Pull-Patching: A Combination of Multicast and Adaptive Segmented HTTP Streaming

Espen Jacobsen, Carsten Griwodz, Pål Halvorsen
Department of Informatics, University of Oslo & Simula Research Laboratory
{espenjac, griff, paalh}@ifi.uio.no

ABSTRACT

Multicast delivery for video streaming gains credibility with the introduction of commercial IPTV. We therefore revisit *patching*, a video-on-demand idea from the 1990s. We have built PULL-PATCHING, an approach that combines the patching ideas with *adaptive segmented HTTP streaming*, a unicast technique that is used by most commercial providers of large-scale, true video-on-demand in the Internet today. The prototype is tested in a combined Internet and lab environment where we show the influence of practical factors like packet loss, delay and limited resource availability, and identify several details that require further study.

Categories and Subject Descriptors

H.5.1 [**Multimedia Information Systems**]: Video

General Terms

Design, Experimentation, Performance

Keywords

Multicast streaming, Patching, Segmented HTTP streaming

1. INTRODUCTION

Video streaming services over the Internet, both live and on-demand, are rapidly increasing in numbers. Today, many providers use either a combination of CDNs (like Akamai and L3) and adaptive segmented HTTP-based technology provided by companies like Move Networks, Microsoft and Apple or P2P-based solutions like PPLive, TVU player and Sopcast. In addition to this, ISPs use IP multicast to provide live TV to customers in their own networks under labels like *IPTV*, *triple play* or *quadruple play*.

The introduction of multicast for commercial streaming services means that the potential of well-researched multicast-based stream scheduling solutions can now be realized. Techniques like patching [6] (stream tapping [2]) and hierarchical multicast stream merging (HMSM [4]) offer true video-on-demand capabilities in ways that only reduce resource consumption. We have therefore implemented PULL-PATCHING, which targets dissemination of popular video content both in live and on-demand scenarios, to demonstrate that this can be achieved by backward-compatible extensions of the HTTP-streaming solutions that are in commercial use today.

PULL-PATCHING overcomes the weaknesses that all these early multicast streaming solutions (with the partial exception of receiver-driven layered multicast (RLM) [9]) have in common. They assume the channel model of TV broadcast, rely on making all decisions at the sending side and deliver all data in a push-based manner. Even the segmented patching techniques [3, 8] that adapt to bandwidth limitations of the patch server assume perfect channels. In contrast, PULL-PATCHING, like the HTTP streaming systems that are used commercially today, is built for imperfect network conditions. The design acknowledges that clients have the best information about current network conditions, and therefore lets the client react to challenges like delay, jitter, packet loss, and limited and oscillating bandwidth. Thus, the control of choosing video quality based on resource availability (adaptation) all the way out to the receiving device, the decision to terminate patch streams, and the ability to repair lost packets is managed by the client.

PULL-PATCHING combines the adaptive segmented HTTP-based streaming solution presented in [7] with patching and simulcast. Multicast is used to stream highly popular content at several bit rates (and qualities). Clients join multicast streams and add video segments that they retrieve using HTTP. These *pulled* segments comprise the initial patch stream as well as all data that is required to repair packet loss and corruption in the multicast stream. The multicast server is informed about client arrivals when the client retrieves the information that is required to join a multicast stream. Incidentally, this still gives the server the required information to estimate stream popularity that is needed to optimize the time between restarts of multicast streams in patching (e.g., λ-patching [5]). For content that is too unpopular to warrant multicast streaming, clients fall transparently back to HTTP streaming. The same applies to clients whose multicast reception is denied for other reasons, such as outdated DSLAMs or cable modems, and older clients.

In our experiments, using HTTP streaming from multiple sites in the Internet and delivering the multicast streams from a lab environment, we show that our initial prototype works. We also present how the system reacts to rate limitations, packet loss, delays and jitter.

2. PULL-PATCHING

Segmented HTTP streaming solutions have been claimed to be scalable and able to support millions of concurrent users [1]. However, this is paid with far from perfect video quality and a huge resource consumption. Pull-Patching combines our adaptive segmented HTTP streaming solution [7] with patching. Highly popular content is streamed from dedicated multicast servers, which use clients' RTSP SETUP requests to optimize the time between multicast stream restarts. HTTP streaming is used for delivering patch streams, repair packet loss and corruption in multicast streams, and serving clients that do not request multicast. Thus, as shown in figure 1, there are three main components: the HTTP-based servers, the multicast server and the client.

