PATTERN-PUSH: A LOW-DELAY MESH-PUSH SCHEDULING FOR LIVE PEER-TO-PEER STREAMING

Guifeng Zheng[*] *S.-H. Gary Chan*[†] *Xiaonan Luo*[‡] *Ali C. Begen*[§]

[*]School of Software, Sun Yat-Sen University
Key Laboratory of Digital Life (Sun Yat-sen University), Ministry of Education, China
[†]Department of Computer Science & Engineering
The Hong Kong University of Science & Technology, Hong Kong China
[‡]Institute of Computer Application, Sun Yat-Sen University
Key Laboratory of Digital Life (Sun Yat-sen University), Ministry of Education, China
[§]Video & Content Platforms Research & Advanced Development
Cisco Systems, Inc., San Jose, CA 95134 USA

ABSTRACT

In live peer-to-peer (P2P) streaming, each peer (child) has a number of supplying parents whose packets have to be scheduled and delivered in time for continuous playback at the child. It is challenging to develop a scheduling algorithm that achieves low delay given heterogeneous bandwidth, propagation delays and available content in all the parents. This paper proposes a novel, simple and effective scheduling scheme called Pattern-Push. As compared to the traditional mesh-pull, pattern-push does not require continuous buffermap advertisements from the parents, and operates on the packet level instead of the larger segment level. In pattern-push, each parent pushes its packets according to a pattern as indicated by a starting packet ID and a cycle bitmap. Pattern-push requires only minimal feedback from the child, as the pattern only needs to be changed when the child detects a marked change in network conditions or its parents. Simulation results show that pattern-push achieves a significantly lower delay and overhead as compared with both traditional and recent scheduling algorithms proposed in the literature.

***Index Terms*—** Live peer-to-peer streaming, mesh-based overlay, packet scheduling

1. INTRODUCTION

In recent years, there have been many live peer-to-peer (P2P) streaming systems implemented and deployed in the Internet. In order to improve the robustness against node churns and meet a certain streaming bandwidth requirement, P2P streaming networks are usually built on a mesh [1–4]. In a mesh overlay, each peer connects to some other peers as its neighbors known as supply *parents*. By retrieving packets from its parents given the mutual end-to-end bandwidth, a child can aggregate and assemble these packets in a timely and orderly manner to achieve stream continuity. The mechanism that determines which parents should deliver which packets is called a scheduling algorithm.

For live applications such as P2P live TV, the overall delay from the source to the peer is an important consideration. Such delay depends on scheduling (scheduling delay) due to the heterogeneous bandwidth, available content and propagation delays between the parents and the child. We will focus in this paper on reducing such delay through the design of scheduling algorithm.

Given a mesh overlay and a set of parents, the scheduling problem is how to schedule the packets in each parent to minimize the playback delay of the child with stream continuity, given the heterogeneous bandwidth, propagation delay and contents of the parents. Traditionally, scheduling algorithm is pull-based, where parents first send their buffer contents in bitmap to the child. Based on that, a child makes decision on which particular packets to "pull" from the parents according to, for example, the rarest-first algorithm. Such an algorithm may not guarantee in-order delivery (and hence continuity). Furthermore, to reduce control overhead in exchanging buffer bitmaps, the parents usually advertise their contents only when their bitmap is reasonably large. This increases the scheduling delay.

We propose and study in this paper a novel, simple and effective packet scheduling algorithm called Pattern-Push, where parents push packets according to a pattern as indicated by a starting packet ID and a cycle bit pattern. The pattern is designed so as to achieve a smooth playback with low delay. As long as the network conditions do not change much, such pattern does not need to be adjusted, which helps reduce the feedback overhead. To calculate the pattern in pattern-push (such as upon peer arrival or a significant change in network conditions), the parents first push their latest packets to the child. Based on the information, the child computes new patterns and feeds back to its parents. With the child feedback, a parent starts from the new starting packet and pushes packets according to the new bitmap cycle by cycle to the child.

We address how to derive the pattern for packet scheduling in pattern-push, which achieves the following:

- *In-order and maximal packet arrival:* The pattern is designed so that packets can arrive contiguously at the child, despite the heterogeneous bandwidth, propagation delay and available contents at the parents, as long as each parent starts at the

This work was supported, in part, by the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation (SVCF08/09.EG01), the Hong Kong Innovation Technology Fund (ITS/013/08), and the National Science Fund of China (60525213, U0735001), the 863 Program of China (2007AA01Z236), the Start Plan for Young Teachers of Project 985 of China (3171910), the National Project of Scientific and Technical Supporting Programs (2007BAH13B01).

