Co-operative Proxy Caching Algorithms for Time-Shifted IPTV Services

T. Wauters, W. Van de Meerssche, F. De Turck, Bart Dhoedt, P. Demeester
Dept. Information Technology (INTEC), Ghent University – IMEC – IBBT,
Gaston Crommenlaan 8, bus 201, B-9050 Ghent, Belgium,
{tim.wauters, wim.vandemeerssche, filip.deturck, bart.dhoedt, piet.demeester}@intec.ugent.be.

T. Van Caenegem, E. Six
Alcatel R&I, Access and Edge, Francis Wellesplein 1, B-2018 Antwerp, Belgium,
{tom.van_caenegem, erwin.six}@alcatel.be.

Abstract

The increasing popularity of multimedia streaming applications introduces new challenges in content distribution networks. Streaming services such as Video on Demand (VoD) or digital television over the Internet (IPTV) are very bandwidth-intensive and cannot tolerate the high delays and poor loss properties of today's Internet. To solve these problems, caching (a sliding segment of) popular streams at proxies could be envisaged. This paper presents a novel caching algorithm and architecture for time-shifted television (tsTV) and its implementation using the IETF's Real-Time Streaming Protocol (RTSP). The algorithm uses sliding caching windows with sizes depending on content popularity and/or distance metrics. The caches can work in stand-alone mode as well as in co-operative mode. This paper shows that the network load can already be reduced considerably using small diskless caches, especially when using co-operative caching. A brief overview of the functionality of a prototype proxy implementation is presented as well.

1. Introduction

The use of Content Distribution Networks (CDNs) for the delivery of bandwidth-intensive streaming multimedia has increased considerably over the last few years. Companies such as Akamai [1] deploy their CDNs by replicating the content to multiple surrogate servers, generally at the edge of the network. This way, the content only has to pass a few nodes in order to reach the end users, resulting in a reduced latency and increased throughput. By using geographically distributed servers, scalability and reliability are improved significantly, while the core network and origin server load are reduced. However, for a large-scale deployment of video services such as IPTV, the installation of VoD servers at the edge of regional networks is not sufficient anymore, due to growing bandwidth requirements in the access network. To meet the increasing user demand, streaming servers would be required at each first aggregation point, resulting in a very high equipment cost.

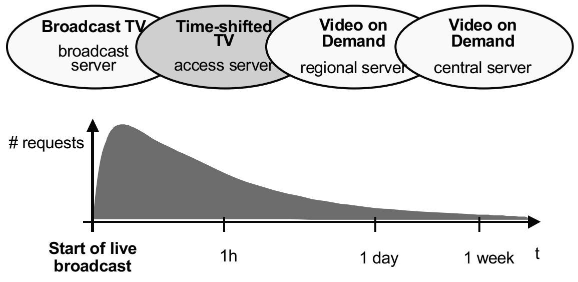
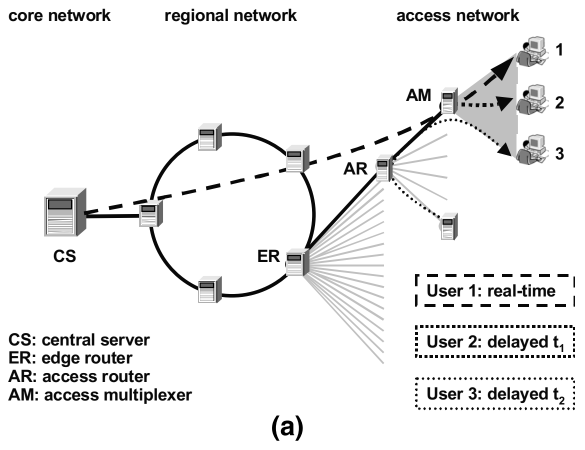


Figure 1. Delivery mechanisms for IPTV

Various approaches to offload the transport networks have been proposed recently. A significant number of these solutions use technologies that allow time-shifted TV, a service that enables the end-user to watch a broadcasted TV program with a time shift, i.e. the end-user can start watching the TV program from the beginning although the broadcasting of that program has already started or is already finished. Dedicated home equipment, such as a home Personal Video Recorder (PVR), enables time-shifted TV through a built-in memory that can be programmed to record broadcasted TV programs. Limited throughput capacity on the storage device and on the access link to the end-user's home however are major disadvantages of home PVR solutions. Furthermore, home PVR devices can be expensive.