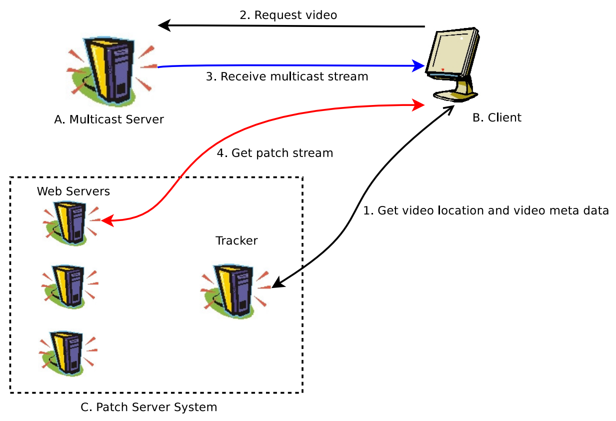


Figure 1: Pull-Patching **system overview.**

The HTTP streaming solution is based on the idea of downloading (2-second long[1]) segments of the video in a torrent-like manner. The client retrieves every segment's location from an index server (tracker) and downloads the segments from plain web-servers in playout order using HTTP GET requests (over TCP). For backward compatibility with the existing HTTP streaming systems, we code every video in several qualities using H.264 advanced video coding (AVC). The advantages of scalable video coding (SVC) are discussed in section 4. Each segment is coded independently of other segments. They can be combined arbitrarily and allow quality adaptation at every segment boundary, i.e., every 2 seconds. There are indications that quality decisions every 2 seconds achieve a better visual quality than more frequent changes [10]. During playout, the client chooses qualities according to resource availability.

The multicast server is implemented using the Live555 streaming framework. It starts multicast streams based on popularity. The video segments are pushed out to the network in a simulcast manner, using one channel for each quality. For each channel, the 2-second video segments are split into MTU sized packets, encapsulated using RTP and transmitted at playout speed.

In contrast to the original patching schemes where the servers had full control of amount and times of data sent, our clients control data retrieval except for the frequency of multicast re-starts. A client joins one multicast stream at a time and switches between multicast streams for quality adaption according to resource availability. If the client joins after the start of the multicast, the multicast data is stored in a cyclic buffer, and the initial missing data is retrieved as a patch stream using the HTTP patching servers. Similarly, if a packet is lost or damaged during the delivery of the multicast stream, a patch is used for repair of this segment. The client decides whether to download the complete segment using traditional HTTP GET requests, or only parts using HTTP GET RANGE requests.

[1]This is a size used by commercial systems [1] and is a tradeoff between coding efficiency, file size, search accuracy (see [7]) and video quality adaption granularity [10].

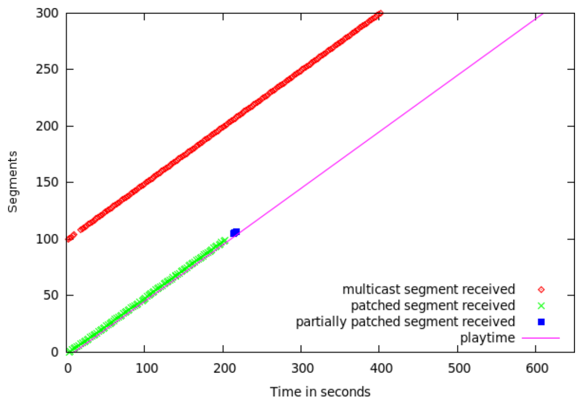
A major challenge is to schedule and prioritize between the multicast and patch streams according to video quality. There are a lot of different properties to take into account and many possible parameters to tune. Finding the optimal scheduling scheme is out of the scope of this paper. For this proof-of-concept, we have used a straight-forward approach. Clients that notice congestion divide the available bandwidth equally between the multicast and unicast streams depending on streaming rate of multicast stream and download time of patching segments.