Corresponding author: Xiaonan Luo (lnslxn@mail.sysu.edu.cn).

specified packet and pushes according to the pattern in a cycle (assuming no loss). A child would not get out-of-order packets, and if there is no loss, there would be no holes in the buffer, resulting in a maximal delivery.

- *Absence of packet redundancy:* The pattern is designed so that each packet is pushed by only one parent, even though the parents push packets in a distributed manner. Therefore, bandwidth is used efficiently.
- *Hole recovery at child:* Due to occasional packet loss or fluctuating network conditions, packets may not arrive at a child by the playback deadline, i.e., the child experiences holes. Pattern-push fixes the holes with backup bandwidth from some parents (which may be the child's streaming parents).

In this work, we present how to design the pattern. Through NS-2 simulations, we show that pattern-push achieves a substantially lower delay than other traditional or recent schemes. Pattern-push is simple to implement. Our study shows that pattern-push achieves low overhead, and efficiently utilizes the end-to-end bandwidth.

We briefly discuss related work below. Content scheduling may be roughly divided into four categories: tree-push, mesh-pull, hybrid pull-push, and mesh-push.

In tree-based push (such as SplitStream [4]), the peers form one or more application-layer multicast trees. Pattern-push is not based on this multi-tree approach, and hence each peer may have different number of parents depending on the end-to-end bandwidth. It also does not require a certain minimum end-to-end bandwidth, and the pattern can be *optimized* (as opposed to be fixed in tree-push) according to the end-to-end bandwidth between a child and its parents. Its mesh construction is separated from the scheduling, making it more flexible in overlay construction and optimization. The pattern can also be locally and dynamically adapted in a child according to the network conditions.

There has been much work on mesh-pull, which basically emulates BitTorrent with additional timing deadlines [3, 5]. In terms of scheduling algorithms, Chainsaw [5] uses a purely random strategy to decide what to request from neighbors, while DONet [3] employ a rarest-first strategy by first scheduling the parents with the most surplus bandwidth and available time. As compared with mesh-pull, pattern-push does not require buffer bitmap or explicit pull requests on per-packet basis. A parent actively pushes its packets to child, and feedback is required only when network conditions have substantially changed. Therefore, it achieves a lower delay and control overhead.

Recently, pull-push has been proposed, which makes use of tree-push for data delivery and mesh-pull for missing packets [6]. Though it is more robust to node dynamics, the approach shares similar weaknesses with the tree-push approach as discussed before. Due to the exchange of buffer bitmap and pull mechanism, it has higher delay than the tree-push approach. As compared with this approach, pattern-push pushes packets according to some calculated pattern that is adaptive to heterogeneous and dynamic bandwidth. There is no bitmap exchange and minimal feedback, resulting in lower delay and better bandwidth utilization.

In mesh-push, packets are pushed without explicit request from a child. A nice example is R^2, which uses network coding to randomly code segments and push downstream [7]. A child can decode all its packets after collecting a sufficient number of independently coded blocks. R^2 is shown to achieve lower delay than mesh-pull. As compared with R^2, pattern-push is much simpler to implement and has lower processing overhead. Due to its smaller scheduling block, pattern-push has a lower packet waiting time, and hence, delay.

The remainder of this paper is organized as follows. We present in Section 2 our pattern-based scheduling algorithm and an example on how to derive the pattern. In Section 3, we describe our simulation environments and evaluate pattern-push by comparing it with other schemes. We conclude in Section 4.

2. PATTERN-PUSH SCHEDULING

Given the bandwidth and propagation delay between a child and its parents, a good scheduling should efficiently utilize the bandwidth and balance the propagation delay so that packets arrive in-order for smooth playback. Consequently, minimal scheduling delay is achieved. This is the design principle of pattern-push.

