TreeClimber: A Network-Driven Push-Pull Hybrid Scheme for Peer-to-Peer Video Live Streaming

Xiangyang Zhang and Hossam Hassanein
School of Computing, Queen's University
Kingston, Ontario, Canada
Email: xiang, hossam@cs.queensu.ca

***Abstract*—Several *push-pull* hybrid peer-to-peer (P2P) video live streaming schemes introduce *data-driven* push trees into the pull scheme and have a short playback delay while being robust against peer churn, but the substrate Internet utilization remains unaddressed. In this paper, we propose TreeClimber, a *network-driven* robust push-pull scheme for large-scale P2P video live streaming applications. The scheme first constructs a neighbourhood overlay with short edges and a small diameter, then uses a robust distributed algorithm to build a short tree on the overlay. We have implemented a discrete-event simulator to examine TreeClimber and a comparative data-driven scheme. Results show that TreeClimber achieves high network utilization while having better or similar playback delay and robustness.**

I. Introduction

P2P video live streaming schemes can be classified into *push* and *pull* schemes [1]. In push schemes, peers self-organize into multicast trees and try to optimize a cost function. Push schemes have a short delay, but loops may occur and packets may be duplicated or lost in the presence of peer churn. Pull schemes are also called *swarming*-based schemes because they use swarming techniques, or *data-driven* schemes because pull operations are driven by availability of chunks in neighbours. In pull schemes, the video is split into fixed-length chunks. Peers exchange buffer maps that describe which chunks they have and pull missing chunks from one another. Since a peer can obtain a chunk from any neighbour, pull schemes are robust against peer churn. However, pull schemes have a long playback delay due to the buffer map advertisement interval and arbitrary chunk arrival order. References [2]–[4] introduce *push trees* into pull schemes. In addition to pulling chunks that neighbours have, peers also request neighbours to push future chunks. The tree-building algorithms are also *data-driven*, i.e., a peer chooses its parent according to the buffer maps of neighbours at the current time and/or the history it receives from neighbours. Data-driven tree-building algorithms preserve the robustness of pull schemes because an orphaned peer can immediately pick another neighbour as parent and pull lost chunks. Push operations reduce the playback delay. However, substrate Internet utilization remains unaddressed.

In this paper, we propose TreeClimber, a *network-driven* push-pull scheme for large-scale P2P video live streaming applications which, compared with data-driven push-pull schemes, achieves high network utilization while having better or similar playback delay and robustness. The scheme first constructs a neighbourhood overlay that have short edges and a small diameter, then uses a robust Distance-Vector (DV)-style routing protocol to build a short tree on the overlay. Peers have small path costs to the server due to short overlay edges and less path hops; the latter also leads to a smaller playback delay. Our contributions include (1) a distributed robust tree-building algorithm that maintains a short tree in the presence of peer churn on an arbitrary-sized overlay, and (2) a network-driven approach of push-pull schemes that utilizes the substrate Internet efficiently in addition to a small playback delay and high robustness. The rest of this paper is organized as follows. Section II discusses related work. Section III presents the scheme. Section IV evaluates the scheme and comparative schemes. Section V concludes.

II. Related Work

Existing push-pull schemes [2]–[5] have similar pull mechanism. They mainly differ in push tree and neighbourhood overlay construction. In [2], peers consider the volume of traffic they have received from neighbours during the last interval to select parents; peers must have synchronized clocks. In [3], peers consider neighbours' proceeding positions in the video to select parents. Reference [4] exploits the phenomenon that "older" peers have a higher probability to stay longer and use older peers to build a "treebone". The push mechanism (and the simulator) in [5] is the same as [2] with one modification—a peer accepts children up to its upload bandwidth.

The neighbourhood overlay is constructed to be a random graph in [3], [4]. In [2], a new peer selects some neighbours randomly and some by round trip time (RTT) from its membership table. This measure only marginally improves network utilization; because of the small membership table size in contrast to the population, the probability of nearby peers being selected is small. In [5], a new peer i selects neighbours from its membership table according to a function, which is the ratio of the residual bandwidth of a neighbour over the delay to the server directly or via the neighbour, to reduce the playback delay. How peers adapt to peer churn to keep the function minimized is not given in [5].

