc. of P2P*, 2008.

[28] N. Magharei and R. Rejaie. PRIME: peer-to-peer receiver-driven mesh-based streaming. In *Proc. of INFOCOM*, 2007.

[29] J. Venkataraman and P. Francis. Chunkyspread: multi-tree unstructured peer-to-peer multicast. In *Proc. of IPTPS*, 2006.

[30] F. Wang, Y. Xiong, and J. Liu. mTreebone: a hybrid tree/mesh overlay for application-layer live video multicast. In *Proc. of ICDCS*, 2007.

[31] F. Wang, J. Liu, and Y. Xiong. Stable Peers: Existence, Importance, and Application in Peer-to-Peer Live Video Streaming. In *Proc. of INFOCOM*, pages 1364–1372, 2008.

[32] J. Mol, A. Bakker, J. Pouwelse, D. Epema, and H. Sips. The design and deployment of a bittorrent live video streaming solution. In *Proc. of ICM*, 2009.

[33] Y. Liu. Delay bounds of chunk-based peer-to-peer video streaming. *TON*, 18(4):1195–1206, 2010.

[34] H. Shen, L. Zhao, Z. Li, and J. Li. A DHT-Aided Chunk-Driven Overlay for Scalable and Efficient Peer-to-Peer Live Streaming. In *Proc. of ICPP*, 2010.

[35] C. Wu and B. Li. Strategies of conflict in coexisting streaming overlays. In *Proc. of INFOCOM*, pages 481–489, 2007.

[36] C. Wu, B. Li, and Z. Li. Dynamic bandwidth auctions in multioverlay p2p streaming with network coding. *IEEE TPDS*, 2008.

[37] C. Wu, B. Li, and S. Zhao. Multi-channel live P2P streaming: Refocusing on servers. In *Proc. of INFOCOM*, 2008.

[38] M., L. Xu, and B. Ramamurthy. Flexible divide-and-conquer protocol for multi-view peer-to-peer live streaming. In *Proc. of P2P*, 2009.

[39] M. Wang, L. Xu, and B. Ramamurthy. Linear Programming Models For Multi-Channel P2P Streaming Systems. In *Proc. of INFOCOM*, 2010.

[40] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *Proc. of IWQoS*, 2008.

[41] R. Gomory and T. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551 – 570, 1961.

[42] Surveymonkey: Free online survey software and questionnaire tool. http://www.surveymonkey.com/.

[43] F. Wang, J. Liu, and Y. Xiong. Stable peers: existence, importance, and application in peer-to-peer live video streaming. In *Proc. of IEEE INFOCOM*, 2008.

[44] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable? In *Proc. of SIGCOMM*, 2007.

[45] The difference between upload and download speed for broadband DSL. http://www.broadbandinfo.com/cable/speed-test.