A time-shifted TV service can also be offered through PVR functionality in the network. In principle, such network PVR solution enables the end-user to watch at any time any program broadcasted on any channel. The end-user experience is similar to a VoD service, but the content is live TV with an arbitrary time shift relative to the original broadcast time. As shown in Fig. 1, the popularity of a television program typically reaches its peak value within several minutes after the initial broadcast of the program and exponentially decreases afterwards. This means that caching a segment with a sliding window of several minutes for each current program can serve a considerable part of all user requests for that program. Therefore, a new network based time-shifted television (tsTV) solution using low cost distributed streamers with limited storage capacity will be presented. These streamers can be located at the proxy caches and store segments of the most popular content (TV programs), so that all requests arriving within these intervals can be served by the cache from start to finish.

In Fig. 2a and 2b, user 1 is the first to request a certain television program and gets served from the central server. Afterwards, other requesting users (e.g. user 2) can be served by the proxy, as long as the window of the requested program is still growing. After several minutes, the window stops growing and begins sliding, so that user 3 cannot be served anymore and will be redirected to the (central or regional) server or, in case of co-operative caching, to a neighbor proxy with the appropriate segment, if present. Pausing (parallel to the horizontal axis) can also be supported within the segment window, as well as fast forward or rewind (parallel to the vertical axis).



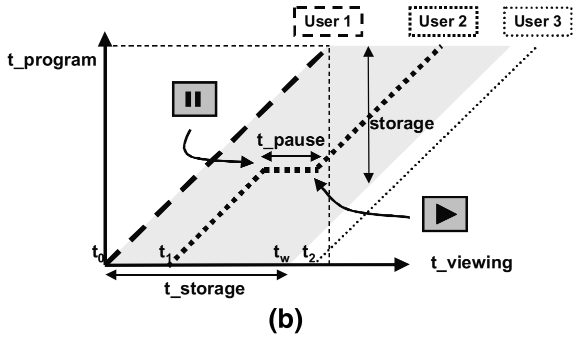


Figure 2. Time-shifted television: (a) typical access network topology and (b) tsTV streaming diagram

The remainder of this paper is structured as follows. Research work related to this research is discussed in section 2. Section 3 presents an analytical model of the sliding-interval caching problem with fixed window sizes, for comparison with our caching algorithms and to have an initial estimate of the required storage space. The next section introduces and evaluates our sliding-interval caching algorithms, for both stand-alone and co-operative caching. It determines the location and the size of the different segments at the proxy caches. Experimental results are obtained using a discrete event simulator. In section 5, the RTSP implementation is discussed briefly. Section 6 concludes this paper.

2. Background and related work

Previous studies on proxy caching techniques [2] or distributed replica placement strategies for CDNs [6,7] show that greedy algorithms that take distance metrics and content popularity into account perform better than more straightforward heuristics such as LRU (Least Recently Used) or LFU (Least Frequently Used).

Segment-based caching techniques have been studied extensively for streaming media, due to the huge size of multimedia streams compared to traditional web objects. A survey on different segment-based strategies such as prefix caching, segment caching, rate-split caching and sliding-interval caching has been presented in [2]. The main goal of prefix caching is to reduce the start-up delay by caching the initial portion of the stream at the proxy. This paradigm is generalized by segment caching, where cache decisions are made for a series of segments of the stream. In rate-split caching, the partitioning is done along the rate axis, instead of along the time axis. This way, the cache takes care of the peak rates in VBR streaming, while the backbone only has to cope with the lower constant rate. Of particular interest for this study is sliding-interval caching [3], where the cached portion of the stream is initially a growing prefix, but afterwards a dynamically updated sliding interval. This way, consecutive requests can be served from start to finish within this window. A more advanced aspect is the use of co-operative proxy caching [4], where a better performance than with independent proxies can be achieved through load balancing and improved system scalability. In this case it is important to continuously keep track of cache states. Note that contrary to standard co-operative proxy caching, there is no need to switch to segments on other proxies when using co-operative proxy caching with sliding intervals. Similar peer-to-peer caching techniques have also been introduced in streaming CDNs, where whole files are stored instead of segments [5]. Several studies such as [8] have been investigating the implementation of segment-based caching techniques on proxies using the RTP / RTCP / RTSP protocol suite. A demonstrator of an IP aware multi-service access network, including our prototype tsTV setup, has been described in [9].

