Title

Tabbycat: an Inexpensive Scalable Server for Video-on-Demand

Authors

Thirumalai, Karthik
Pâris, Jehan-François
Long, Darrell DE

Publication Date

2003

DOI

10.1109/icc.2003.1204467

Peer reviewed

Tabbycat: an Inexpensive Scalable Server for Video-on-Demand

Karthik Thirumalai[1] Jehan-François Pâris[1]

Department of Computer Science
University of Houston
Houston, TX 77204-3010

{karthik, paris}@cs.uh.edu

Darrell D. E. Long[2]

Department of Computer Science
Jack Baskin School of Engineering
University of California
Santa Cruz, CA 95064

darrell@cse.ucsc.edu

***Abstract*—Tabbycat is a video server prototype demonstrating the benefits of a proactive approach for distributing popular videos on demand to a large customer base. Rather than reacting to individual customer requests, Tabbycat broadcasts the contents of the most popular videos according to a fixed schedule. As a result, the number of customers watching a given video does not affect the cost of distributing it. We found that one workstation with a single ATA disk drive and a Fast Ethernet interface could distribute three two-hour videos while achieving a maximum customer waiting time of less than four minutes.**

I. INTRODUCTION

Current wisdom is that the size of a server distributing videos on demand is more or less proportional to the maximum number of concurrent users the server has to support. Hence a metropolitan video-on-demand service capable of handling thousands of concurrent users is generally assumed to require a complex infrastructure, typically consisting of a large number of computing nodes and a sophisticated interconnection network. The Tiger and the nCUBE video servers are good examples of this approach [1, 13].

We built the Tabbycat video server to demonstrate that a simpler, cheaper alternative is possible. Instead of assigning a separate data stream to each customer request, Tabbycat broadcasts each video according to a deterministic schedule guaranteeing that customers will never have to wait more than a few minutes for the video of their choice. Hence, it is the ideal solution for services offering ten to twenty "hot" videos to a large customer base. All studies of video and movie popularity indicate that these top ten or twenty video would be the most profitable to distribute [5]. Even videocassette rental stores focus their efforts in having on hand enough videocassettes of the top videos of the day.

To demonstrate the cost-effectiveness of the approach, we build our Tabbycat using off-the-shelf hardware and software. Our prototype runs on a PC running an unmodified version of Red Hat Linux and is connected to the network through a standard Fast Ethernet interface. Despite these limitations, it can broadcast three two-hour videos and achieve a customer waiting time of less than four minutes.

[1] Supported in part by the Texas Advanced Research Program under grant 003652-0124-1999 and the National Science Foundation under grant CCR-9988390.

[2] Supported in part by the National Science Foundation under grant CCR-9988363.

First Channel	S_1	S_1	S_1	S_1
Second Channel	S_2	S_3	S_2	S_3
Third Channel	S_4	S_5	S_6	S_7

Figure 1. The first three channels for fast broadcasting (FB).

The remainder of this paper is organized as follows. Section 2 reviews the relevant broadcasting protocols for video-on-demand. Section 3 presents our server architecture. Section 4 discusses two possible extensions to the Tabbycat architecture while Section 5 reviews some relevant work. Finally, Section 6 contains our conclusions.

II. BROADCASTING PROTOCOLS FOR VIDEO-ON-DEMAND

All recent VOD broadcasting protocols derive in some way from Viswanathan and Imielinski's *pyramid broadcasting* protocol [16]. Like it, they require special customer set-top boxes (STBs) (a) capable of receiving data at data rates exceeding the video consumption rate and (b) having enough buffer space to store up between, say, ten to sixty minutes of video data. This allows the server to distribute the different segments of each popular video according to a deterministic schedule ensuring that no customer would have to wait more than a few minutes.

The simplest broadcasting protocol is Juhn and Tseng's *fast broadcasting* (FB) protocol [11]. The FB protocol allocates to each video k data channels whose bandwidths are all equal to the video consumption rate b. It then partitions each video into $2^k - 1$ segments, S_1 to S_{2^k-1}, of equal duration d. As Figure 1 indicates, the first channel continuously rebroadcasts segment S_1, the second channel transmits segments S_2 and S_3, and the third channel transmits segments S_4 to S_7. More generally, channel j with $1 \le j \le k$ transmits segments $S_{2^{j-1}}$ to S_{2^j-1}.