Pull schemes [6], [7] build multicast trees on an unstructured overlay using network metrics. In [6], peers start with a random tree and swap parents to satisfy peers' load target to improve throughput and reduce delays. In [7], peers run a DV-style protocol to construct a unicast routing table of the overlay and form trees using reverse unicast paths. Both schemes avoid

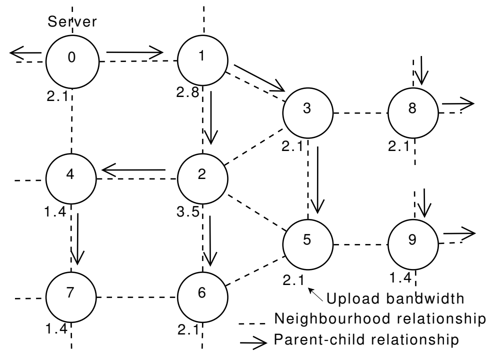


Fig. 1. Neighbourhood Overlay and Push Tree

loops by appending path information to routing updates to neighbours. Recovery of lost packets is not addressed.

III. TreeClimber

A peer has four components. The overlay module constructs the neighbourhood overlay. Many schemes that find nearby peers on the Internet exist, and the distance can be defined in many ways. TreeClimber is capable of functioning with any scheme. In this paper, we use the scheme in [8]. Each peer has a virtual network coordinate, obtained by measuring RTTs to a set of landmark hosts; the distance between peers can be calculated using their coordinates. Interested readers are referred to [8] for details. The subscription module builds a short tree on the overlay. The pull module pulls missing chunks close to playback deadlines. The buffer module stores recently received chunks and feeds the media player. The video server is a peer that neither subscribes nor pulls chunks from other peers. The *tracker* provides membership tables to peers.

A. Neighbourhood Overlay Construction

A new peer browses a website to obtain the IP address of the server and tracker. The new peer reports its network coordinate to and obtains a membership table of size M_p from the tracker. A peer contacts the tracker for more peers if its table size drops to a certain level; typically, it contacts the tracker only once. The new peer randomly selects neighbours from the table, and try to maintain $1.5u_p$ (minimum 4) neighbours, where u_p is the peer's upload bandwidth. Bandwidth is expressed in units of the streaming rate in this paper.

The tracker maintains a repository of size M_t. The repository is large enough to be a representative sample of the whole population, i.e., given a peer p, the distribution of peers in the repository is the same as in the population with respect to the distance to peer p. Upon receiving a membership table request from a peer p, the tracker uses a greedy algorithm to select peers with the shortest distance to peer p and possibly add p to its repository. The tracker also selects a small number of peers randomly. The randomness helps to prevent system partition and reduce the diameter of the overlay. On average, a peer's neighbours are among the top $\frac{M_p}{M_t}$ of the population with respect to distance to the peer. The Meridian project [9] measured delays between 2500×2500 hosts on the Internet.

```
1: $neighbours.sort\_by\_hops()$
2: $h \leftarrow neighbours.front().hops$
3: **for** $i$ in $neighbours$ **do**
4:     **if** $i.hops \neq h$ **then break**
5:     **if** $i$ is parent **then return** $i$
6:     **if** $i$ is not child **then** $cp.insert(i)$
7: **return** $cp.at(rand() \% cp.size())$
```

Fig. 2. Parent Selection

```
1: **if** $s$ is child or $has\_spare\_bandwidth()$ **then**
2:     $send(s, ACCEPT)$
3: **else**
4:     $d \leftarrow calc\_disown\_target(s)$
5:     **if** $d \neq s$ **then**
6:         $send(s, ACCEPT)$
7:         $send(d, DISOWN)$
8:     **else** $send(s, REJECT)$
```

Fig. 3. Subscription Request Handling

The average delay is 77 ms for all the edges and 7.8 ms for the shortest 5% edges. A repository of several thousands of peers is representative enough and can be handled by a PC.

B. Pull Operations

Each chunk has a unique sequence number. Each peer maintains a sliding window a recently received chunks and advertise its buffer map to neighbours every interval of length T_{bma}. Peers only pull chunks from neighbours that they think will not arrive before deadlines. Every interval of length T_{pull}, a peer requests all the missing chunks in the emergency window, which consists of chunks approaching playback deadlines. The targets of requests are randomly chosen from neighbours. Upon receiving a pull request, a peer replies with the chunk if it has spare bandwidth; it rejects otherwise.

