Tiny Network Caches with Large Performance Gains for Popular Downloads

Piotr Srebrny, Dag Sørbø, Thomas Plagemann

Tiny Network Caches with Large Performance Gains for Popular Downloads

Piotr Srebrny[1], Dag Henning Liodden Sørbø[2], and Thomas Plagemann[3]

[1] Nevion, Oslo, Norway piotrs@ifi.uio.no
[2] Bekk Consulting AS, Oslo, Norway daghso@student.matnat.uio.no
[3] University of Oslo, Oslo, Norway plageman@ifi.uio.no

Abstract. File transfers are and will in the future be responsible for a substantial part of the Internet traffic. However, with present solutions transfers of popular files lead to a lot of redundant data transfers in the network. In this paper, we investigate how a link level caching scheme can reduce the number of redundant data transfers. We serve requests from clients that download a file concurrently, but arrived at different times in such a way that they get at a given point in time the same data chunk of the file. This enables link caches to efficiently remove the redundancy. The data chunks are rearranged at the client to compose the original file. Through implementation and experimental studies we show that this approach clearly outperforms traditional file servers in terms of file server capacity and bandwidth consumption; especially when encoding the original file with fountain codes.

Keywords: link level caching, file server, fountain codes

1 Introduction

File transfer via protocols like FTP and HTTP is besides streaming of entertainment content one of the applications that dominate the Internet and causes a substantial part of the overall traffic. Increasing popularity of files leads to increasing file server load and increasing redundancy of data transfers from the file server to the clients. In order to distribute popular files and especially to handle flash crowds, P2P solutions have proven to be very useful. The core mechanism in these P2P solutions is to leverage networking and computing resources from the origin server, e.g., a torrent, and from the peers. Thus, the more concurrent downloads of a file, the more peer resources are available. This self-scaling property enables to handle flash crowds, but it leads to increasing overall resource consumption and increasing redundancy of data transfers.

In order to reduce the overall resource consumption and increase the file server performance, we investigate in this paper the feasibility of eliminating respectively reducing the number of redundant data transfers through tiny caches at the link layer. This link level caching approach, called CacheCast, has been originally designed for single-source multiple-destination applications like live streaming [8]. Due to the missing support of Internet-wide IP-multicast, many

streaming servers use unicast connections to the clients. If n clients receive a live stream, the file server sends n packets which all have the same payload. This redundancy is eliminated in CacheCast through link level caching. The streaming server sends a data element of the stream to n clients in form of a so-called packet train, which consists of one link level packet containing link, network, and transport layer headers and the data element as payload; and $n-1$ packets without the redundant payload. Furthermore, all packets are marked as CacheCast packets and contain some meta-data for cache management. At the link exit, the payload of the first packet is stored and can be used to reconstruct all following packets from the packet train. This procedure is performed individually at each link (see Section 2). The streaming server support simplifies the cache management and allows the use of very small caches that can operate at link speed. This enables a performance that is very close to IP multicast.

Creating a packet train in a streaming server is relatively easy; because the nature of single-source multiple-destination applications imposes that the same data needs to be send to all clients at "the same time". In other words, the transmission of data chunks is timely synchronized. This is different in file servers, because clients might request the same data, but not at the same time. Many clients might download at the same time a popular file, but all of those that arrived at different times at the server will receive different data chunks from the file at a given point in time. One approach to cope with the larger distance in time between redundant transfers is to increase the cache size. However, this comes at the cost of more expensive caches in monetary terms and implementation complexity, which makes it hard or impossible to operate the caches at link speed.

Therefore, we examine the use of the original CacheCast implementation to improve the performance of popular downloads. The key idea to solve this problem is based on the insight that there is no need to sequentially send data chunks of a file to clients. In contrast to streaming applications, clients downloading files consume the data only after the entire file has been received. Out of order packets can be re-ordered at the client before the file is provided to the application. This property can be used at the server to send at a given point in time the same data chunk to clients that arrived at different times. Thus, a packet train can be send over the link and very small caches are sufficient. We show in this paper that fountain codes can be used to alleviate the CacheCast file server implementation. A file is a priori encoded and chunks of the encoded file are sent to the clients. Clients just need to receive data chunks until they have sufficient information to decode the file. With an implementation in ns-3 and extensive simulation studies we show that this solution clearly outperforms traditional file servers, both in terms of file server capacity and of bandwidth consumption. The more concurrent downloads, the more redundancy we can remove. As such this solution is self-scaling like P2P, but not at the cost of peer resources.