When customers want to watch a video, they wait until the beginning of the next transmission of segment S_1. They then start watching that segment while their STB starts receiving data from all other channels. By the time the customer has finished watching segment S_1, segment S_2 will either have been already received or be ready to be received. More generally, any given segment S_i will either be already received or ready to be received by the time the customer has finished watching segment S_{i-1}.

0-7803-7802-4/03/$17.00 © 2003 IEEE

The newer *fixed-delay pagoda broadcasting* (FDPB) protocol [15] requires all users to wait for a fixed delay w before watching the video they have selected. This waiting time is normally a multiple m of the segment duration d. Hence the FDPB protocol can assume that all its clients will start downloading data from the moment they order the video rather than from the moment they start receiving the first segment. The FDPB protocol also uses more sophisticated segment-to-channel mappings than the FB protocol: it uses time-division multiplexing to partition each of the k channels allocated to the video in a given number s_i of *subchannels* of equal bandwidth and allocates the n segments of the video to these subchannels in strict sequential order. As a result, it can achieve smaller waiting times than protocols that do not impose a fixed delay on their customers.

Figure 2 summarizes the segment-to-channel mappings of a FDPB protocol requiring customers to wait for exactly nine times the duration of a segment. The first channel is partitioned into three subchannels each having one third of the channel segment slots. This allows the protocol to repeat segments S_1 to S_3 every 9 slots, segments S_4 to S_7 every 12 slots and segments S_8 to S_{12} every 15 slots. By repeating the same process over all successive channels, the FDPB protocol can map 308 segments into four channels and achieve a deterministic waiting time of 9/308 of the duration of the video, that is, three and half minutes for a two-hour video. Adding a fifth channel would allow the server to broadcast 814 segments and achieve a waiting time of 80 seconds for the same two-hour video.

Channel	*Number of Subchannels*	*First Segment*	*Last Segment*
C_1	3	S_1	S_{12}
C_2	5	S_{13}	S_{42}
C_3	7	S_{43}	S_{116}
C_4	11	S_{117}	S_{308}
C_5	18	S_{309}	S_{814}

Figure 2. The first five channels for the FDPB protocol with $m = 9$.

III. The Tabbycat Prototype

Our goal in building the Tabbycat server was to develop a proof-of-concept prototype of a video server using a state-of-the-art broadcasting protocol. We picked the fixed-delay pagoda broadcasting protocol (FDPB) for several reasons. First, it requires less server bandwidth than most other protocols to achieve the same customer waiting times. Second, it uses fixed-size segments and fixed-bandwidth channels, making it easier to implement than protocols that use variable-bandwidth channels [14] or variable-length segments [8]. Finally, the protocol ensures that each and every segment can be completely received by the STB before the customers start to watch it. As a result, it provides implicit *forward buffering*, which was expected to take care of most of the bandwidth fluctuations inherent to compressed video signal.

A. The Tabbycat Server

A Tabbycat server consists of one or more workstations each distributing a few of the most popular videos. These workstations act autonomously under normal circumstances. Our prototype consisted of a single Pentium 4 system whose characteristics are summarized in Table 1.

We first benchmarked the transfer rate of the ATA drive and found it was about 40 MB/s. Having measured the instantaneous bandwidths of several MPEG-2 videos [12], we found that a typical MPEG-2 would require an average channel bandwidth of 750 kB/s with cartoons having a slightly higher bandwidth than other videos. We also found that the occasional peaks in bandwidth would average out because of *forward buffering* of the FDPB. Hence a Tabbycat with a single ATA drive should be able to broadcast 50 channels. With these 50 channels, we could broadcast 10 videos using 5 channels per video.