3. EXPERIMENTS AND RESULTS

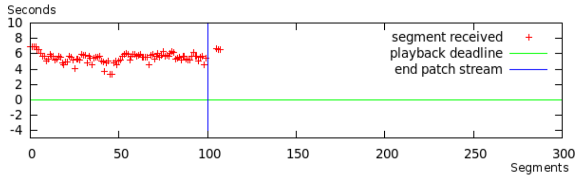
The video data is coded in independent, 2-second video segments in 4 different qualities with average bit rates of 738 Kbps, 1301 Kbps, 2171 Kbps and 3194 Kbps. The multicast server is located in a lab environment and the HTTP patch servers are distributed in the Internet (Oslo and Tromsø). To add network delay, jitter (and out-of-order delivery), loss, and to limit the available bandwidth, we used the network emulator *netem*. In the plotted results, the client arrives 100 segments (200 seconds) into the multicast stream and must use patching for the initial part of the video and for multicast repair. The system can adapt at every 2-second segment boundary, but to better show the quality development in these tests, we only check resource availability every 20th segment. Finally, in the current system configuration, we request patch segments as late as possible to reduce the bandwidth competition between streams.

In a perfect scenario with unlimited resources and no packet loss, jitter and delay, the Pull-Patching approach behaves just like the initial patching ideas. The multicast stream is stored without loss in the cyclic buffer, and the client starts viewing the patch stream retrieved over HTTP – both in full quality. A similar scenario can be observed in figure 2 where the available bandwidth is far more than requested for one multicast and one patch stream. Here, we see the typical patching pattern, i.e., the client starts receiving the multicast stream upon arrival (segment 100) and downloads the patch segments simultaneously. Figure 2(c) shows a lower quality right after segments 0 and 100. This is due to the policy of starting at a low quality for a quick start and adapt later as the client observes available resources. This decision is made independently for the multicast stream, resulting in the low quality around segments 100, and the patch stream, resulting in the low quality after segment 0. This can be improved as discussed in section 4.

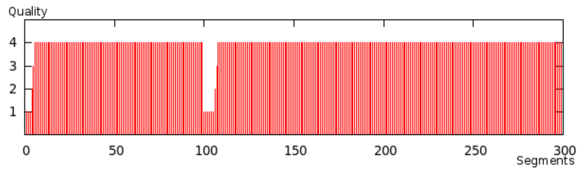
When bandwidth is limited, we do not have resources to retrieve both the multicast and patch streams at full quality simultaneously. Figures 3 and 4 show this for 4 and 2 Mbps, respectively. This makes the scheduler choose a lower quality as shown in figures 3(b) and 4(c). The bandwidth sharing between multicast and patch streams also results in dropped multicast packets that are visible as holes in the multicast line ("multicast segment received") and the corresponding repair stream to fill the gaps ("partially patched segment received") in figure 4(a). The 2 Mbps experiment supports only the lowest quality of 738 Kbps, and remaining resources



(a) Received segments

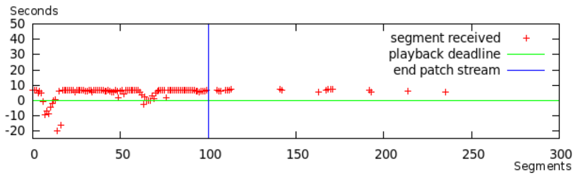


(b) Patch delivery time according to playback deadline

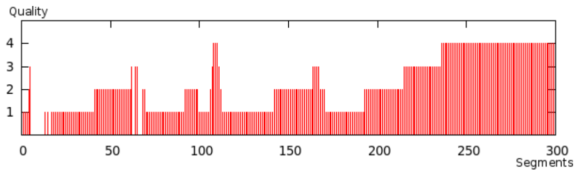


(c) Playout quality

Figure 2: Available bandwidth: 8 Mbps.