The remaining of the paper is structured as follow: the idea and basic functionality of CacheCast is presented in Section 2 and 3. The results of our extensive evaluation are presented in Section 4 and Section 5 concludes this work.

2 CacheCast Basics

The server architecture relies on the CacheCast mechanism to deliver the same data chunk to multiple destinations. Therefore, it is essential to understand how applications benefit from CacheCast, and how CacheCast transports data.

CacheCast is a system of packet caches operating on network links. A single cache consists of two processing elements that are installed at the link end-points. The element installed at the link entry is called Cache Management Unit (CMU) and the element installed at the link exit is called Cache Store Unit (CSU). The CMU keeps a short record of payloads that have been recently transmitted over the link. Similarly, the CSU keeps a record of recently received payloads and maintains the consistency of these records[4]. The CMU inspects packet payloads immediately before transmission. If the CMU finds the packet payload in the record of the recently transmitted packets, it substitutes the payload with a short unique identifier. Thus, only the packet header with the identifier is transmitted over the link. Upon receiving the packet on the link exit, the CSU uses the identifier to find the payload in the local record and to reconstruct the packet. The reconstructed packet is processed further in the standard way on the router.

The operation of link caches is transparent to the traffic above the link layer, i.e., the standard IP network and an IP network with link caches provide the same functionality. The IP network with link caches can transport much more efficiently the same data from a single source to multiple destinations. This is achieved by suppressing redundant payload transmissions over the network links. Since the link caches are transparent to the IP layer and above, network hosts can communicate using the standard IP based transport protocols such as UDP, TCP, or DCCP.

Link caches are designed to operate in the Internet infrastructure which is based on fast links transporting large amounts of data. To achieve high efficiency at low implementation costs, the link caches process only packets that are part of single source multiple destination data transfers. This type of transfer creates at the link layer a burst of IP packets that have different headers, but carry the same payload which can be processed very efficiently by link caches. When traversing a link with a link cache the burst of packets resembles a packet train where the first packet in the train carries payload while the remaining packets are truncated by the CMU to the header size. In order to guarantee the minimum time span of the packet burst and consequently to minimise link cache resource requirements, CacheCast introduces a new system call *msend* to the OS. The application uses the system call to send data to multiple destinations. The *msend* API resembles the standard POSIX *send* system call with the difference that *msend* takes a set of file descriptors as the input instead of only one file descriptor taken by the *send* system call. The *msend* system call operates only on the file descriptors, which are referring to network connections.

[4] The records can be temporally inconsistent due to link transmission errors. For further information please refer to [9].

The choice of transport protocol invoked by *msend* is important, because for CacheCast we require that message boundaries are preserved when passing messages through the protocol layers. Furthermore, one of the approaches we investigate is based on the idea to not send blocks from a file in their given order, but instead adjust this order to the needs of clients. As such TCP cannot be used and we prefer DCCP over UDP since DCCP includes rate control.

3 CacheCast Server Architecture

A file server has to perform two basic tasks: *File Selection* and *File Transmission*. The file selection is initiated by a client, which selects one file from the server repository using a command set provided by the file transport protocol (FTP). The *File Transmission* is the underlying functionality which transfers the file from the server to the client. We assume that the CacheCast file server implements the *File Selection* functionality of the standard FTP server. Thus, the FTP server and the CacheCast file server differ only in how the *File Transmission* is handled. In the standard FTP, the client selects the file to download and tells the server to initiate the file transmission. The server starts forwarding the contents of the file to the client over a TCP connection. The default transfer mode in FTP is Stream mode [7]. When it is enabled FTP sends the data as a sequential stream. TCP ensures that the file is transferred correctly by dividing the data into segments, assigning sequence numbers to these segments, and issuing retransmissions when segments are lost in the network. By using the sequence numbers, TCP assures that all received segments are correctly ordered. TCP also adjusts the transmission rate to the client's available bandwidth.