TABLE I. Our Prototype Configuration

Server	Intel Pentium 4 1.7 GHz 512 MB Rambus RAM 40 GB ATA-100 (7200 RPM) HDD 100 Mb/s Ethernet Interface Linux Kernel 2.4.x with ext2fs
Network	Fast Ethernet
Clients	Intel Pentium III 600 MHz 256 MB RAM 10 GB ATA-66 HDD 100 Mb/s Ethernet Interface Linux Kernel 2.4.x with ext2fs
Videos	Full-length videos in DVD format (MPEG-2)

TABLE II. Table 2. Summary of Relevant Bandwidths

VOD Channel	750 kB/s
ATA-100 Drive	50 channels (around 40 MB/s)
Fast Ethernet	13 channels (around 10 MB/s)
Gigabit Ethernet	100 channels (around 80 MB/s)

We decided to use UDP instead of TCP because of its low overhead. Though UDP is an unreliable protocol, we found that in a LAN there was almost no packet loss as long as the client and the server were connected to the same Ethernet switch. Moreover, using FDPB in a cable TV environment would mean that the network would have majority of its traffic from the server and given the improvements in network reliability, this seems to be a reasonable choice.

We found that we could achieve speeds of about 10MB/s on a 100Mb/s Ethernet. As a result, we could have about 13 channels per server.

B. The Clients

As shown on Table 2, the clients were older workstations with 600 MHz processors and 256 MB of memory. They rely on the freely available *xine* video player for decoding and

Channel	*Number of Subchannels*	*First Segment*	*Last Segment*
C_1	3	S_1	S_{12}
C_2	5	S_{13}	S_{42}
C_3	8	S_{43}	S_{119}
C_4	13	S_{120}	S_{318}
C_5	19	S_{319}	S_{847}

Figure 3. The first five channels for the modified version of the FDPB protocol with $m = 9$.

playing videos. We increased their kernel network buffer sizes from 65kB to about 70MB to avoid packet losses due to congestion on the client kernel buffers.

C. The Distribution Protocol

Tabbycat uses a slightly modified version of the FDPB protocol. First, Tabbycat clients keep in their buffer all the previously watched segments of each video until the end of that video. As a result, Tabbycat customers are provided with equivalents of the VCR *pause* and *rewind* commands: they can either temporarily suspend the viewing process or return to any video scene they have already watched. Since these two additional features are provided by the STB alone without any server intervention, their sole cost is a few extra gigabytes of additional temporary data on the STB hard drive.

Second, Tabbycat uses a slightly different heuristic for partitioning each channel into subchannels. In the original FDPB protocol, channel C_i was partitioned into a number of subchannels equal to the square root of the period of lowest numbered segment broadcast by C_i. Returning to Figure 2, we see that the lowest numbered segment broadcast by channel C_1 is segment S_1. Since customers have to wait for exactly nine times the duration of a segment, that segment has to be repeated once every 9 slots. Hence channel C_1 is partitioned into $\sqrt{9} = 3$ subchannels.

We found that slightly more efficient segment-to-channel mappings could be achieved by increasing the number of subchannels by one or two in some channels. As seen on Figure 3, this optimization allowed us to map 847 segments into 5 channels and achieve a waiting time of 77 seconds for a two-hour video.

D. Experimental Results

We measured the performance of our server when it was broadcasting three videos on twelve 720 kB/s channels.

All three videos were broadcast using the modified version of the FDPB protocol with $m = 9$. As shown on Figure 3, this allowed us to partition each video in exactly 318 segments, which should allow us to achieve a customer waiting time equal to 9/318 of the duration of each video. Note that this value assumes that the client can start downloading data from segments that have already started. Since our clients could not do that, our customer waiting times will be closer to 10/318 of the video duration

Our first video is a full-feature movie in MPEG-2 format lasting 140 minutes. The observed customer waiting time on a client machine was 273 seconds, that is, 9 seconds more than expected. Our second video is another full-feature movie lasting 130 minutes. Unlike the first video it was encoded at a lower bandwidth (slightly below than 360 kB/s). As a result, our segment transmission time is roughly equal to half its viewing time. This resulted in an observed customer waiting time of 156 seconds. Our third video shows highlights of professional hockey games in MPEG-2 format. The video lasts 25 minutes and the observed customer waiting time is 52 seconds, that is, 5 seconds more than expected.