3. Analytical approach

Before presenting our sliding-interval caching algorithm, we introduce an analytical model of a tsTV solution based on sliding-interval caching with fixed window sizes, offering a method to estimate the required storage space in the network.

3.1. Model parameters

Consider a model where each TV program is characterized by a start time τ_i, a duration T_i and a function $\lambda_i(t)$, representing the request arrival rate for this program. $N(t)$ denotes the total number of programs with $\tau_i \le t$. The proxy cache I, placed between the server and the clients, contains the first X minutes of any currently streaming file with $t - T_i \le \tau_i \le t$.

3.2. Cache hit rate

We derive an expression for the hit rate of cache I, $h_I(t)$. Consider further the time period $|t, t + \Delta t|$, then the total number of requests is given by

$$\sum_{i=1}^{N(t)} \lambda_i(t)\Delta(t).$$

To find the total number of successful requests (i.e. requests that can be served by the cache) for the currently broadcasted program j in a single channel situation, we assume a uniform distribution for τ_j and make the following observations:

- these requests have to arrive at most X minutes after τ_j
- only a fraction X / T_i of the requests is served from cache I

Therefore the total number of successful requests is given by

$$\lambda_i(t)\Delta(t)\frac{X}{T_j}.$$

Averaging over all programs j for which $t - X \le \tau_i \le t$, multiplying by the total number of channels K and supposing that popularity and duration are uncorrelated, we obtain the following expression:

$$h_I(t) = K\frac{<\lambda_i(t)>^*}{<T>}\frac{X}{\sum_{i=1}^{N(t)} \lambda_i(t)},$$

with <>* denoting averaging, on the condition that $t - X \le \tau_i \le t$. Supposing further that λ_i is a separable function of i and t, such that $\lambda_i(t) = \lambda_i f(t - \tau_I)$, with $f(t)$ a normalized function such that $f(t) = 0$ for $t < 0$ and

$$\int_0^\infty f(t)dt = 1,$$

we can write:

$$<\lambda_j(t)>^* = <\lambda_j><f(t-\tau_j)>^*$$
$$= \frac{<\lambda_j>}{X}\int_0^\infty f(t)dt$$

as long as $X < <T>$. Hence,

$$h_I(t) = K\frac{<\lambda>}{<T>}\frac{\int_0^X f(t)dt}{\sum_{i=1}^{N(t)} \lambda_i(t)}.$$

Further consider a time period P, then the total number of broadcasted programs is $N(P) = KP/<T>$. Suppose a user group of size G, each requesting r programs per second on average, then the total number of requests is given by GrP. Therefore, the average number of requests for a long enough period of time will satisfy

$$<\lambda> = \frac{GrP}{N(P)} = \frac{GrP}{KP/<T>} = \frac{Gr<T>}{K}.$$

On the other hand, the total number of requests per time unit is given by

$$\sum_{i=1}^{N(t)} \lambda_i(t) = Gr,$$

simplifying our expression for the cache I hit ratio to

$$h_I = \int_0^X f(t)dt.$$

Taking for $f(t)$ an exponentially decreasing function bexp($-bt$) (for t>0), we get

$$h_I = 1 - e^{-bX}$$

as long as $X << T$. The size of cache I is simply KX.

3.3. Example results

When the content popularity is halved after each interval Δ ($b = -\ln(0.5)/\Delta$), the server load looks like presented in Fig. 3. It is given by ($X = a\Delta$)

$$1 - h_I = \left(\frac{1}{2}\right)^a .$$

Similar results for the server load can be found using the sliding-interval caching algorithm presented in the following section (comparable to the "s -> c1" curve in Fig. 9a, for stand-alone caches at level 2).

