Fast Source Switching for Gossip-based Peer-to-Peer Streaming

Zhenhua Li [1,2], Jiannong Cao [2], Guihai Chen [1], and Yan Liu [2]
[1] State Key Lab for Novel Software Techonology, Nanjing University, China
[2] Internet and Mobile Computing Lab, Hong Kong Polytechnic University
lizhenhua@dislab.nju.edu.cn, csjcao@comp.polyu.edu.hk, gchen@nju.edu.cn, and csyliu@comp.polyu.edu.hk

Abstract

In this paper we consider gossip-based Peer-to-Peer streaming applications where multiple sources exist and they work serially. More specifically, we tackle the problem of fast source switching to minimize the startup delay of the new source. We model the source switch process and formulate it into an optimization problem. Then we propose a practical greedy algorithm that can approximate the optimal solution by properly interleaving the data delivery of the old source and the new source. We perform simulations on various real-trace overlay topologies to demonstrate the effectiveness of our algorithm. The simulation results show that our proposed algorithm outperforms the normal source switch algorithm by reducing the source switch time by 20%-30% without bringing extra communication overhead, and the reduction ratio tends to increase when the network scale expands.

1 Introduction

In general, existing Peer-to-Peer (P2P) streaming systems can be classified into two categories: tree-based and gossip-based. The gossip-based method is often named as mesh-based. Tree-based systems [1,2,7,11] organize nodes into a multicast tree. The root of the tree is the media source and data segments are always delivered from parent to children. Tree-based method can minimize redundant data delivery and ensure full coverage of data dissemination, but cannot well adapt to network dynamics because the failure of a single node will partition the tree to a forest. Gossip-based systems have been proved to be effective and resilient especially in dynamic and heterogeneous network environments. In a typical gossip algorithm [4], every node maintains a limited number of neighbors and sends a newly generated or received data segment to a random subset of its neighbors. The random choice of data forwarding targets achieves high resilience to random failures and enables distributed operations. However, direct use of gossip for streaming is ineffective because its random push may cause significant redundancy. As a result, existing gossip-based P2P streaming systems, e.g. CoolStreaming [10], PeerStreaming [5] and AnySee [6], adopt a smart pull-based gossip algorithm: every node periodically exchanges data availability information with its neighbors and then retrieves required data segments from a subset of its neighbors.

A gossip-based P2P streaming system may have one source or multiple sources which disseminate data segments to other nodes. For a multiple-source system, the sources may work serially or in parallel. For example, in a video conferencing system or a distance education system, every member can become the streaming source but there is usually only one source (that is the speaker) at a time so the sources work serially. In this paper we consider gossip-based P2P streaming applications where multiple sources exist and they work serially. In this scenario, one critical problem is how to make the source switch process fast, that is to say, how to minimize the startup delay of the new source.

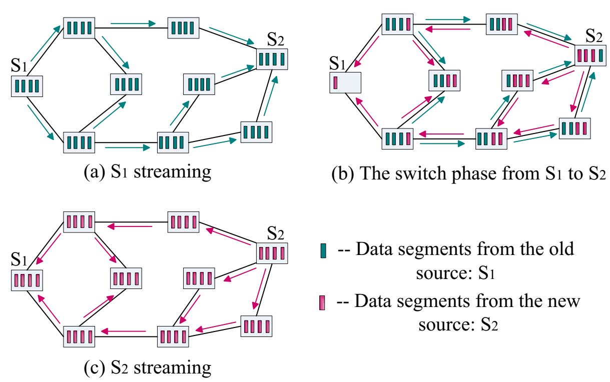

Figure 1. A source switch process from the old source S_1 to the new source S_2.

Figure 1 demonstrates a source switch process. It is composed of three phases. (a) At first the old source S_1

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.21

was streaming its contents and every node was receiving and playing the data segments of S_1. (b) Then S_1 stopped streaming and the new source S_2 started streaming. Both the data segments of S_1 and S_2 were being disseminated amongst all the non-source nodes. (c) Finally every node had finished the whole playback of S_1, and only the data segments of S_2 were being disseminated in the system. Obviously, the source switch problem is essentially how to minimize the duration of phase (b). More specifically, we need to design a proper source switch algorithm for every node to minimize its playback start time (or says the startup delay) of S_2, on condition that a node can start its playback of S_2 only when 1) it has finished the whole playback of S_1, and 2) it has gathered sufficient data segments of S_2.