E. Attacking the Network Interface Bottleneck

Using a 100Mb Ethernet causes the network bandwidth to be a bottleneck as we can only use 25 percent of the available disk bandwidth.

A better solution would be to use a gigabit Ethernet interface. This would allow transfer rates of about 80 MB/s. Unfortunately, the disk bandwidth limits us to 40 MB/s, that is, half of that bandwidth.

F. Attacking the Disk Bottleneck

There would be several ways to eliminate that disk bottleneck. First we could attach to each workstation two SCSI disk drives and divide the disk workload among these two drives. The main disadvantage of this solution is the higher cost of SCSI drives.

A second option would be to wait for newer and better ATA drives. Disk densities have been doubling every year over the last few years and there is no reason to expect a sudden halt to this trend. Even without an increase of disk rotational speeds, we can thus expect disk transfer rates to increase by a factor of $\sqrt{2}$ every year. Within two years, we should be able broadcast 100 channels from a single disk drive. This will be enough to broadcast the 20 hottest videos using 5 channels per video and achieve a waiting time of 81 seconds.

A third option would be to store in the server's main memory the most frequently transmitted segments of each video . Since we are using a deterministic broadcasting protocol, we can predict ahead of time the I/O bandwidth savings that could thus be achieved.

Assume that our broadcasting protocol partitions each video to be broadcast into n fixed size segments of equal duration $d = D/n$ where D is the duration of the video. Assume also that the protocol repeats segment S_i every $z(i)$ slots. For all protocols that impose a fixed delay, we would have $z(i) > i$. Then the total bandwidth required to broadcast the first k segments of the video is given by:

$$\sum_{i=1}^{n} \frac{b}{z(i)}$$

where b is the video consumption rate.

The fraction of the total bandwidth occupied by the first k segments is then given by:

$$\frac{\sum_{i=1}^{k} \frac{b}{z(i)}}{\sum_{i=1}^{n} \frac{b}{z(i)}} = \frac{\sum_{i=1}^{k} \frac{1}{z(i)}}{\sum_{i=1}^{n} \frac{1}{z(i)}}$$

Recall that the bandwidth of a single ATA drive allows us to broadcast up to 10 two-hour videos on 50 channels and achieve a waiting time of 81 seconds. Assuming $m = 9$, the first channel allocated to each video would broadcast the first 12 segments of the video, that is $12/847^{th}$ of the total duration of the video. Caching these 12 segments in main memory would require $12/847 \times 7200 \times 0.75 = 76.5$ MB per video and reduce the disk bandwidth by one channel. Caching the first 12 segments of the 10 videos would reduce the disk bandwidth by the equivalent of ten channels. This would allow us to broadcast 2 additional videos at the cost of 765 MB of additional main memory. Note that the cost of caching the first few segments of a video is directly proportional to the duration of these segments and the duration of the video. Hence, caching would work very well for a server distributing shorter video clips.

G. Fault-Tolerance Issues

In the current state of the technology, a reasonably sized Tabbycat server would probably consist of 3 workstations connected to the net through Gigabit Ethernet interfaces that would allow it to broadcast the 18 most popular videos on a total of 90 channels, using slightly less than two-thirds of the available bandwidth. Each video would be replicated on two of the three workstations in such a way that each workstation will have backup copies of one half of the videos normally broadcast by the two other workstations. Should one of the workstations fail, each of the two remaining workstations will add to its normal broadcast schedule one half of the videos that were broadcast by the machine that failed.

IV. Possible Extensions

Two possible extensions could greatly enhance the current implementation of our Tabbycat prototype.

A. Implementing a Reliable Multicast Protocol

Since our prototype relies on UDP for distributing the videos, its applicability is limited to either cable TV environments or well-controlled LANs, where packet losses are small enough to be tolerated by the video-encoding scheme. Deploying Tabbycat over a shared WAN would require implementing a reliable multicast protocol. We are currently investigating several possible solutions.