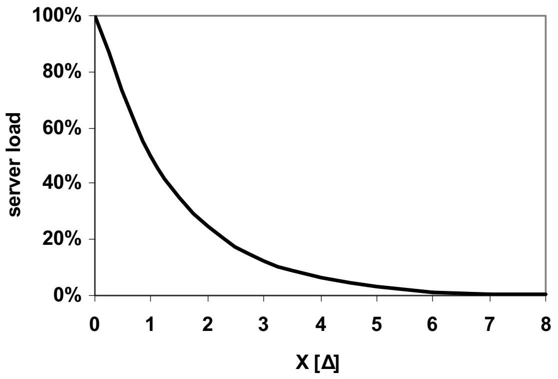


Figure 3. Analytical solution for the server load, for different values of the segment size

We can conclude that in case of an exponentially decreasing temporal content popularity, the server load decreases proportionally, for increasing segment sizes.

4. Sliding-interval caching algorithm

Our caching algorithm for tsTV services is presented in this section. Since we assume that in general only segments of programs will be stored, cache sizes can be limited to a few gigabyte (corresponding to a few hours of streaming content). This way smaller streaming servers can be deployed closer to the users, without increasing the installation cost excessively.

4.1. Basic principle

We propose that the cache is virtually split up in two parts: a small part S and a main part L.

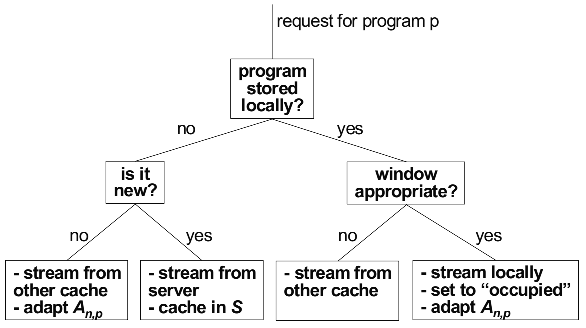


Figure 4. Basic principle of the tsTV caching algorithm at each proxy

Part S will be used to cache the first few (e.g. 5) minutes of every newly requested (or broadcasted) program, mainly to determine its initial popularity. Its size is generally smaller than 1 GB (typically 1 hour of streaming content). Part L will be used to actually store the appropriate segments (with growing or sliding windows). This part is again virtually divided into two separate storage spaces. Part L_1 is used to store unique segments only, shared among all co-operating cache nodes. This way, all parts L_1 on all cache nodes represent one large cache, mainly to offload the central server. The second part L_2, if there is still storage space left, is then used to store segments that are locally most popular. The main goal of that part is to offload the access network links, used by the co-operative caching mechanism (requests served by L_1 on a neighbor cache). The actual size of each segment in part L_2 will be determined and, if necessary, adapted after each interval Δ (e.g. 5 minutes). After Δ, one of the following decisions has to be taken:

- let the segment grow (for very popular programs);
- let the segment slide (to finish the current requests, for less popular programs);
- drop the segment (for unpopular programs, with no current requests to be served).

Fig. 4 shows the basic principle of the tsTV caching algorithm. During each interval Δ, program requests arrive at the different proxies. Each time, a parameter $A_{n,p}$ will be updated in proxy n, for program p. In general, this parameter tries to determine the popularity of the program, while taking distance metrics into account. This means that a (segment of a) popular program might not be cached, because a nearby proxy already stores that (segment of the) program. $A_{n,p}$ is calculated as follows:

Each time a request for program p *arrives at proxy* n, $\mathrm{A}_{\mathrm{n,p}}$ *is increased by 1 (only taking popularity into account) or by the hopcount between proxy* n *and the serving node (also taking distance into account).*