There are several features of pattern-push to achieve low delay:

- *Active push:* In pattern-push, once a parent receives a packet, it actively forwards the packet to its child according to a preset pattern of its child. Therefore, as compared with other pull-based approaches with bitmap exchange, the buffering delay, and hence the overall packet delay, is significantly reduced.
- *Pattern-based:* In order to indicate which packets are needed from a particular parent with minimal control traffic, pattern-push uses a pattern, which is simply a short bitmap (of a cycle) and a starting packet ID. With the parents following the pattern, duplicate packets are eliminated. The pattern is designed in such a way that the packets arrive contiguously (in order), so that smooth playback is guaranteed and reassembly delay is minimized.
- *Recovery parents:* Packet loss is inevitable due to fluctuating network conditions and node dynamics. To recover the lost packets, a child needs backup parents (which may or may not be its streaming parents). Upon detecting holes (packets that are late for their arrival or miss their playback deadlines), the child requests them from the backup parents. To know which backup parent has the missing packets, a backup parent piggybacks its buffer bitmap on its normally delivered packets. Note that due to our low delay, the buffer bitmap is small (several bits) as it only needs to indicate a short range of available packets.

The main procedures of pattern-push are shown in Fig. 1, which are elaborated as follows:

1. *Start up*: Each parent pushes its latest received packet, say z, to the child (at time X). The child receives the packet at time $A(z)$.
2. *Schedule point*: A new pattern is generated whenever there is a major change in the network conditions, e.g., when the parents or the end-to-end bandwidths have changed. At the scheduling point (at time T), the scheduling algorithm is calculated.
3. *Packet delivery*: After a new pattern (in terms of cycle bitmap and starting packet ID) is generated, the child feeds it back to the respective parent. Upon receiving the pattern (at time Y), the parent pushes its packets according to it.
4. *Hole fixing/recovery*: A child keeps some backup parents, which are the nodes with high bandwidth and newer content. Note that backup parents may also be streaming parents with some of their bandwidths reserved for packet recovery. They

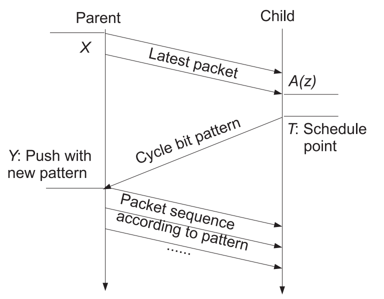


Fig. 1. Procedure of pattern-push for a child.

piggyback their pushed packets with several bits to indicate their packet availability. Upon detecting a hole, the child sends requests to its backup parents for recovery.

We discuss now how to derive the bit pattern. Consider that the end-to-end bandwidth is quantized to some unit of bitrate (say, 10 kbps), so that all parent bandwidths are quantized into some integral units. As mentioned, the scheduling pattern of a parent run at the scheduling point is to ensure timely and contiguous packet arrival at the child given the bandwidth and delay. This pattern consists of a starting packet ID and cycle bitmap. For example, if a parent receives a pattern with the starting packet 7 and bit pattern 101, it will push packets $7, 9, 10, 12$, and so on.

The length of cycle bitmap is the same for all parents, and indicates the total number of packets the child receives in a cycle. Due to its limited upstream bandwidth, a parent can only push a certain limited number of packets in a cycle. This is given by the number of assigned bits in the cycle for the parent. To save space, the bit patterns may be encoded using run-length encoding.

There are some important symbols used in our derivations. The video is of bandwidth R bits/s, and the packet size is S bits. The normalization bit rate is u bits/s, and hence, $R = mu$, for some $m \in \mathbb{Z}^+$. For a child, let $\mathcal{P}$ be the set of parents that push source packets to it. For each parent $j \in \mathcal{P}$, let its propagation delay to the child be D_j seconds, and its end-to-end bandwidth to serve the child be B_j bits/s.

To ensure timely and contiguous packet delivery, there are two steps to generate the cycle bitmap for each parent:

1. *Calculation of the cycle length L:*
   We first calculate the cycle length of a pattern denoted as L. For a child, let $\widehat{B}_j$ be the normalized bandwidth of its parent j ($j \in \mathcal{P}$) given by $\widehat{B}_j = \lfloor B_j/u \rfloor$. Naturally, for stream continuity, we need $\sum_{j\in\mathcal{P}} \widehat{B}_j \geq m$. Further, let n be the gcd (greatest common divisor) for all the parent bandwidths of the child, as given by $n = \gcd_{j\in\mathcal{P}} \widehat{B}_j$. Let b_j ($j \in \mathcal{P}$) be some integer proportional to the parent bandwidth as given by $b_j = \widehat{B}_j/n$. The cycle length is then obtained as $L = \sum_{j\in\mathcal{P}} b_j$, and b_j is the number of packets parent j pushes in the cycle.
2. *Pattern assignment:*
   Given L, the child then assigns bitmap to each parent. Under normal stable network conditions and low packet loss, the parents receive their packets continuously. A child may, therefore, estimate the arrival time of a certain packet as shown Fig. 1. It can then sort the arrival times of the packets and assign the starting packet ID and bitmap to them accordingly.