The CacheCast server is designed to distribute efficiently popular files. Therefore, before we discuss how it solves the problem of reliable file transfer and transmission rate control, first we present how it achieves high efficiency when delivering the same file to multiple clients.

3.1 Synchronous Transmission

An FTP server is a multi user system, i.e. it has support for multiple clients downloading files concurrently. A client can connect to the server at any time and request any file. FTP is designed for single source, single destination transfer. Thus, for each client connected to the server there is one TCP connection. Each client is served separately on each unicast connection. When multiple clients are downloading the same file at approximately the same time, there are overlapping time periods between the download procedures. During these periods the same file is transferred to multiple clients at the same time. When the request rate to a file server is high as in the event of a flash crowd many clients connect to the server within a small time interval, creating multiple overlapping download time periods as shown in Figure 1. Within these time periods clients are downloading the same file concurrently; however, they are receiving different parts of the file, since TCP transmits data sequentially.

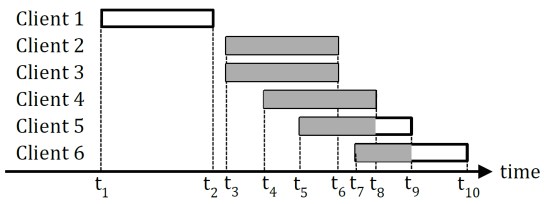

Fig. 1. Multiple clients with different arrival times downloading the same file

In order to optimize the file transmission within the overlapping time periods using CacheCast, it is necessary that individual file blocks are transmitted synchronously to all clients. However, synchronous transmission of blocks to many clients is difficult to achieve due to different arrival times of clients and even if sufficiently enough clients arrive close enough to each other in time they typically will drift apart from each other. The reason is that the available bandwidth on the paths to the clients can differ substantially.

The core idea of our approach is based on the insight that clients use the content first after it has been entirely received. The order in which the blocks of the file are sent and whether the blocks are encoded before transmission or not, is not of relevance for the client. The only important aspect for the client is that the entire file is recreated before it is passed to the client. Therefore, the server can send those blocks that are needed by several concurrent clients to achieve synchronous transmission. In particular, we have investigated two block selection schemes to achieve synchronous transmission, namely, block-by-block transmission and fountain code transmission. In addition to the block selection, rate control and end-to-end reliability has to be supported.

3.2 Rate Control

The rate control works identically for both block selection approaches. For a group of clients downloading the same file, the rate control aligns the transmission rate to the fastest client in the group. Hence, this client can take full advantage of its downlink speed which is not limited by slower clients. Since slower clients are not able to receive all blocks at the selected rate, the congestion control algorithm in DCCP will drop some packets to these clients at the sender side (i.e., it will not be passed to the network layer). With this approach all clients download the file with their individual maximum available speed.

In order to determine the fastest client in the group, the rate control uses status information returned by DCCP after block transmission attempt. The status tells whether the packet has been dropped or sent. By keeping a record of the last N transmission attempts for each client it is easy to identify the fastest client, which is the client with most packets sent. The parameter N can be used to control how fast rate control should adapt to changes. For the results presented in this paper we use $N = 100$. We use the Additive Increase

Multiplicative Decrease algorithm to (1) adapt the sending rate to the available bandwidth to the fastest client, and (2) achieve fair share for multiple flows over contented links.

$$r = \begin{cases} r + I & \text{if last packet was set} \\ r/D & \text{if last packet was dropped} \end{cases} \quad (1)$$

The transmission rate r is increased with factor $I > 0$ or decreased by factor D $(0 < D < 1)$. We start with an initial transmission rate of $r = 64$ kb/s. The transmission of subsequent blocks is scheduled to match the current rate r.