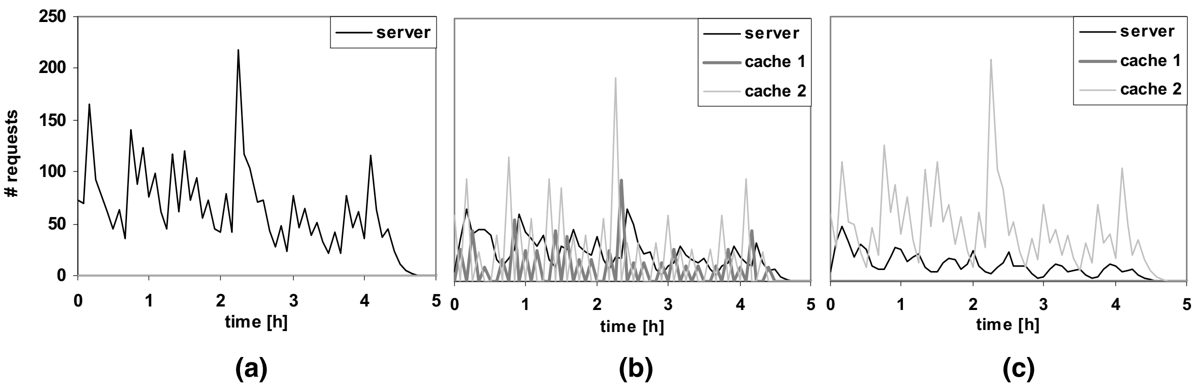


Figure 5. Server and cache load. All requests are made within 30 minutes. The cache sizes are 0 GB (a), 0.5 GB (b) and 4 GB (c)

After each interval Δ, first all segments (sliding or growing) with status set to "occupied" are stored in L_2. Afterwards L_2 is filled with segments with growing windows for the most popular programs (i.e. with the highest values of $A_{n,p}$). All other segments are dropped, S is cleared and all values of $A_{n,p}$ are reset to 0.

4.2. Numerical results for stand-alone caching

4.2.1. Input parameters. To illustrate the stand-alone caching principle (with hierarchical caches), a first set of simulations was performed on one branch of the access network tree of Fig. 2a: a regional server with two hierarchical caches (Fig. 6).

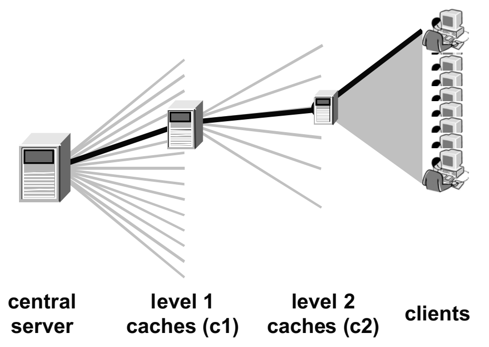


Figure 6. Basic access network topology

The regional server offers 20 channels: 5 very popular channels (80% of all requests), 5 less popular channels (10% of all requests) and 10 unpopular channels (10% of all requests). The top 5 channels are served as a tsTV service, the other channels through standard VoD technology on the regional server. The popularity of the programs per channel follows a Zipf-like distribution with parameter β = 0.7 (the popularity of the *i*'th most popular program is proportional to $i^{-\beta}$). This distribution is commonly used for content distribution [10,11] and TV viewing measurements like [12] confirm this trend. A total of 3000 requests are made during one evening, of which 200 for the most popular program on the most popular channel. The popularity of a program reaches a peak during the first interval Δ (= 5 minutes) and decreases exponentially afterwards (halved every interval Δ) (similar to Fig. 1). Each channel offers 6 programs of 45 minutes per evening, with a streaming bandwidth of 2.5 Mbps.

4.2.2. Server and cache load. In Fig. 5, the server and cache load are presented. When both cache sizes are limited to 0.5 GB (*S* only: the number of channels times Δ or 25 minutes, Fig. 5b), the server load is much lower than without caches (Fig. 5a) and the caches serve most of the tsTV requests. What happens is that cache c1 (closest to the server) and cache c2 first store all 5-minute prefixes of each new program, but since only cache c2 receives new requests afterwards, cache 1 will drop these segments after Δ. Afterwards cache 1 will store the next 5 minutes of each program, while cache c2 is storing the sliding "occupied" windows from the first interval. This means that the caches serve all requests made during the first 10 minutes of each single program. For infinite cache sizes (or 4 GB or higher in this example, Fig. 5c), the regional server only serves the VoD requests for channels 6 to 20. Cache c2 stores and serves all currently broadcasted popular programs, thereby effectively offloading the transport network.

More detail on the regional server and cache load is given in Fig. 7 (tsTV only, top 5 channels). Note that the server load never drops to 0, since at least the first request for a certain program has to be served from the regional server. In Fig. 8 the server load is shown for different values of the maximum request period per program. Since no upstream links are used in these simulations, the bandwidth on the links can easily be determined from the server and cache load.