C. Push Tree Building and Maintenance

TreeClimber provides a heuristic algorithm to build a degree bounded shortest path tree (DBSPT). When peers have enough bandwidth, the resulting tree is a shortest path tree (SPT). The tree repairing mechanism in TreeClimber is as robust as data-driven schemes in the presence of peer churn—an orphaned peer can immediately pick another neighbour as parent and pull lost chunks. TreeClimber differs from a typical DV routing protocol in three aspects. First, peers exchange *tree-hop*s, the hops of the path from a peer to the server along the push tree, rather than overlay hops. In Fig. 1, peer 4's hops is 1 and tree-hops is 3. Second, the routing protocol itself has no explicit mechanisms, such as exchanging path information or poison-reverse, to avoid loops. TreeClimber keeps loop occurrence at a low level by three factors: the greedy algorithm in Fig. 2, a small T_{bma}, and a minimum of 4 neighbours peers have. Peers identify chunks by sequence numbers and stop them from circling in loops by sending a chunk to a child only once. Third, TreeClimber tolerates partitioning of the tree; the partitions are joined by pull operations.

A tree-hops field is piggy-backed on buffer map messages. Every interval of length T_{sw}, peers use the greedy algorithm in Fig. 2 to find parents to reduce tree-hops. Neighbours are ranked by their tree-hops, and the neighbours with the lowest tree-hops are candidate parents (lines 1–4). A peer selects its current parent to reduce switching if it is a candidate, or a random candidate otherwise (lines 5 and 7). A peer does not select its children (line 6). A child must subscribe to its parent every interval of length T_{sub} to keep its status as a child.

A peer accepts children up to a limit (lines 1–2 in Fig. 3). The bandwidth for push operations are guaranteed; pull operations use the residual bandwidth. If a peer receives subscription requests from more than one peer, it accepts the peer with the largest upload bandwidth. TreeClimber has a preemption mechanism (lines 4–8). Suppose a low bandwidth peer l sends a request to peer t earlier and becomes a child. Then peer t receives a request from a high bandwidth peer h but has no spare push bandwidth. Peer t will accept peer h and disown the child with the lowest bandwidth. In Fig. 1, if peer 4 arrives earlier and attaches to peer 0, peer 1 can preempt peer 4.

IV. Performance Evaluation

We have implemented a discrete event simulator to evaluate TreeClimber and a comparative data-driven push-pull scheme, called FastNeighbour. In FastNeighbour, a peer selects the neighbour with the most advanced proceeding position in the video as its parent. To avoid frequent switching, a peer switches to a new parent only when it has a margin of 2 seconds. Since an overlay with short edges can improve network utilization of any tree-building algorithm, we use same overlays for both schemes and measure a peer's cost to the server by its tree-hops to the server. The playback delay refers to the time elapsed from when a chunk appears at the server to the time the chunk is played at peers. The chunk delivery rate is the fraction of chunks that arrive at a peer before playback deadlines. The overhead caused by advertising buffer maps is determined by system parameters rather than by schemes; it is less than 1% in the simulation. Therefore, the overhead is measured by the chunk duplication rate, which is the fraction of duplicate chunks a peer receives or the fraction of peers at which a chunk arrives more than once.

A. Simulation Setting

We use the Doar-Leslie model of GT-ITM [10] to generate a sparse random graph with 3000 nodes, then add short edges. The revised graph serves as the tracker's repository to initialize peers' membership tables. The propagation delays of overlay edges are proportional to their costs with an average of 100 ms. The streaming rate is 512 Kb/s. The chunk size is 16 KB. Peers maintain a buffer of 480 chunks and advertise buffer map every 1 second. Peers check whether to pull missing chunks or switch parents every 2 seconds. Subscription interval is 10 seconds. To simulate asynchronous operations of peers, we start peers' interval timers with a random offset. Peers buffer 5 seconds of chunks before starting to play.

We use three upload bandwidth settings. The average bandwidth is 6.8, 4.9 and 2.8 when the resource coefficient (RC) is 1.0, 0.7, and 0.4, We use two scenario. In the static scenario, all peers join at time 0 as a "flash crowd", the server starts streaming at time 5, and the simulation ends at time 300. In the dynamic scenario, the peer churn rate is 10% of the population per minute according to [3]. From time 10, a peer is turned off every 0.2 second until 600 peers are turned off at time 130. Then every 0.2 second, a peer is turned off and a previously turned off peer is turned on. Statistics are collected from time 155 to 275 on peers that have never been turned off.