3.3 Block Selection and Reliability

The block-by-block transmission approach requires from the server to keep track of the blocks each client is missing at any point in time. Thus, the initial state for all blocks for a newly arrived client is MISSING. This information is used to decide to which clients each block should be sent. There are several policies to determine which block should be sent, including most wanted block and round robin. Due to its simplicity we have chosen to implement round robin in our current prototype. With this policy the block selection algorithm selects the blocks from a file in sequential order from the beginning to the end of the file and starts again at the beginning as long as there are clients downloading the file. The selected block is only sent to those clients that are missing it, i.e., the blocks state is MISSING. To achieve reliability retransmission is used. When sending a packet to a group of clients, the server starts a transmission timer for this packet and sets for each client the status of the bock to SENT. The client sends for each received packet an acknowledgement including the sequence number of the packet. If the server receives the acknowledgement before the timer expires it sets the status of the block to RECEIVED and otherwise to MISSING. Retransmission of missing blocks will then naturally happen in the next round when the block is selected again. Negative acknowledgments cannot be used, because the client cannot know which packets have been lost in the network and which ones are dropped by DCCP. One artifact of all block-by-block transmissions is that the last remaining clients can miss disjoint sets of blocks. Thus, it is no longer possible to achieve synchronous transmission.

Block selection and reliability is substantially simpler with the fountain codes [5]. With fountain codes files are encoded in such a way that any block can be used to reconstruct the original file. As such, this scheme provides Forward Error Correction. Thus, adding reliability to DCCP comes for free and all blocks are useful for all clients. No book keeping of packet states, timers, and acknowledgements are needed, a client just receives encoded packets until it is able to reconstruct the entire file. The overall architecture of our prototype with the two approaches is illustrated in Figure 2. The feedback from each DCCP instance is used to determine the sending rate and to call at the proper point in time the *msend()* system call with a pointer to the selected block and the file descriptors for those clients that should receive the block.

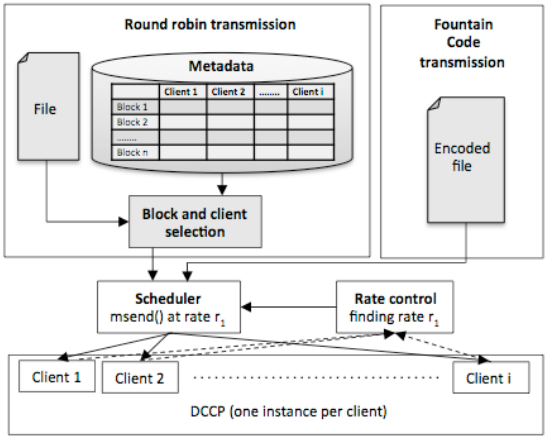

Fig. 2. Architecture of prototype implementation

4 Evaluation

We have implemented the above described architecture of the CacheCast file server in ns-3 and performed extensive simulation studies to evaluate it. Since this work aims at the scalability of file servers we focus our study on the outgoing link from the file server, later on called server uplink. The performance advantages of batching several client requests and sending them from a server in form of a packet train are documented in our earlier work through simulation and real world implementation. Furthermore, we have shown in [9] that CacheCast can achieve network-wide close to IP multicast performance and that it can be with great benefits incrementally deployed in the Internet.

Considering the fact that in modern networks congestion occurs at the network edges, i.e., on links attached to the server and the clients, we model those links and ignore the intermediate links. This leads us to a simple topology in which a file server is connected to a router, which in turn has n links to n clients. The bandwidth between the router and the clients is based on measurement results from [4] and ranges between 64 kb/s and 5000 kb/s. It is distributed in this range in six groups as described in Table 1.

To study the scalability of the CacheCast file server, we set the bandwidth of the link between file server and the router to 10 Mb/s such that congestion occurs. The end-to-end propagation delay between the file server and the clients is uniformly distributed between 30 ms and 50 ms, which correspond to a medium sized ISP network. The workload in the experiments consists of a file with the size of 18 MB that is downloaded by clients arriving at the server at a rate of eight clients per second (these numbers are based on download statistics of the VLC media player). We use a constant arrival rate to simplify the analysis of the results. The insights gained with these settings are also valid for more advanced arrival rate distributions, such as Poisson and Zipf distribution. The

client number is set to 100 in all experiments besides when measuring system fairness where the client number is specified separately. All experiments are performed multiple times with different seeds to the random generator. We present the average over these runs and when appropriate the standard deviation.