In this paper we first model the source switch process by capturing its essential features, formulate the source switch problem into an optimization problem, and deduce the optimal solution to this optimization problem. Then we propose a practical greedy algorithm, named *fast switch algorithm*, that can approximate the optimal solution by properly interleaving the data delivery of the old source and the new source. This algorithm is triggered and executed by every node independently and it relies on only local computation.

We have done comprehensive simulations on various real-trace overlay topologies, scaling from 100 to 10000 nodes, to demonstrate the effectiveness of our algorithm. The simulation results show that our proposed fast switch algorithm outperforms the *normal switch algorithm* by reducing the source switch time by 20%-30% without bringing extra communication overhead, and the reduction ratio tends to increase when the network scale expands. The normal switch algorithm does not interleave the data delivery of the old source and the new source. Instead, it always gives priority to the data delivery of the old source. The example in Figure 2 shows the difference between the two algorithms. The current node can receive 7 data segments per scheduling period but there exist 10 available data segments, 5 of S_1 and 5 of S_2. Each algorithm arranges the order of data delivery according to its own computation of the data priorities.

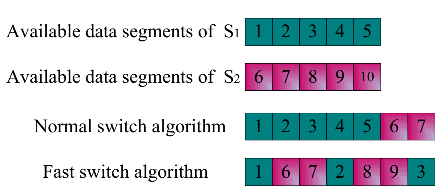

Figure 2. A comparison of our fast switch algorithm and the normal switch algorithm.

Our contributions can be summarized as follows:

1. To the best of our knowledge, we are the first to investigate the source switch problem of gossip-based P2P streaming. We model the source switch process and formulate it into an optimization problem.

2. We propose a practical greedy algorithm that can approximate the optimal solution by properly interleaving the data delivery of the old source and the new source.

3. We demonstrate the effectiveness of our proposed algorithm through comprehensive simulations on various real-trace overlay topologies.

The rest of this paper is organized as follows. Section 2 overviews related work. Section 3 models the source switch process. Section 4 presents our proposed fast source switch algorithm and we evaluate its performance by simulation in Section 5. Finally, we conclude the paper and point out the future work in Section 6.

2 Related Work

Existing gossip-based P2P streaming systems optimize some performance aspects like playback continuity, startup delay, bandwidth utilization, and so on. Our work optimizes the source switch time, which is the startup delay of the new source. Such optimization is different from the traditional optimization of startup delay because it takes into consideration the playback requirements of both the old source and the new source. Besides, since our proposed algorithm accelerates the source switch process, it indirectly increases the playback continuity and bandwidth utilization.

CoolStreaming [10] utilizes the gossip-based membership protocol [4] to construct a practical and resilient streaming system. It provides support of multiple sources but exhibits little description about its source switch mechanism. The P2P live streaming system AnySee [6] employs locality-aware and inter-overlay optimizations to improve performance aspects like startup delay, source-to-end delay, etc. However, we have not seen its consideration of source switch methods.

Zhang et al. [9] observe that *pure-pull* method in P2P streaming brings tremendous latency and thus propose a *push-pull* system called GridMedia. They classify the streaming packets into pulling packets and pushing packets. A pulling packet is delivered by a neighbor only when the packet is requested, while a pushing packet is relayed by a neighbor as soon as it is received. The main goal of GridMedia is to reduce latency and it has the extra effect of accelerating the source switch process. However, pushing packets would bring considerable communication overhead.

Xu et al. [8] consider the problem of media data assignment for a multi-supplier P2P streaming session. Given a requesting peer and a set of supplying peers with heterogeneous out-bound rates, their algorithm, named OTS_{p2p}, computes optimal media data assignments for P2P streaming sessions to achieve minimum buffering delay and thus to reduce the startup delay. But OTS_{p2p} has very strict assumptions that can hardly hold in practical gossip-based P2P streaming systems.

3 Model the Source Switch Process

Since the P2P streaming system we consider is fully distributed, a node does not know the source switch process until it discovers data segments of a new source in its neighbors, that is to say, the source switch algorithm assumes no knowledge on the ordering of the sources' sessions. When a node discovers the new source it triggers its source switch algorithm to execute and then re-executes the algorithm per scheduling period until it finishes the whole playback of the old source. We assume there exists a mechanism for synchronizing the old source S_1 and the new source S_2 so that S_2 knows when S_1 finishes streaming and adds the id of S_1's ending segment into S_2's first several data segments to notify the other nodes. Such synchronization mechanism is out of this paper's range so we do not address it here.