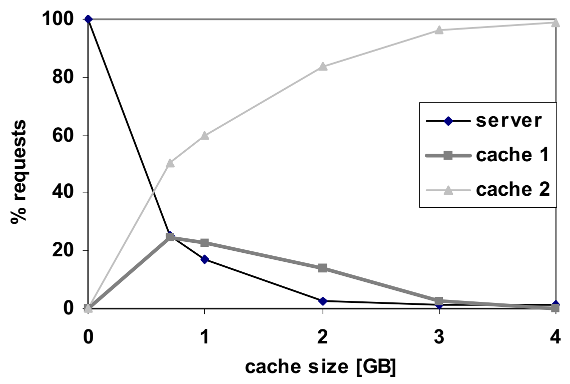


Figure 7. Server and cache load. All requests are made within 30 minutes

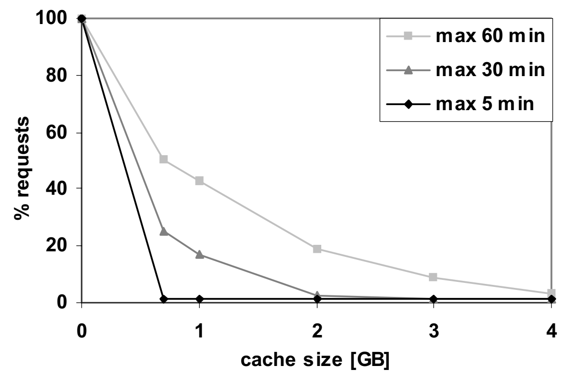


Figure 8. Server load for different values of the maximum request period

4.3. Numerical results for co-operative caching

The same caching principles can be applied for a co-operative caching mechanism, where caches on the same level of the broadcast tree can collaborate, using peer-to-peer protocols to exchange information on stored content. Contrary to stand-alone caching, where a request that cannot be served is forwarded to the next cache on the path to the central server (hierarchical caching), caches can now forward requests to caches on the same level. However, the decision on when to store a certain fragment not only depends on the value of $A_{n,p}$, but also on the source node serving the request. Two different approaches can be distinguished.

The first heuristic only takes the values of $A_{n,p}$ into account ("Cache from All sources", *CfA*). This means that the storage space $L = L_2$, so that most caches store the same fragments, since content popularity is similar for most nodes. The numerical results will therefore be comparable to the results for stand-alone caching.

The second heuristic also takes the values for $A_{n,p}$ into account, but never stores content that is already stored on another cache ("Cache from Server only", *CfS*), so that $L = L_1$. This way the central server will be offloaded considerably, even with small caches, but many requests will have to be served by other caches over the access network links.

Both alternatives have their benefits (the first one is optimal in case of larger caches, the second one in case of small caches). The optimal heuristic however takes the best of both worlds, storing unique content segments in L_1 and locally popular segments in L_2.

This way the central server load is always minimized first: the expected server load using the storage space combining all parts L_1 can then be determined out of Fig. 3. The access network load can be reduced afterwards, if the cache space is large enough. This heuristic is called "Cache from Elected sources" (*CfE*).

4.3.1. Input parameters. The input parameters for the simulations are the same as in the previous section. The network topology (similar to Fig. 6) now consists of a central server, one node at level 1 (without storage capabilities) and 6 proxy caches at level 2. The level 1 node is connected to the level 2 caches with bidirectional links, so that cache co-operation is possible.

Note that no storage space is available at the level one node so that the results of the simulations for cache co-operation are not influenced by hierarchical caching.

The cost of using the link from the central server to the node at level 1 has been set to a value higher than 1 (the cost of an access network link). This way the central server will be avoided when the requested segment can already be found on a neighbor level 2 cache (when calculating the shortest path using the weighted Dijkstra algorithm).

4.3.2. Server, cache and network load. In case of stand-alone caching, the network bandwidth can easily be determined out of the cache and server load (Fig. 7), since only downstream traffic is present on the access network. With co-operative caching, the uplinks in the access network are used as well.