(a) Patch delivery time according to playback deadline

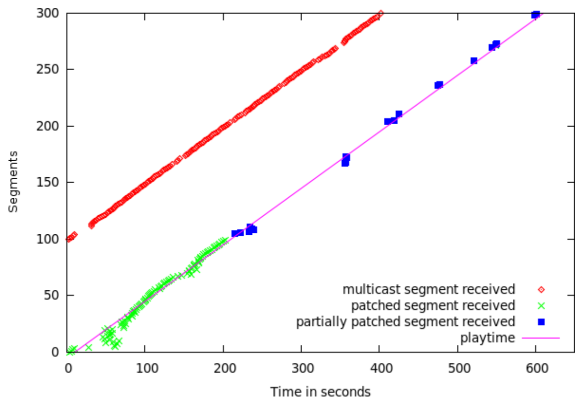


(b) Playout quality

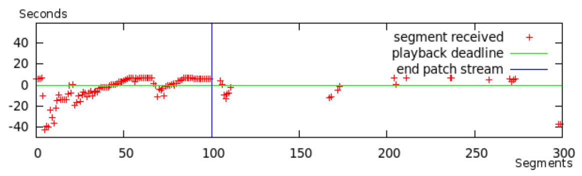
Figure 3: Available bandwidth: 4 Mbps.

are used for repair or to try a higher quality. Furthermore, as shown in figure 4(b), this competition for resources can sometimes delay segments until they are too late for playout, which results in an incomplete segment which is displayed with artifacts (see the missing pulses in figure 4(c)). This also means that when the initial patch stream is finished, and only the repair streams remain as competition for the multicast stream, the system increases the quality (figure 3(b) and 4(c)).

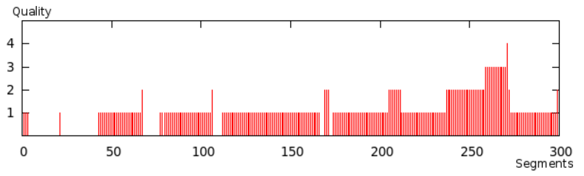
The system can also handle loss that occurs for other reasons than congestion caused by the system's own streams. Loss can influence the video playout directly. Figure 5 shows an 8 Mbps-bandwidth scenario with 1% packet loss. Again, figure 5(a) presents the segment delivery time according to



(a) Received segments

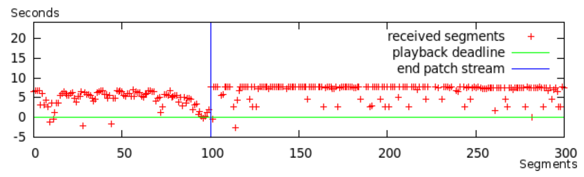


(b) Patch delivery time according to playback deadline

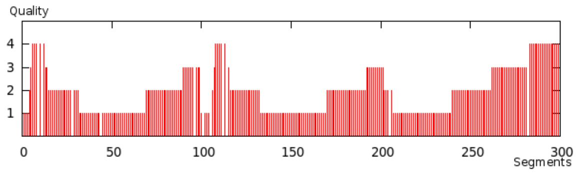


(c) Playout quality

Figure 4: Available bandwidth: 2 Mbps.



(a) Patch delivery time according to playback deadline



(b) Playout quality

Figure 5: Packet loss: 1%.

the playout deadline where segments before the "end patch stream" line show the initial patch stream and segments after it are used for multicast repair. We see that quite a few segments are affected and must be repaired, and that this has a direct effect on the chosen video quality. We see also that even a small loss introduces some deadline misses (points below the "playback deadline" line in figure 5(a) and missing pulses in figure 5(b)). Loss also affects the download time of patch streams (TCP retransmissions), making it harder for the current scheduler to choose an appropriate quality according to available bandwidth. The client often retrieves a lower quality to have a higher probability of a continuous playback.

In our scenario, network delays do not noticeably influence the multicast stream, but as delays directly influence TCP congestion control and recovery mechanisms, we see the results in higher patch delivery times. Similarly, network jitter and out-of-order delivery have large negative effects on interactive and live streaming systems. In our system with minimal playout delays, this leads the segment downloader to choose low quality segments, while higher qualities are chosen for the multicast stream.

4. DISCUSSION AND OPEN ISSUES

One of the goals of this work was to show that the efficiency of existing HTTP streaming systems can be increased easily by combining them with multicast schemes like Patching, and that incremental deployment would be possible. We consider this goal reached.