The parameters used in modeling the source switch process are shown in Table 1. We use Figure 3 to visualize these parameters. The stream from S_1 is played once Q consecutive data segments of S_1 have been gathered, but the stream from S_2 is started to play when the first Q_s data segments of S_2 have been gathered. In a practical P2P streaming system usually Q_s is configured much bigger than Q to guarantee a smooth startup of the new source. The total inbound rate I is a constant and I is divided into I_1 and I_2 to receive data segments of S_1 and S_2 respectively. I_1 and I_2 are dynamically configured by the source switch algorithm.

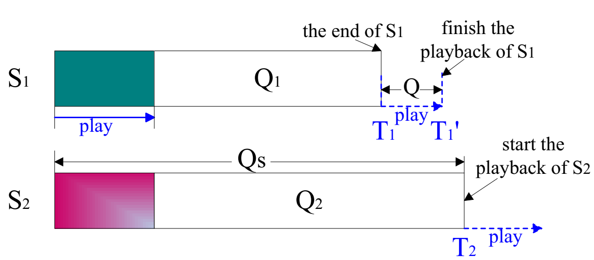

Figure 3. The time sequence graph corresponding to our model.

The problem of fast source switching can be formulated into the following optimization problem:

Table 1. Model parameters

Param	Description
S_1	The old source.
S_2	The new source.
Q	The stream from S_1 is played once Q consecutive data segments of S_1 have been gathered.
Q_1	The number of undelivered data segments of S_1.
Q_s	The number of required data segments of S_2 to start the playback of S_2.
Q_2	The number of undelivered data segments of S_2 to start the playback of S_2. Initially Q_2=Q_s.
p	The number of segments being played per second.
I	Total inbound rate of the local node. The rate is measured by the number of data segments per second. I is a constant.
I_1	The inbound rate allocated to receive data segments of S_1. I_1 is dynamically configured.
I_2	The inbound rate allocated to receive data segments of S_2. I_2 is dynamically configured.
T_1	The expected time to receive all the undelivered data segments of S_1.
T_1'	The expected time to finish the playback of S_1.
T_2	The expected time to receive the first Q_s data segments of S_2.

Minimize T_2
subject to the following conditions:

$$\begin{cases} I = I_1 + I_2; \\ T_1 = \frac{Q_1}{I_1}; \\ T_1' = T_1 + \frac{Q}{p}; \\ T_2 = \frac{Q_2}{I_2}; \\ T_2 \geq T_1'; \end{cases}$$

The conditions can be rewritten as

$$\begin{cases} T_1' = \frac{Q_1}{I_1} + \frac{Q}{p}; \\ T_2 = \frac{Q_2}{I - I_1}; \\ T_2 \geq T_1'; \end{cases}$$

So we get the inequality

$$\frac{Q_2}{I - I_1} \geq \frac{Q_1}{I_1} + \frac{Q}{p}; \qquad (1)$$

which can be rewritten as

$$I_1^2 + (\frac{p(Q_1 + Q_2)}{Q} - I)I_1 - \frac{pIQ_1}{Q} \geq 0; \qquad (2)$$

Solving the above inequality, we have the following

$$I_1 \geq r_1 \quad or \quad I_1 \leq r_1'; \qquad (3)$$

$$r_1 = \frac{I - \frac{p(Q_1+Q_2)}{Q} + \sqrt{\left(\frac{p(Q_1+Q_2)}{Q} - I\right)^2 + \frac{4pIQ_1}{Q}}}{2} \quad (4)$$

$$r'_1 = \frac{I - \frac{p(Q_1+Q_2)}{Q} - \sqrt{\left(\frac{p(Q_1+Q_2)}{Q} - I\right)^2 + \frac{4pIQ_1}{Q}}}{2} \quad (5)$$

Clearly $r'_1 < 0$ and thus r'_1 is not a reasonable solution. $I_1 \geq r_1$ is the only solution. Therefore, in order to minimize T_2 we let $I_1 = r_1$ and $I_2 = r_2 = I - r_1$, which is the optimal solution to the optimization problem.

4 Fast Source Switch Algorithm

The ideal condition for achieving the optimal solution does not always hold when applied to practical systems because the real environments usually involve more complicated constraints. Therefore, we need a practical source switch algorithm that can approximate the optimal solution.