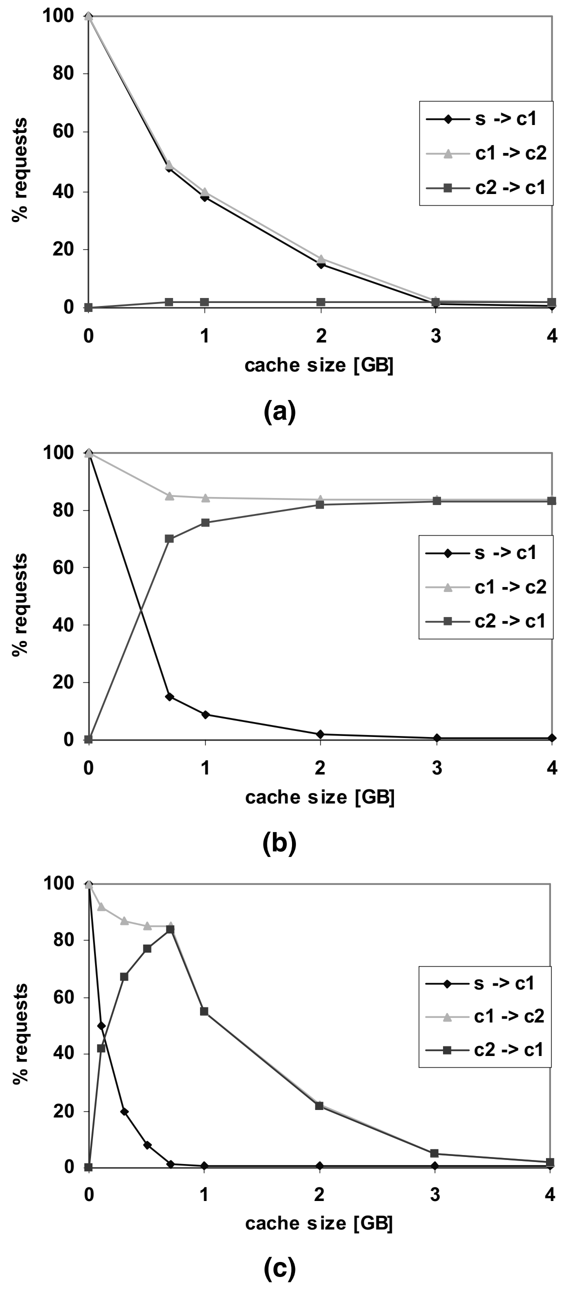


Figure 9. Fraction of the streams on the links between the server and the level 1 node (s -> c1) and between the level 1 and 2 nodes (downlink c1 -> c2 and uplink c2 -> c1) for the CfA (a), CfS (b) and CfE (c) heuristics

Using the *CfA* heuristic (Fig. 9a), the server load is almost identical to the case where stand-alone caches on level 2 are used. The only difference is that the central server does not need to serve the first stream to all of the 6 proxies, but only to one of them. Again the central server load for the tsTV channels drops to (almost) zero when 4 GB caches would be used. The uplinks from the level 2 caches to node 1 are almost never used, since all caches store the same fragments. The results are therefore very similar as for stand-alone caching (remember the analytical results of Fig. 3, with 1GB = 10 minutes per channel = 2Δ).

When the *CfS* heuristic is used (Fig. 9b), each 5-minute (Δ) fragment is only stored on one cache. This way, the central server load is already almost zero for the tsTV channels when only 0.5 GB caches are used. The total storage space is then 3 GB, therefore one could expect that the results for the central server load would correspond to the situation with 3 GB caches in stand-alone mode. This is not entirely the case, since it is possible that the first requests for a new program arrive at caches that have no storage place left in L_1. These first requests are then served by the central server.

The “core network load” (represented by the link “s -> c1”) is reduced considerably, while the “access network load” (represented by the links “c1 <-> c2”) is load balanced.

The *CfE* heuristic (Fig. 9c) offers the best of both worlds. The server load is reduced effectively, while, in case of larger caches, the access network is offloaded as well. The server load (link “s -> c1”) is even lower then for the *CfS* heuristic. This is due to the RTSP request forwarding mechanism, allowing requests that arrive at a cache that has no storage space left in L_1, to be forwarded automatically to another cache with enough storage space. This way the virtual cache consisting of all parts L_1 is filled up in an optimal way.

5. Proxy implementation

An RTSP proxy for time-shifted TV has been implemented for evaluation purposes. This section gives a brief overview of the most important components and protocols used.

In order to implement the proxy, its functionality is divided into logical parts. The communication with the users and the central server includes messages containing data about which program or channel has to be streamed, or VCR like commands such as PAUSE and STOP. A protocol commonly used for this interaction is RTSP (Real-Time Streaming Protocol, RFC 2326). The streams themselves are encapsulated and delivered with RTP (Real-Time Protocol, RFC 1889), a standard protocol for live streamed media.