In the following sections, we describe experiments and results of the rate control mechanism, download time, bandwidth consumption, fairness, effect of small file size, and effect of bandwidth overprovisioning on the first link.

4.1 Rate Control

The goal of the CacheCast server is to provide to all clients the shortest possible download time. Therefore, we evaluate how well the rate controller is able to adapt the transmission rate to the available bandwidth to the fastest client.

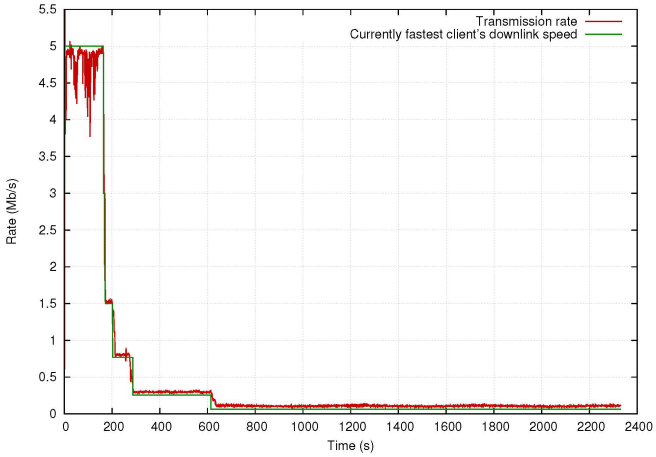

Fig. 3. Transmission rate and currently fastest client

Figure 3 relates the transmission rate determined by the rate controller with the actual download speed of the fastest client over time, i.e., the ground truth. The transmission rate is, as expected, reduced over time since the fastest clients finish first and the slowest last. Furthermore, the transmission rate aligns very well with the currently fastest client. This is further supported through the data in Table 1 in the row labelled "Last client finish time" which presents the time at which the last client in each downlink speed group finishes.

Looking at the length of the time interval for the different transmission speeds we can distinguish three phases. The first one is defined by the time it takes until the last client with 5 Mb/s downlink speed finishes. In the second phase, the client groups with 3 Mb/s and 1.5 Mb/s downlink speed finish very briefly after the 5 Mb/s group. This can be attributed to the fact that those clients could substantially benefit from the data send during the first phase. In the third

Downlink speed [kb/s]	64	256	768	1500	3000	5000
Client share	2.8%	4.3%	14.3%	23.3%	18%	37.5%
Last client finish time [s]	2339.4	622.7	269.3	198.6	161.9	157.8
Single TCP download time [s]	2491	623.5	208.7	106.7	53.7	32.6

Table 1. Client downlink speed distribution and download performances

phase, the slower clients require more and more time to download the entire file and as such the transmission rate is reduced in increasing larger steps.

4.2 Download Time

We use the same settings as in the previous experiment and compare the distribution of download time for all clients served with (1) a traditional FTP server, (2) CacheCast server with original block transmission, and (3) CacheCast server with a fountain encoded file. To achieve comparability, we related the measured download times to the time it would take a single client to download the file with a TCP connection when accessing all server resources exclusively, i.e., no other clients are present. The download times for this exclusive download (called "single TCP connection") are given in Table 1 in the row labelled "Single TCP download time".