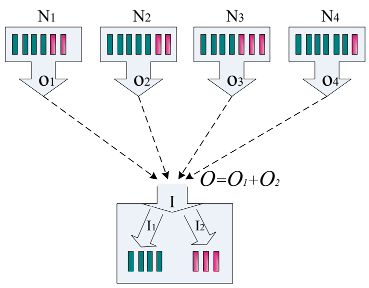

Figure 4. The local working environment of a node.

Figure 4 demonstrates the local working environment of a node. The local node has neighbors N_1, N_2, N_3, N_4 with outbound rate o_1, o_2, o_3, o_4 respectively. Suppose O_1 is the total available outbound rate for the data delivery of S_1 and O_2 is the total available outbound rate for the data delivery of S_2, then the optimization problem in Section 3 changes to:

Minimize T_2
subject to the following conditions:

$$\begin{cases} I_1 + I_2 \leq I; \\ I_1 \leq O_1; \\ I_2 \leq O_2; \\ T_1 = \frac{Q_1}{I_1}; \\ T'_1 = T_1 + \frac{Q}{p}; \\ T_2 = \frac{Q_2}{I_2}; \\ T_2 \geq T'_1; \end{cases}$$

Under the above conditions, the solution $I_1 = r_1, I_2 = r_2$ we get in Section 3 can only hold when $r_1 \leq O_1$ and $r_2 \leq O_2$. r_1 is defined in the equation (4) and $r_2 = I - r_1$. Therefore, when $r_1 > O_1$ or $r_2 > O_2$ we try to maximize the inbound throughput of the local node. Then the solutions become:

- Case 1: when $r_1 \leq O_1$ and $r_2 \leq O_2$, then $I_1 = r_1, I_2 = r_2$;
- Case 2: when $r_1 \leq O_1$ and $r_2 > O_2$, then $I_1 = \min(O_1, I - O_2), I_2 = O_2$;
- Case 3: when $r_1 > O_1$ and $r_2 \leq O_2$, then $I_1 = O_1, I_2 = \min(O_2, I - O_1)$;
- Case 4: when $r_1 > O_1$ and $r_2 > O_2$, then $I_1 = O_1, I_2 = O_2$;

Now the critical problem is how to compute O_1 and O_2, more exactly, to compute the two sets $\mathbb{O}_1$ and $\mathbb{O}_2$, where $O_1 = |\mathbb{O}_1|$ and $O_2 = |\mathbb{O}_2|$. Data segments in $\mathbb{O}_1$ are in descending order of their priorities and $\mathbb{O}_2$ is alike. Required parameters for our algorithm are listed in Table 2.

Table 2. Parameters for our algorithm

Param	Description
τ	Data scheduling period.
id_i	The id of data segment D_i.
n_i	The number of neighbors that can supply the data segment D_i.
R_{i_j}	The receiving rate of segment D_i from the jth neighbor.
R_i	The maximum receiving rate of segment D_i.
id_{play}	The id of the segment being played at this moment.
id_{end}	The id of the ending segment of S_1.
id_{begin}	The id of the beginning segment of S_2. We set $id_{begin} = id_{end} + 1$.
t_i	The expected deadline left time of segment D_i.
B	Buffer size, i.e. the number of data segments Buffer can accommodate.
p_{i_j}	Segment D_i's position in the jth neighbor's buffer. The replacement strategy of Buffer is FIFO, and the position is the distance from the tail of Buffer.
$urgency_i$	The urgency of segment D_i, i.e. the probability of D_i to miss its deadline.
$rarity_i$	The rarity of segment D_i, i.e. the probability that D_i will be replaced in all its suppliers' buffers.
$priority_i$	The requesting priority of segment D_i. It takes both urgency and rarity into consideration.

Taking both the urgency and rarity of each data segment into consideration, a data segment D_i's requesting priority is computed through equations (6) to (9).

$$R_i = \max\{R_{i_1}, R_{i_2}, \cdots, R_{i_{n_i}}\} \quad (6)$$

$$t_i = \frac{id_i - id_{play}}{p} - \frac{1}{R_i} \quad \text{then} \quad urgency_i = \frac{1}{t_i} \quad (7)$$

Segment i's rarity is the probability it will be replaced in all its suppliers' buffers, which we think is more reasonable than the traditional computation $rarity_i = \frac{1}{n_i}$.