B. Limiting the Client Bandwidth

Tabbycat now requires each STB to receive at the same time data from all the k channels allocated to the video being currently watched. This requirement complicates the design of the client and increases its cost [9, 6]. A better solution would be to use a FDPB protocol limiting the STB receiving bandwidth to two channels.

Channel	*Number of Subchannels*	*First Segment*	*Last Segment*
C_1	3	S_1	S_{12}
C_2	5	S_{13}	S_{42}
C_3	6	S_{43}	S_{95}
C_4	8	S_{96}	S_{193}
C_5	11	S_{194}	S_{369}

Figure 4. The first five channels for a FDPB protocol with $m = 9$ restricting the client bandwidth to two channels and not allowing channel hopping.

Channel	*Number of Subchannels*	*First Segment*	*Last Segment*
C_1	3	S_1	S_{12}
C_2	5	S_{13}	S_{42}
C_3	6	S_{43}	S_{100}
C_4	10	S_{101}	S_{220}
C_5	14	S_{237}	S_{474}

Figure 5. The first five channels for a FDPB protocol with $m = 9$ restricting the client bandwidth to two channels and allowing channel hopping.

One possible solutions to this problem is to design an FDPB protocol assuming that the client STB will not be able to receive data from channel C_{l+1} until it is done with all channels C_i with $i < l$ [15]. The main advantage of this approach is that the STB will never have to hop back and forth between the channels. Its major drawback is that we will not be able to map as many segments into the same number of channels. This will result into either an increase of the customer waiting time or an increase of the number of channels required to achieve the same waiting time.

As shown in Figure 4, a FDPB protocol with $m = 9$ restricting the client bandwidth to two channels and not allowing channel hopping would only be able to map 193 segments into four channels. Hence it would only achieve a waiting time equal to $9/193^{th}$ of the video duration, that is, a little less than 6 minutes for a two-hour video. Adding a fifth channel would bring the waiting time below three minutes for the same two-hour video.

We present here a more efficient solution. Rather than waiting for the STB to be entirely done with channel C_i before starting to receive data from channel C_{i+2}, we will let the STB receive data from some of the subchannels of channel C_{i+2} at the same rate it stops receiving data from some of the subchannels of channel C_i. This process will be conducted in such fashion that the customer STB will never have to receive data at the same time from channels C_i and C_{i+2}.

As shown in Figure 5, a FDPB protocol with $m = 9$ restricting the client bandwidth to two channels while allowing channel hopping could map 220 segments into four channels. Hence it would achieve a waiting time equal to $9/220^{th}$ of the video duration, that is, a little less than 5 minutes for a two-hour video. Adding a fifth channel would allow us to partition the video into 474 segments and achieve a waiting time of 137

seconds for the same two-hour video. Hence allowing channel-hopping results in a 22 percent reduction of the customer waiting time.

V. Relevant Work

The Berkeley Distributed Video-on-Demand System [3] allowed clients across the Internet to submit requests to view audio, video and graphical streams. Playback was accomplished by streaming data from a media file server through the network to the client's computer. No effort was made to share data among overlapping requests.

The Tiger system was a scalable, fault-tolerant multimedia file system using commodity hardware [1]. Unlike Tabbycat, it dedicated a separate data stream to each customer request and made no attempts to share data among overlapping requests. Hot spots were avoided by striping all videos across all workstations and disks in a Tiger system. Tiger prevented conflicts among requests by scheduling incoming requests in a way that ensures that two requests will never access the same resource at the same time. This task was distributed among all of the workstations in the system, each of which having an incomplete view of the global schedule. The main disadvantage of the approach was its poor scalability: Tiger designers found that a system with ten workstations could only handle one hundred concurrent user requests.

More recently, Bradshaw et al. have presented an Internet streaming video testbed [2] using both periodic broadcast and patching/stream tapping [4, 10]. This allowed the server to select the best distribution protocol for each video, namely, broadcasting videos in very high demand while distributing less popular videos through stream tapping/patching. This feature resulted in a much more complex system than our Tabbycat server. In addition, the *greedy disk-conserving broadcasting* protocol [7] used by their system is less bandwidth-efficient than the optimized FDPB protocol used by Tabbycat. While the optimized FDPB protocol only requires five channels to achieve a waiting time of 77 seconds for a two-hour video, the greedy disk-conserving broadcasting protocol requires 6 channels to achieve a waiting time of 114 seconds for the same video.

