Periodic Broadcast and Patching Services - Implementation, Measurement, and Analysis in an Internet Streaming Video Testbed

Michael K. Bradshaw, Bing Wang, Subhabrata Sen, Lixin Gao†, Jim Kurose, Prashant Shenoy, and Don Towsley
Dept. of Computer Science   †Dept. of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
{bradshaw,bing,sen,kurose,shenoy,towsley}@cs.umass.edu, lgao@ecs.umass.edu

1. INTRODUCTION

Multimedia streaming applications consume a significant amount of server and network resources due to the high bandwidth and long durations of audio and video clips. *Patching* and *periodic broadcast* schemes [10, 6, 5, 7, 12] use multicast transmission and client buffering in innovative ways to reduce server and network resource use. To date, existing research on periodic broadcast and patching has been algorithmic in nature, with performance studied either analytically or through simulation. In either case, simplifying assumptions are necessarily made (e.g., abstracting out control and signaling overhead, operating system issues such as the interaction between disk and CPU scheduling, multicast group join and leave times, and more) in order to evaluate performance.

In this paper we report on the implementation, measurement, and analysis of a working video server testbed implementing both periodic broadcast and patching algorithms. Our testbed consists of a Linux-based, application-level video server and a collection of both Linux- and Windows-based clients. While there are a number of existing production (e.g., RealServer, Windows Media Server, Oracle Video Server) and experimental [4, 2] video server efforts, these use either unicast streaming to clients, or a single IP multicast stream; no empirical evaluations of either periodic broadcast or patching algorithms have been made. The goal of this paper is to investigate the underlying system issues that arise when putting such idealized algorithms into practice.

2. DESIGN OVERVIEW

Our server and client implementations are based on existing streaming media standards such as RTP (Real Time Transport Protocol) [9], RTSP (Real Time Streaming Protocol) [11], and SDP (Session Description Protocol) [8], and are designed to run on generic best-effort operating systems such as Linux and Windows. The design and implementation of our video server testbed is based on the following principles: (i) The architecture of our server and end-client is designed to be modular and extensible. The addition of a new service, e.g., patching, involves writing a new module that *plugs* into the existing server architecture and requires minimal changes to other server code. As the control and data path impose significantly different demands on the underlying system, the architecture separates the two functionalities, allowing us to independently optimize each component for superior performance. (ii) *Support for IP Multicast:* Our server and client implementations are designed to take advantage of IP multicast where available. Of particular interest to us are practical issues that arise in the use of IP multicast (e.g., multiplexing a finite number of multicast channels among users, join/leave latencies for users and techniques to hide such latencies). (iii) *Scalability and resilience to network impairments:* A key goal of our server design has been to keep system overheads low. This allows our server to scale to a large number of concurrent users and handle large volumes of streaming media data with timeliness requirements. Our end-clients are designed to be resilient to packet losses and delays in the network.

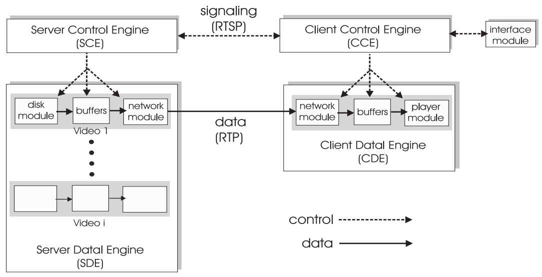

Figure 1: Server and client architecture and interaction

2.1 Server and Client Overview

The server (Fig. 1) consists of two main modules, a **Server Control Engine (SCE)** and a **Server Data Engine (SDE)**, embodying our principle of of separating control and data functionalities. The SCE is primarily responsible for handling (control) interactions between the server and end-clients, using RTSP and SDP protocols. For each client request, it computes: (i) a **transmission schedule** that specifies how each video segment should be retrieved from disk and transmitted over the network, and (ii) a **reception schedule** that specifies the order in which the end-client should receive this data. The transmission schedule is then handed over to the SDE, which then retrieves and transmits data (using RTP) based on this schedule. The reception schedule is formatted as an SDP message and sent to the client in an RTSP response; the client then uses this schedule to receive data on the specified multicast channels. Conceptually, the SDE is an efficient data pump carefully designed to handle high date volumes with low overheads This is important since the data engine will typically need to simultaneously retrieve and transmit multiple portions (segments) of each video.

The client consists of the client control engine (CCE) and data engine (CDE). The CCE obtains user requests using a GUI interface and communicates them to the server using RTSP messages. The client currently supports user interactions such as play, stop, pause, resume and indexed jump. The CDE receives data from multiple video segments according to the reception schedule, possibly in out-of-playback order, and presents the illusion of a logically sequential stream to the decoder software. A clean separation of functionality between the end-client (which is responsible for signaling and receipt of data) and the video player (which is responsible for decoding and display) allows a great deal of flexibility, and enables our client software to inter-operate with several widely used players, including mpeg2decode [1], Real and Windows media players.