$$rarity_i = (\frac{p_{i_1}}{B}) \times (\frac{p_{i_2}}{B}) \times \cdots \times (\frac{p_{i_{n_i}}}{B}) \quad (8)$$

And finally, $\quad priority_i = \max\{urgency_i, rarity_i\} \quad (9)$

Having got each segment's priority, our proposed fast source switch algorithm is able to compute $\mathbb{O}_1$, $\mathbb{O}_2$ and then arrange the data retrieval process, see Algorithm 1. The data segments are sorted in the descending order of their priorities. Usually the data segments of S_1 and S_2 are mixed in this order. Suppose the order is like $D_1, D_2, D_3, \cdots, D_m$. For a segment D_i, there may exist several neighbors who can supply it, and usually the neighbor who can send it earliest will become D_i's supplier. But here we encounter a conflict problem where two segments choose the same supplier, so one of them needs to wait or choose another supplier. The problem is: how to choose a proper supplier for every data segment so that the number of segments missing deadlines or being replaced can be the minimal? In fact, even a simple special case of this problem is NP-hard (known as the *Parallel machine scheduling problem* [3]), so we use a greedy algorithm trying to get high-priority segments as early as possible. In this algorithm, the scheduler makes greedy efforts to minimize the expected receiving time t_{min} of every data segment. For a data segment D_i, the scheduler checks all its suppliers to find a proper supplier which can send D_i earliest.

After getting $\mathbb{O}_1$ and $\mathbb{O}_2$, the computation of I_1 and I_2 follows one of the four cases described formerly. And the data retrieval is straightforward.

Algorithm 1 Fast Source Switch Algorithm

Input:
Data segments $D_1, D_2, D_3, \cdots, D_m$, in descending order of priority;
Supplier set for each segment: $S_1, S_2, S_3, \cdots, S_m$;
Sending rate of node j: R(j);
Queuing time of node j: $\tau(j)$, initially $\tau(j) = 0$;

Step 1: Computing $\mathbb{O}_1$ and $\mathbb{O}_2$
for $i = 1$ to m **do**
  set segment D_i's earliest receiving time $t_{min} = \infty$;
  suppose S_i contains k suppliers $S_{i_1}, S_{i_2}, \cdots, S_{i_k}$;
  for $j = 1$ to k **do**
    compute the expected transfer time of D_i from S_{i_j}: $t_{trans} = \frac{1}{R(S_{i_j})}$;
    if $t_{trans} + \tau(S_{i_j}) < t_{min}$ **and** $t_{trans} + \tau(S_{i_j}) < \tau$ **then**
      $t_{min} \leftarrow t_{trans} + \tau(S_{i_j})$; $supplier_i \leftarrow S_{i_j}$;
    end if
  end for
  if $supplier_i \neq null$ **then**
    $\tau(supplier_i) \leftarrow t_{min}$;
    add D_i to its corresponding set $\mathbb{O}_1$ or $\mathbb{O}_2$;
  end if
end for

Step 2: Arranging Data Retrieval
compute I_1 and I_2 according to $\mathbb{O}_1$, $\mathbb{O}_2$, r_1 and r_2;
retrieve the first I_1 data segments of $\mathbb{O}_1$;
retrieve the first I_2 data segments of $\mathbb{O}_2$;

5 Performance Evaluation

5.1 Simulation Methodology

To evaluate the performance of our algorithm we perform simulations on 30 real-trace P2P overlay topologies whose data was collected from Dec. 2000 to Jun. 2001 on dss.clip2.com (this web site is unavailable now). The data contains each node's ID, IP, host name, port, ping time, speed and so on, but we just use the ID, IP and ping time information. The trace topologies scale from 100 to 10000 nodes. Because their average node degree is too small for media streaming, we add random edges into each overlay to let every node hold M=5 connected neighbors. According to our simulation experience, M=5 is usually a good practical choice and using a larger M cannot bring more benefit. The default streaming rate is 300 Kbps and each data segment contains 30 Kb, so the playback rate p=$\frac{300Kb}{30Kb}$=10. Each node maintains a Buffer of 600 data segments. We randomly arrange inbound rate (from 300 Kbps to 1 Mbps) to each node and let the average inbound rate be 450 Kbps, i.e. $I \in [10, 33]$ and I=15 in average. The arrangement of outbound rate is alike. An exception is that the source node has zero inbound rate and much larger outbound rate. The data scheduling period τ=1.0 second.