The paper is also addressing early stream scheduling proposals' assumption of a perfect network with enough bandwidth to receive several concurrent streams. We have introduced the choice of stream duration and between several video qualities and repair capabilities into clients to handle this flaw. These options should be explored further to determine better policies for bandwidth sharing between multicast and unicast, explore acceptable playout delays, and increase the stability of video quality.

We have also seen that some multicast packets are lost under nearly all conditions. This implies that the client should be given at least one round trip time (RTT) to try fixing this. It would then never be necessary to start a multicast stream from the very start of a video.

The available bandwidth can be used to repair packet loss and enhance the quality. Our "as-late-as-possible" repair approach is too optimistic and results in some deadline misses. Repair should be considered in the client's bandwidth budget. A global budget can also be used to re-transmit segments received from a multicast stream at a higher quality using unicast, especially the first segments that are always received at a low quality. This budget can be enforced on the HTTP stream by TCP flow control. A precondition would be the replacement of *libcurl* with a more flexible HTTP client implementation.

In the current prototype, we use AVC-coded segments in several qualities to support quality adaption according to resource availability. This is a reasonable choice for a unicast system designed for HTTP streaming that is not bounded by server disk I/O, and our system is meant to be backwards-compatible. However, a layered codec such as SVC save resources in a system that supports multicast, and wastes less resources when quality decisions can be revised between multicast reception or loss repair and video playout. It would also enable multicast reception according to RLM [9] and PALS-inspired [11] HTTP streaming schemes.

In previous patching schemes, the servers had full control and managed load balancing. Our system is client controlled. Here, the variable loads introduced by the patch streams are managed by separate web servers, i.e., not influencing the multicast, and servers and video quality are selected based on observed load using the tracker. Furthermore, a more P2P-like variant of PULL-PATCHING could also be imagined, where patches are delivered from peers rather than dedicated web servers. Technically, the tracker would be allowed to received registrations for clients' segments.

Another question that should be explored is how close one could get to the same flexibility by applying sub-stream ideas of HMSM [4] to every AVC stream, where subsets of segments are assigned to several multicast streams, where each transmits at less than playout speed.

5. CONCLUSION AND FUTURE WORK

PULL-PATCHING is a streaming technique that combines patching [6] with segmented HTTP streaming [7] to provide a quality-adaptive, scalable video service. PULL-PATCHING overcomes drawbacks of earlier patching-based approaches that make server-side decisions only and that suffer from bandwidth limitations, packet loss and delay, as well as patch server scalability issues. Experiments show that our prototype combines patching, segmented HTTP streaming and quality adaptation quite efficiently and handles the mentioned problems. It is an early prototype, however, and several open challenges for increasing performance are discussed.

6. REFERENCES

[1] Move Networks. http://www.movenetworks.com/.

[2] S. W. Carter and D. D. Long. Video-on-demand server efficiency through stream tapping. In *Proceedings of the International Conference on Computer Communications and Networks (IC3N)*, 1997.

[3] S. Chand, B. Kumar, and H. Om. Segmented patching broadcasting protocol for video data. *Elsevier Computer Communications*, 32(4):679–684, 2009.

[4] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Proceedings of the International Workshop on Multimedia Information Systems (MIS)*, Oct. 1999.

[5] C. Griwodz, M. Liepert, M. Zink, and R. Steinmetz. Tune to lambda patching. *ACM Performance Evaluation Review*, 27(4):202–206, Mar. 2000.

[6] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, Sept. 1998.

[7] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Sav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwodz, and P. Halvorsen. DAVVI: A prototype for the next generation multimedia entertainment platform (demo). In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, Oct. 2009.

[8] Y. Liu, S. Yua, and J. Zhou. Adaptive segment-based patching scheme for video streaming delivery system. *Elsevier Computer Communications*, 29(11):1889–1895, 2006.

[9] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. *ACM Computer Communication Review*, 26(4):117–130, Aug. 1996.

[10] P. Ni, A. Eichhorn, C. Griwodz, and P. Halvorsen. Fine-grained scalable streaming from coarse-grained videos. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2009.

[11] R. Rejaie and A. Ortega. PALS: peer-to-peer adaptive layered streaming. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, June 2003.