3. LESSONS LEARNED

We have gained several key insights from the development of this streaming media platform and subsequent experimental evaluations. The main results are summarized here (details in [3]).

Our server runs on top of Linux, without any underlying real-time support, and with access to clocks with coarse time granularities of 10 msec. or higher. Evaluations indicate that the server is nevertheless well-able to support the real-time, bandwidth intensive data delivery requirements imposed by periodic broadcast and patching. Under periodic broadcast, our server can easily process a client request rate of 600 requests per minute (returning periodic broadcast schedule information to each client), while at the same time streaming video segments over multiple multicast groups and missing very few data transmission deadlines. Under patching, our server can come close to fully loading a 100 Mbps network connection with patched-in clients, again while missing very few data transmission deadlines. Our measurements also show that in a loaded LAN environment, an initial client startup delay of less than 1.5 sec. is sufficient to handle startup signaling and absorb data jitter induced at either the client or the server.

We find that insidious interactions can occur between schedules for different videos. This can lead to situations where the server generated data request rates well above the average demand for short period of times, and these peak loads were above what the server could deliver. This burstiness can be reduced or eliminated through careful scheduling across the different videos [3].

Our evaluations show that dramatic reductions in the bandwidth demand on the underlying server operating system can be gained via application-level data caching even when using a simple Least Recently Used (LRU) cache replacement policy, and that substantial additional performance gains can be realized under a Least Frequently Used (LFU) Replacement Policy.

Finally, it had been shown [7] that there exists a request arrival rate below which patching results in lower bandwidth overheads, and above which periodic broadcast results in lower overheads. In the context of reducing the load on the underlying server operating system, our evaluations demonstrate that the caching scheme and cache buffer size at the server both impact the crossover point and need to be factored in its computation.

4. CONCLUSIONS

In this paper, we presented the design and implementation of an experimental streaming media testbed for investigating scalable streaming solutions like periodic broadcast and patching. We explored and presented solutions to the system and network issues involved in actually implementing these services. Extensive experimental evaluations indicate that the server is well-able to support the real-time, bandwidth intensive data delivery requirements imposed by schemes like periodic broadcast and patching, vindicating many of the key design principles incorporated in the architecture. More generally, our results highlight the importance of combining theoretical work with implementation and empirical evaluation to fully understand systems issues.

5. ACKNOWLEDGMENTS

The authors thank Abhinav Garg and Ellen Zhang for their respective contributions to the development of this testbed. This research was supported in part by the National Science Foundation under Grants CDA-9502639, ANI-9977635, ANI-9805185, NCR-9508274, ANSIR-9977555, and ANIR-9875513, and by a gift from Intel Corporation. Any opinions, findings, and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

6. REFERENCES

[1] mpeg2decode: Public domain MPEG2 Video decoder software. `http://www.mpeg.org/MPEG/video.html#video-softwares`.

[2] K. Almeroth and M. Ammar. An alternative paradigm for scalable on-demand applications: Evaluating and deploying the interactive multimedia jukebox. In *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, July/August 1999.

[3] M. K. Bradshaw, B. Wang, S. Sen, L. Gao, J. Kurose, P. Shenoy, and D. Towsley. Periodic broadcast and patching services - implementation, measurement, and analysis in an internet streaming video testbed. Technical Report 00-06, Department of Computer Science, University of Massachusetts Amherst, 2000.

[4] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. Demonstrating the effect of software feedback on a distributed real-time mpeg video audio player. In *Proc. ACM Multimedia*, November 1995.

[5] D. Eager, M. Ferris, and M. Vernon. Optimized regional caching for on-demand data delivery. In *Proc. Multimedia Computing and Networking (MMCN '99)*, January 1999.

[6] L. Gao, D. Towsley, and J. Kurose. Efficient schemes for broadcasting popular videos. In *Proc. Inter. Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[7] L. Gao, Z. Zhang, and D. Towsley. Catching and selective catching: Efficient latency reduction techniques for delivering continuous multimedia streams. In *Proc. ACM Multimedia*, 1999.

[8] M. Handley and V. Jacobson. SDP: Session description protocol, April 1998.

[9] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. RTP payload format for MPEG1/MPEG2 video, January 1998.

[10] K. Hua and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proc. ACM SIGCOMM*, September 1997.

[11] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp), April 1998.

[12] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proc. Inter. Workshop on Network and Operating System Support for Digital Audio and Video*, 1999.