For each simulation, we first let the system run for a sufficient period of time to enter its stable phase, and then stop S_1 from generating new data segments and meanwhile choose a new source S_2 to generate new data segments. Therefore, in all the following paragraphs the simulation time "0" means the time when S_1 stops and S_2 starts. The stream from S_1 is played once Q=10 consecutive data segments of S_1 have been gathered. The total number of required data segments of S_2 to start the playback of S_2 is Q_s=50.

We compare the performances of our fast switch algorithm with the normal switch algorithm. The normal switch algorithm works as follows: for a node n when its neighbors can supply data segments of both S_1 and S_2, node n would retrieve data segments of S_1 in priority. If n still has available inbound rate after retrieving data segments of S_1, it would allocate the remaining inbound rate to retrieve data segments of S_2.

5.2 Metrics

We mainly use the following three metrics to evaluate the performance of our fast switch algorithm:

1. *Average preparing time of S_2 (= Average switch time)* means the average time for all nodes to prepare sufficient data segments of S_2 to start the playback of S_2.

2. *Reduction ratio* means the reduction ratio of average source switch time by using the fast switch algorithm compared with using the normal switch algorithm.

3. *Communication overhead:* For every scheduling period each node exchanges buffer information with its neighbors. Communication overhead is defined as the ratio of communication cost for buffer information exchange over the real communication cost for data segments transfer.

We also measure some supplementary metrics which can help to understand the source switch process. The supplementary metrics include: (1) *Undelivered ratio of S_1 ($= \frac{Q_1}{Q_0}$)* means the ratio of the undelivered data segments of S_1 currently (Q_1) to the undelivered data segments of S_1 at time "0" (Q_0). (2) *Delivered ratio of S_2 ($= \frac{Q_s - Q_2}{Q_s}$)* means the ratio of the delivered data segments of S_2 ($Q_s - Q_2$) to the total required data segments of S_2 to start the playback of S_2 (Q_s). (3) *Average finishing time of S_1 ($= T_1'$)* means the average time for all nodes to finish the playback of S_1.

5.3 Simulation Results in Static Environments

We first track the undelivered ratio of S_1 and delivered ratio of S_2 of our fast switch algorithm and the normal switch algorithm in a static network environment with 1000 nodes. From Figure 5 we can see that the normal switch algorithm gathers the undelivered data segments of S_1 more quickly than the fast switch algorithm but prepares sufficient data segments to start the playback of S_2 more slowly. By using the normal switch algorithm, the last node finishes S_1 at time 15 but prepares S_2 at time 24. Note that the last node that finishes S_1 is usually different from the last node that prepares S_2. Meanwhile, by using the fast switch algorithm, the last node finishes S_1 and prepares S_2 both at time 18. So we can find the fast switch algorithm brings on a "compromise" between the speeds of gathering data segments of S_1 and S_2, and thus makes the whole source switch process faster.

We further examine the average finishing time of S_1 and average preparing time of S_2 of overlay networks with different sizes, ranging from 100 to 8000, working in static network environments. The bar graph in Figure 6 illustrates the results. For each size there are 4 bars corresponding to (from left to right): 1) the average finishing time of S_1 by using the normal switch algorithm; 2) the average finishing time of S_1 by using the fast switch algorithm; 3) the average preparing time of S_2 by using the fast switch algorithm; 4) the average preparing time of S_2 by using the normal switch algorithm. The 4 bars of each size indicates that the fast switch algorithm splits the difference between the average finishing time of S_1 and preparing time of S_2 of the normal switch algorithm, and thus makes the startup delay of the new source shorter. To illustrate the effect more clearly, the average switch time and its reduction by using the fast switch algorithm are shown in Figure 7. We can see the reduction ratio lies between 0.2 and 0.3, and it tends to increase when the network scale expands.

Besides, we measure the communication overhead of the two algorithms in overlay networks with different sizes. The buffer can accommodate $B = 600$ data segments, so we use 600 bits to record the data availability, with bit 1 indicating this segment is available and bit 0 indicating this segment is unavailable. The id of the first segment in the buffer is indicated by 20 bits because the source will disseminate at most $10 \times 3600 \times 24 = 864000 \in (2^{19}, 2^{20})$ data segments per day (one hour is 3600 seconds, and one day is 24 hours). Therefore, getting the buffer information of one neighbor takes 620 bits' communication cost in total. Every data segment contains 30 Kb data of streaming. If every node can get $p = 10$ required data segments from its neighbors per second, i.e. the data delivery rate just matches the media play rate, then the communication overhead is about $\frac{620 \times M}{30 \times 1024 \times 10} = \frac{5}{495} \approx 1\%$. Simulation results in Figure 8 are a little larger than 1% because in fact most nodes' data delivery rate cannot catch the media play rate. The communication overhead of the fast switch algorithm is a bit lower than that of the normal switch algorithm because the fast switch algorithm indirectly increases the bandwidth utilization.