VI. Conclusion

Current wisdom is that distributing video on demand to large audiences requires complex expensive farms of video servers. We built the Tabbycat video server to show that a metropolitan video server distributing the top ten to twenty videos could consist of a few powerful workstations running unmodified versions of a standard operating system. The secret of Tabbycat's low cost is the broadcasting protocol it uses to distribute the videos. We knew ahead of time that the FDPB protocol would provide smaller waiting times than any other protocol broadcasting fixed-size segments over channels of equal bandwidth [15]. We found it easy to implement and easy to tune thanks to its regularity. In addition, we showed that a FDPB protocol restricting the client bandwidth to two channels could still achieve a waiting time of slightly less than 126 seconds for a two-hour video broadcast over five channels.

As it stands now, Tabbycat is a mere proof-of-concept prototype. More work is still needed to allow its deployment in environments where packet losses are likely to happen.

References

[1] W. J. Bolosky, R. P. Fitzgerald and J. R. Douceur. "Distributed schedule management in the Tiger video fileserver." *Proc. 16th ACM Symp. on Operating Systems Principles*, pp. 212–223, October 1997.

[2] 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." *Proc. 9th ACM Multimedia Conf.*, Oct. 2001

[3] D. W. Brubeck and L. A. Rowe. "Hierarchical storage management in a distributed video-on-demand system." *IEEE Multimedia*, 3(3):37–47, 1996.

[4] S. W. Carter and D. D. E. Long. "Improving video-on-demand server efficiency through stream tapping." *Proc. 5th Int'l Conf. on Computer Communications and Networks*, pp. 200–207, Sept. 1997.

[5] A. Dan, D. Sitaram, and P. Shahabuddin. "Dynamic batching policies for an on-demand video server." *Multimedia Systems*, 4(3):112–121, June 1996.

[6] D. L. Eager, M. K. Vernon and J. Zahorjan. "Minimizing bandwidth requirements for on-demand data delivery." *IEEE Trans, on Knowledge and Data Engineering*, 13(5):742-757, Sept.-Oct. 2001.

[7] L. Gao, J. Kurose, and D. Towsley. "Efficient schemes for broadcasting popular videos." *Proc. Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[8] A. Hu, I. Nikolaidis, P. van Beek. "On the design of efficient video-on-demand broadcast schedules." *Proc. 7th Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 262–269, Oct. 1999.

[9] Hua, K. A., and S. Sheu, "Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems," *Proc. SIGCOMM 97 Conf.*, pp. 89–100, Sep. 1997.

[10] K. A. Hua, Y. Cai, and S. Sheu. Patching: a multicast technique for true video-on-demand services. *Proc. 6th ACM Multimedia Conf,* pp. 191–200, Sep. 1998.

[11] L. Juhn and L. Tseng. "Fast data broadcasting and receiving scheme for popular video service." *IEEE Trans. on Broadcasting*, 44(1):100–105, March 1998.

[12] Saurabh Mohan. *Characterizing the Bandwidth Requirements of Compressed Videos*. MS Thesis, Department of Computer Science, University of Houston, May 2001.

[13] nCUBE Corp., http://www.ncube.com/vod/index.html.

[14] J.-F. Pâris, S. W. Carter and D. D. E. Long. "A low bandwidth broadcasting protocol for video on demand." *Proc. 7th Int'l Conf. on Computer Communications and Networks*, pp. 690–697, Oct. 1998.

[15] J.-F. Pâris. "A fixed-delay broadcasting protocol for video-on-demand." *Proc. 10th Int'l Conf. on Computer Communications and Networks*, pp. 418–423, Oct. 2001.

[16] S. Viswanathan and T. Imielinski. "Metropolitan area video-on-demand service using pyramid broadcasting." *Multimedia Systems*, 4(4):197–208, 1996.