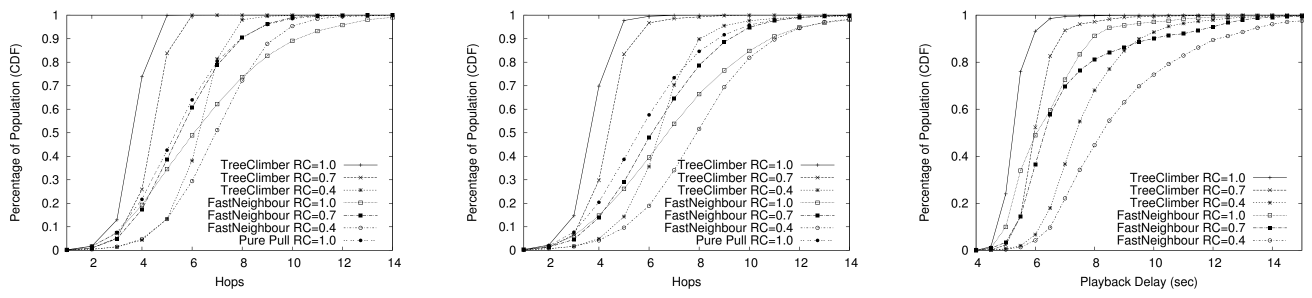
B. Simulation Results

1) Network Utilization: Fig. 4a shows that, in the static scenario, TreeClimber has fewer hop counts than FastNeighbour with all three bandwidth settings. When RC is 1.0, the average hop count is 3.66 for the SPT, 4.11 for TreeClimber, and 6.94 for FastNeighbour. The gap between TreeClimber and FastNeighbour is significant considering the system has only 3000 nodes and the SPT's height is 6. With smaller upload bandwidth and less neighbours, the gap becomes smaller because peers have fewer choices; both schemes increasingly choose the same neighbour as the parent for a peer because it is the only choice. Fig. 4b shows similar results in the dynamic scenario. Since the overlay is constantly changing, parameters of the overlay's SPT cannot be obtained, so we compare TreeClimber in the static and dynamic scenario to get a feel of whether TreeClimber can maintain a short tree in the presence peer churn. In the dynamic scenario, the overlays has 20% less nodes and, on average, 20% less edges; the difference of average hop counts is within 5%.

We also plot peers' hop counts in a pull scheme in Fig. 4. Interestingly, the pull scheme achieves fewer hop counts than FastNeighbour. We estimate that the reason is asynchronous operations of peers. The availability of a chunk sometimes propagates faster on a longer overlay path. In pull schemes, the path a chunk traverses is independent from other chunks. In data-driven push-pull schemes, once peers have decided on the parent–child relationships, future chunks propagate faster along the path even if it has more hops.

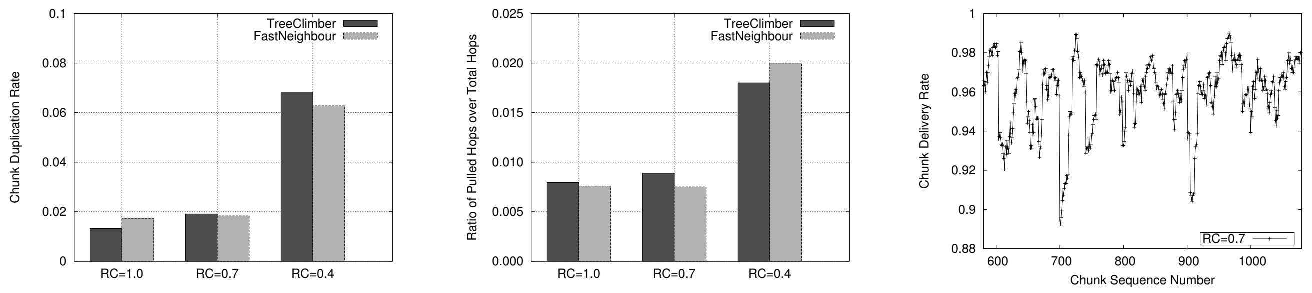
2) Playback Delay: The playback delay consists of the time a chunk takes from the video server to a peer and the time the chunk stays in the peer's buffer before being played. The delivery delay consists of the propagation delay of the edges and the queueing delays at peers on the path. We only plot the results in the dynamic scenario in Fig. 4c; the results in the static scenario are similar except having smaller values. TreeClimber has smaller playback delays than FastNeighbour in all bandwidth settings due to less hops to the server. The difference is not significant since both schemes buffer 5 seconds of chunks before starting to play. When RC is 1.0, the difference is 1 second.