5.4 Simulation Results in Dynamic Environments

To create a dynamic network environment, we randomly let 5% old nodes leave and 5% new nodes join per scheduling period. A new joining node does not need to retrieve all the disseminated data segments from each source, and

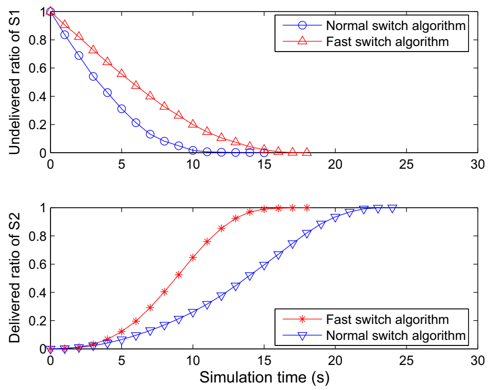

Figure 5. Ratio track in a static network with 1000 nodes.

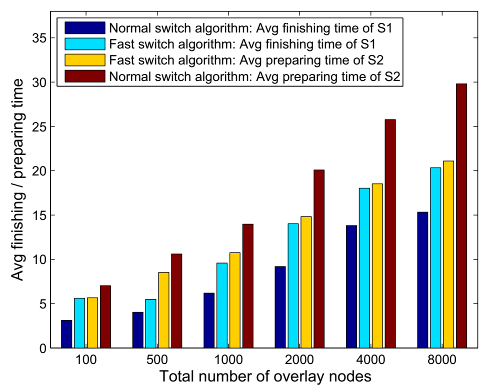

Figure 6. Avg finishing time of S_1 and preparing time of S_2 in static environments.

it just requests the data segments being played or will be played by its neighbors. That is to say, a new joining node starts its media playback by following its neighbors' current steps.

In general, simulation results in dynamic environments, as shown in Figure 9, 10, 11 and 12, are consistent with those in static environments.

6 Conclusion and Future Work

This paper discusses about how to minimize the delay of source switching between two sources in P2P streaming systems. we model the source switch process of gossip-based P2P streaming and formulate it into an optimization problem. Then we propose a practical greedy algorithm that can approximate the optimal solution by properly interleaving the data delivery of the old source and the new source. Simulation results confirm the effectiveness of our algorithm. Our current work considers the application scenario where multiple sources exist and they work serially. Next step we would try to extend our work to the scenario where multiple sources work in parallel.

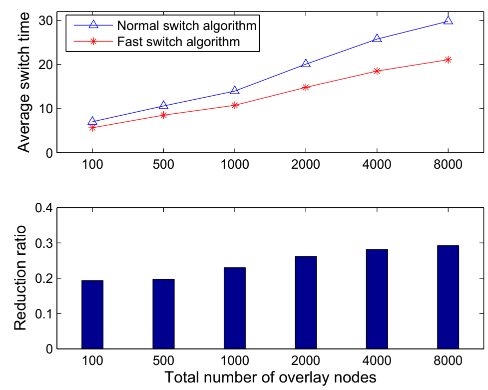

Figure 7. Avg switch time and its reduction ratio in static environments.

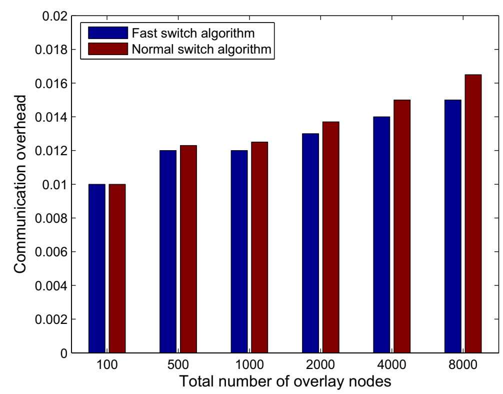

Figure 8. Communication overhead in static environments.