By referring to Fig. 1, the complete procedure of the pattern assignment to a parent is detailed as follows:

(a) Estimation of packet arrival time for parent j:
For each parent $j \in \mathcal{P}$, the child first estimates the latest packet ID l received by the parent at point A (See Fig. 1). Noting the time between X and Y is $T - A(z) + 2D_j + S/B_j$, the latest packet index the parent should have received at point Y is

$$l = z + \left\lfloor (T - A(z) + 2D_j + \frac{S}{B_j})\frac{R}{S} \right\rfloor. \quad (1)$$

Starting from l, the child chooses b_j packets, with the arrival time of the k^{th} packet at the child estimated as $(k = 0, 1, \ldots, b_j - 1)$

$$A(l + k) = T + 2D_j + (k + 1)\frac{S}{B_j}. \quad (2)$$

(b) Ordering of arrival time and sequence adjustment:
After computing the arrival times of the packets from all the parents (a total of L packets), the child sorts them in increasing order. Label the sorted packet sequence $q(i)$ and the corresponding parent sequence $f(i)$, $i = 0, 1, \ldots, L - 1$. The sequence $q(i)$ is the possible packet order that can be pushed from respective parent.
The child then adjusts packet sequence $q(i)$ to eliminate overlaps and gaps while guaranteeing arrival continuity and low delay. By observing that the entries of $q(i)$ are the latest packets from the parents, it achieves that by adjusting some packet ID to be lower than the entries as they are cached packets, and hence, available.
To adjust $q(i)$, simply define $H = \min_i(q(i) - i)$, and update $q(i)$ to $\bar{q}(i) = i + H \leq q(i)$.

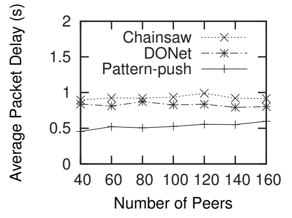
(c) Bitmap generation:
For the cycle bitmap, set $(\bar{q}(i) - \bar{q}(0))^{th}$ bit for parent $f(i)$ to 1, $0 \leq i \leq L - 1$, with the starting packet ID for all the parents $\bar{q}(0)$. Note that the first non-zero bit of the bitmaps are different, so that the first packets to push for all parents are different even they have the same starting packet ID.

The algorithm can be applied to all peers in the overlay. For example, the first hop peers get stream from the source node directly. The pattern cycle bitmaps for these peers are all 1, i.e., the source continuously pushes packets to them.

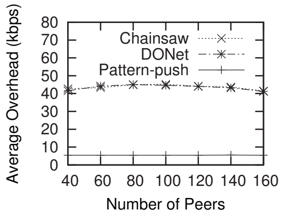
To take into account of delay jitter and peer churn, one may use some averaging mechanism (e.g., the exponential averaging) on the historical packet arrival statistics to calculate the propagation delay and bandwidth. The patterns would be updated when the network change exceeds a certain threshold, e.g., 20%.

3. ILLUSTRATIVE SIMULATION RESULTS

In this section, we present illustrative simulation results on pattern-push and its comparisons with some recently studied scheduling algorithms. We use NS-2 to evaluate the performance of our system, and compare it with two recent schemes, DONet [3] and Chainsaw [5]. They are both mesh-pull, which dynamically chooses neighbors, exchange buffermap and request packet periodically. We

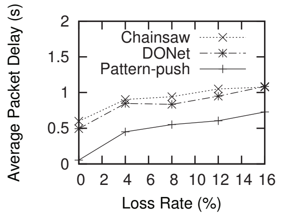


(a) Average packet delay.

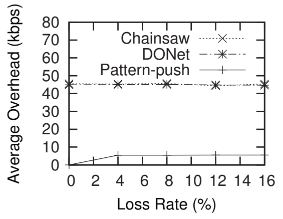


(b) Control message overhead.

Fig. 2. Performance comparison with respect to the number of peers.



(a) Average packet delay.



(b) Control message overhead.

Fig. 3. Performance comparison with respect to loss rate.

first discuss the simulation environment and metrics, followed by some illustrative results.