3) Robustness against peer churn: TreeClimber and FastNeighbour have similar chunk delivery rates, which are 100% when RC is 1.0 and 0.7 and approximately 99.5% when RC is 0.4 in the dynamic scenario. Fig. 5a shows that the two



(a) Hops in Static Scenario (b) Hops in Dynamic Scenario (c) Playback Delay in Dynamic Scenario

Fig. 4. Hops to the Server and Playback Delay



(a) Chunk Duplication Rate (b) Fraction of Pulled Chunks (c) Chunk Delivery Rate in Push Only Mode

Fig. 5. Robustness and Contributions of Push and Pull Operations

schemes have similar chunk duplication rates. We turn off the pull function in TreeClimber, the chunk duplication rate drops from 1.91% to 0.28% when RC is 0.7. This result shows that TreeClimber keeps loop occurrence at a low level. By examining the event log, we find that duplications are mostly caused by peers pulling urgent chunks that are pushed by parents later. Fig. 5a and 5b shows that less upload bandwidth leads to more pull operations and duplications. When bandwidth is smaller, peers spend more time finding parents, hence more chunks are pulled, resulting in more duplicated chunks.

Although chunks are mostly pushed between peers, as shown in Fig. 5b, pull operations are necessary. Fig. 5c shows the delivery rate when the pull function is turned off in TreeClimber. The average is 96.0% when RC is 0.7. Aggressive pull operations increase the chunk delivery rate at the penalty of a higher chunk duplication rate. In the simulation, we enlarge the emergency window to allow more chunks to be pulled. When RC is 0.4, the chunk delivery rate increases from 99.5% to 99.9% and the chunk duplication rate increases from 6.9% to 19.1%.

V. Conclusion

Data-driven push-pull schemes are robust against peer churn while having a small playback delay, but the substrate Internet utilization is not addressed. We propose a network-driven push-pull scheme. Simulation results show that the scheme achieves high network utilization while having a small playback delay, high robustness, and low overhead.

References

[1] J. Liu, S. G. Rao, B. Li, and H. Zhang, "Opportunities and challenges of peer-to-peer internet video broadcast," *Proc. IEEE*, vol. 96, no. 1, pp. 11–24, 2008.

[2] M. Zhang, L. Zhao, Y. Tang, J. G. Luo, and S. Q. Yang, "Large-scale live media streaming over peer-to-peer networks through global Internet," in *Proc. ACM Workshop on Advances in Peer-to-Peer Multimedia Streaming*, 2005, pp. 21–28.

[3] S. Xie, B. Li, G. Y. Keung, and X. Zhang, "Coolstreaming: Design, theory and practice," *IEEE Transactions on Multimedia*, vol. 9, no. 8, pp. 1661–1671, 2007.

[4] F. Wang, Y. Xiong, J. Liu, B. Burnaby, and C. Beijing, "mTreebone: A hybrid tree/mesh overlay for application-layer live video multicast," in *Proc. ICDCS*, 2007, p. 49.

[5] A. Ouali, B. Kerherve, and B. Jaumard, "Revisiting Peering Strategies in Push-Pull Based P2P Streaming Systems," in *2009 11th IEEE International Symposium on Multimedia*. IEEE, 2009, pp. 350–357.

[6] V. Venkataraman and P. Francis, "Chunkyspread: Multi-tree unstructured peer-to-peer multicast," in *Proc. Int. Workshop on Peer-to-Peer Systems*, 2006.

[7] Y. Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1456–1471, 2002.

[8] T. Ng and H. Zhang, "Towards global network positioning," in *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2001, pp. 25–29.

[9] "http://www.cs.cornell.edu/people/egs/meridian/," Accessed Mar. 2010.

[10] "http://www.cc.gatech.edu/projects/gtitm/," Accessed Mar. 2010.