7 Acknowledgements

The work is partly supported by Hong Kong RGC under the CERG grant PolyU 5103/06E, China NSF grants (60573131, 60673154,60721002), Jiangsu High-Tech Research Project of China (BG2007039), and China 973 project (2006CB303000).

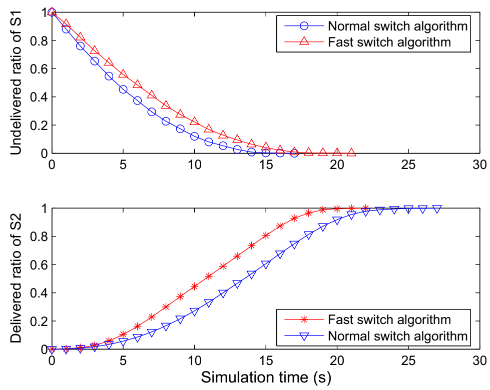

Figure 9. Ratio track in a dynamic network with 1000 nodes.

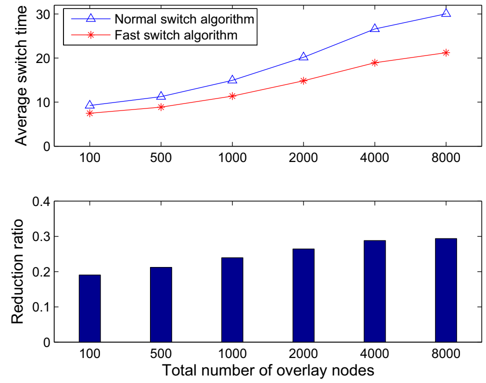

Figure 11. Avg switch time and its reduction ratio in dynamic environments.

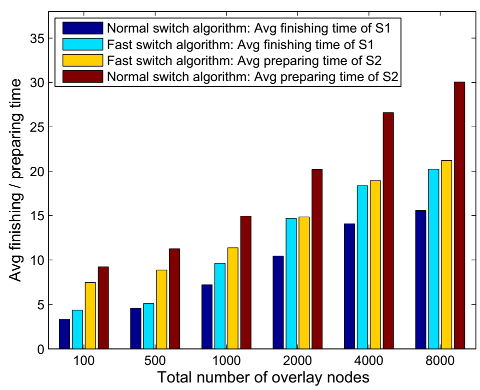

Figure 10. Avg finishing time of S_1 and preparing time of S_2 in dynamic environments.

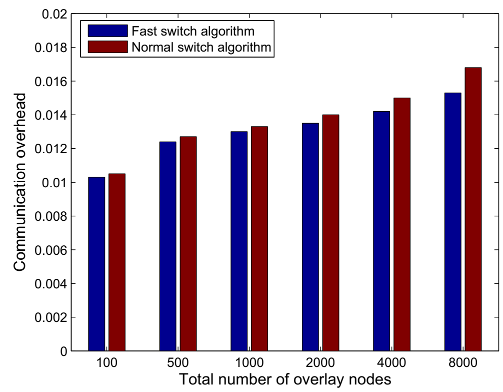

Figure 12. Communication overhead in dynamic environments.

References

[1] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. *Proc. ACM SOSP*, pages 298–313, 2003.

[2] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, 2002.

[3] T. Cormen, C. Leiserson, and R. Rivest. Introduction to algorithms. *MIT Press Cambridge, MA, USA*, 1990.

[4] A. Ganesh, A. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans on Computers*, 52(2):139–149, 2003.

[5] J. Li. PeerStreaming: An On-Demand Peer-to-Peer Media Streaming Solution Based On A Receiver-Driven Streaming Protocol. *IEEE 7th Workshop on Multimedia Signal Processing, 2005*, pages 1–4, 2005.

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

[7] D. Tran, K. Hua, and T. Do. ZIGZAG: an efficient peer-to-peer scheme for media streaming. *Proc. INFOCOM 2003*.

[8] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. *Proc. ICDCS 2002*, pages 363–371, 2002.

[9] M. Zhang, J. Luo, L. Zhao, and S. Yang. A peer-to-peer network for live media streaming using a push-pull approach. *Proc. ACM Multimedia*, pages 287–290, 2005.

[10] X. Zhang, J. Liu, B. Li, and T. Yum. CoolStreaming/DONet: A Data-driven Overlay Network for Peer-to-Peer Live Media Streaming. *Proc. IEEE Infocom*, 2005.

[11] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. *Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination*. ACM Press New York, NY, USA, 2001.