In NS-2, we randomly attach a certain number of agents to the BRITE nodes, and set one of them as the media source. We use 1000 nodes as the underlying topologies. Peers have an outbound bandwidth of 1000-2500 kbps, and the streaming rate is 500 kbps with packet size of 1000 bytes. The time intervals for buffermap exchange or packet request (for mesh-pull only) are chosen according to [6], which are shown to achieve a low delay. Unless otherwise stated, the baseline parameters are 100 peers, 5 parents and 8% packet loss rate.

The performance metrics we are interested in are:

- *Packet arrival delay*: Let T_p be the arrival time of a packet at its corresponding parent according to the delivery pattern, and T_c be the arrival time of the packet at the child including retransmission from the backup parent. The packet arrival delay is defined as $T_c - T_p$. Obviously, a good scheduling should achieve low packet arrival delay. We are interested in its average value for all peers in our study.
- *Control overhead*: The overhead is the total bandwidth used for control messaging. We are interested in its average value for all peers.

We show in Fig. 2(a) the average packet delay versus number of peers for Chainsaw, DONet and pattern-push methods. Clearly, the average packet delay is not affected by on-line peers, because we measure the one-hop delay only. The average packet delay for pattern-push is substantially lower, with about half of those for mesh-pull when the number of peers increases. This is because in mesh-pull, round-trip time is continuously incurred to exchange buffermap and request packets. In mesh-pull, when a packet arrives, it stays in buffer until buffer map containing it is sent out and request on it is received. In contrast, in pattern-push, the packets are actively pushed out as they arrive.

We plot in Fig. 2(b) the control overhead versus the number of peers. The control overhead does not increase as the number of peers increases, showing that the overhead is independent of the number of peers. From the plot we see the scalability of our scheme. In addition, pattern-push achieves a significantly lower control message overhead than pull-based ones, because it does not require periodic exchange of buffermap and packet requests.

We show in Fig. 3(a) the average packet delay versus the packet loss rate. As the loss rate increases, delays increase, because the recovery process makes the buffering delay higher. Pattern-push achieves much lower delay than the pull-based methods, especially when the loss rate is low.

We show in Fig. 3(b) the control overhead versus the packet loss rate. In all schemes, the overhead does not depend sensitively on loss rate. We can see that pattern-push achieves significantly lower overhead because it does not require periodic message exchange.

4. CONCLUSIONS

We present a pattern-push scheme that achieves low-delay scheduling for live P2P streaming. Each parent pushes its packets according to a pattern as indicated by a starting packet ID and a cycle bitmap. As the pattern only needs to be changed when there is a marked change in network conditions or its parents, its overhead is low. The pattern can be computed efficiently so that packets arrive at a child in a timely manner. Backup parents are used to recover the missing packets. We have conducted simulation study on pattern-push with NS-2. The results show that pattern-push achieves much better performance as compared to mesh-pull schemes, in terms of packet arrival delay and control message overhead.

5. REFERENCES

[1] Dongni Ren, Y.-T. Hillman Li, and S.-H. Gary Chan, "On reducing mesh delay for peer-to-peer live streaming," in *IEEE INFOCOM*, Phoenix, Arizona, Apr. 2008, IEEE.

[2] Xing Jin, Kan-Leung Cheng, and S.-H. Gary Chan, "SIM: Scalable island multicast for peer-to-peer media streaming," in *Proc. IEEE International Conference on Multimedia Expo (ICME)*, Toronto, Canada, July 2006, pp. 913–916.

[3] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: A data-driven overlay network for live media streaming," in *Proceedings of IEEE INFOCOM*, Miami, FL, USA, Mar. 2005, pp. 2102–2111.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-bandwidth multicast in cooperative environments," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, The Sagamore, Bolton Landing (Lake George), New York, Oct. 2003, pp. 298–313.

[5] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr, "Chainsaw: Eliminating trees from overlay multicast," in *The Fourth International Workshop on Peer-to-Peer Systems*, Feb. 2005.

[6] Meng Zhang, Qian Zhang, Lifeng Sun, and Shiqiang Yang, "Understanding the power of pull-based streaming protocol: Can we do better?," *Selected Areas in Communications, IEEE Journal on*, vol. 25, no. 9, pp. 1678–1694, December 2007.

[7] Mea Wang and Baochun Li, "R2: Random push with random network coding in live peer-to-peer streaming," *Selected Areas in Communications, IEEE Journal on*, vol. 25, no. 9, pp. 1655–1666, December 2007.