A first functional component of the proxy is the *RTSP Proxy*, a component that communicates with the tsTV clients and the server using RTSP, interprets their messages and commands the other components to execute these requests. The *RTSP Proxy* component delegates the caching algorithm decisions to another component, the *Cache Verdict Manager*, a component that uses information from the *Cache State Manager*, which is updated through a centralized or distributed Cache State Exchange (CSE) protocol. The task of the *Cacher* component is to store popular streams, sent to the proxy by the server (or another cache), in sliding windows. The streams are sent to the clients from these windows, a function that is handled by the *Streamer* component. The proxy also keeps track of the streams that are being sent to the proxy (which program, channel, starting time, …), through the *Stream Tracker* component, with help from the *Program Guide* component, which communicates with the electronic program guide (EPG) server. Fig. 10 gives an overview of the different components.

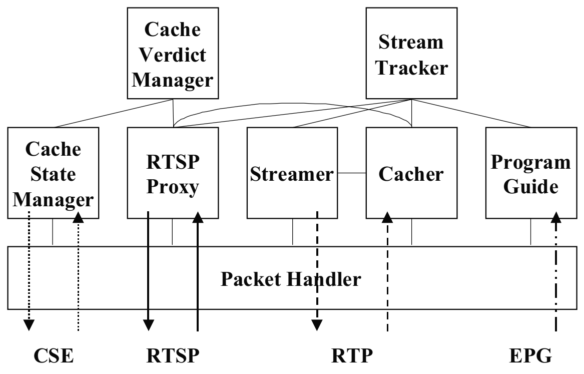


Figure 10. Overview of the different components in the proxy cache

A demonstrator that includes this tsTV proxy implementation has been presented and described in more detail in [21].

Conclusions

In this paper a novel sliding-interval caching algorithm for a time-shifted television service is presented. Cache decisions (on segment size, stored programs, …) at low cost distributed streamers are made after each learning interval Δ, based on popularity and distance metrics. Experimental results for a basic network topology showed promising results in terms of server and network load, especially for co-operative caching. An RTSP proxy implementation has been introduced as well. The transparent RTSP request forwarding principle for co-operative caching further reduces the server load.

Acknowledgment

This work is partly funded by the IST FP6 MUSE project. MUSE contributes to the strategic objective "Broadband for All" of IST (Information Society Technologies) and it is partially funded by the European Commission.

References

[1] Akamai. http://www.akamai.com.

[2] J. Liu, J. Xu, "Proxy caching for media streaming over the internet", *IEEE Communications Magazine*, vol. 42, no. 8, August 2004, pp. 88-94.

[3] S. Chen et al., "SRB: Shared running buffers in proxy to exploit memory locality of multiple streaming media sessions", *24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[4] Y. Chae et al., "Silo, rainbow, and caching token: Schemes for scalable, fault tolerant stream caching", *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, September 2002, pp. 1328-1344.

[5] D. Turrini, F. Panzieri, "Using p2p techniques for content distribution internetworking: a research proposal", *2nd IEEE International Conference on Peer-to-Peer Computing*, September 2002.

[6] M. Karlsson, C. Karamanolis, M. Mahalingam, "A Framework for Evaluating Replica Placement Algorithms", Technical Report HPL-2002, HP Laboratories, July 2002.

[7] T. Wauters, J. Coppens, B. Dhoedt, P. Demeester, "Load balancing through efficient distributed content placement", *NGI 2005*, April 2005, Rome, Italy.

[8] S. Gruber, J. Rexford, A. Basso, "Protocol Considerations for a Prefix-Caching Proxy for Multimedia Streams", *Computer Networks*, vol. 33, no. 1-6, 2000, pp. 657-668.

[9] E. Gilon, W. Van de Meerssche et al., "Demonstration of an IP Aware Multi-service Access Network", *BroadBand Europe 2005*, December 2005, Bordeaux, France.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications", *IEEE Infocom*, 1999.

[11] P. Backx, T. Wauters, B. Dhoedt, P. Demeester, "A comparison of peer-to-peer architectures", *Eurescom Summit*, 2002.

[12] Broadcasters' audience research board, http://www.barb.co.uk.