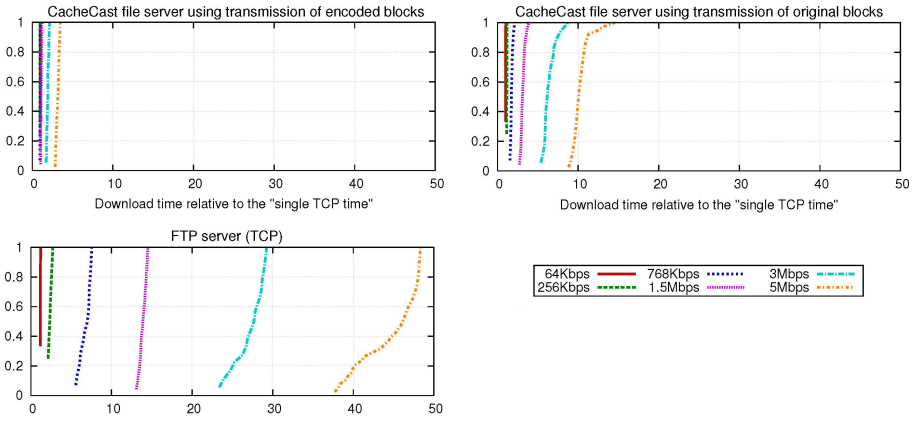

Fig. 4. Download times for FTP server and CacheCast file servers

Figure 4 shows the CDF of the download times for clients grouped by downlink speed. The relative download time increases for all servers with the downlink speed, because fast clients are more affected by congestion than slow clients. Both CacheCast implementations clearly outperform the FTP server due to the reduction of redundant transfers over the first link. There is also a clear difference

in performance for the faster clients in the two CacheCast implementations, i.e., transmitting encoded blocks leads to much shorter download times.

In the following studies, we focus only on the comparison of the fountain code based CacheCast server and FTP server.

4.3 Bandwidth Consumption

One important metric for the content provider is the bandwidth consumption, especially for the server uplink. The less bandwidth is consumed per client, the more scalable the server is and less costs incur. Figure 5 illustrates the bandwidth consumption on the server uplink over time for the CacheCast and FTP servers. Both consume in the beginning of the experiment the entire bandwidth.

This seems to be contradicting to the results of the rate adaptation evaluation in Section 4.1, which show that the maximum transmission rate in the beginning is very close to 5 Mb/s. However, the transmission rate shows only application layer throughput to a single client while the uplink bandwidth consumption shows the resulting network traffic that carries also packet headers to all receivers. The period in which the bandwidth of the link is fully consumed is substantially shorter for the CacheCast server, since the total amount of data sent over the link is much smaller due to the redundancy reduction.

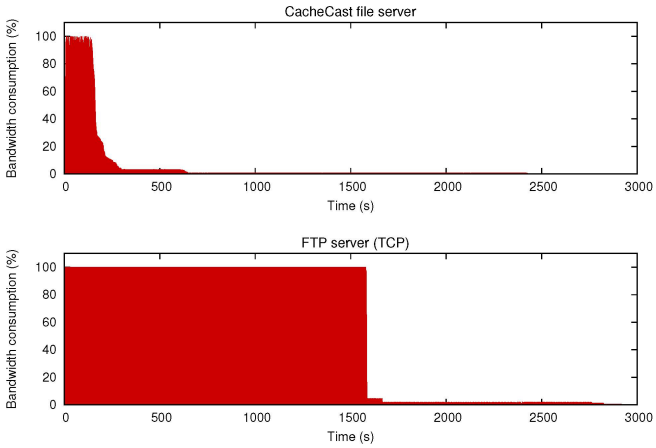

Fig. 5. Bandwidth consumption on the first link

4.4 Fairness

The previous experiments focused on transmission of a single file to multiple clients. In this experiment, we analyse how the CacheCast server distributes the uplink bandwidth capacity between two groups of clients that download different

files. The CacheCast server uses DCCP, which ensures fair bandwidth sharing among concurrent data streams in the network. Since the rate controller is build on the feedback from DCCP, the core issue investigated in this experiment is the question whether it preserves the fairness achieved by DCCP.

To simplify the presentation and analysis of results, we study in this section two groups of clients that download each one file. We study how the number of clients impact fairness in experiments with 10 and 200 clients, and with different group sizes. For the 10 clients the two groups comprise of 5/5, 6/4, 7/3, 8/2, and 9/1 members; and for 200 clients the groups comprise of 100/100, 80/120, 60/140, 40/160, and 20/180 members. The download speed of all clients is set to 10 Mb/s to (1) enable each individual group to consume the entire bandwidth on the first link, and (2) to easily compare the bandwidth share. We measure the end-to-end throughput for each client of the groups and show the aggregated throughput per group. The measurements are performed when no clients are arriving or leaving the CacheCast server.

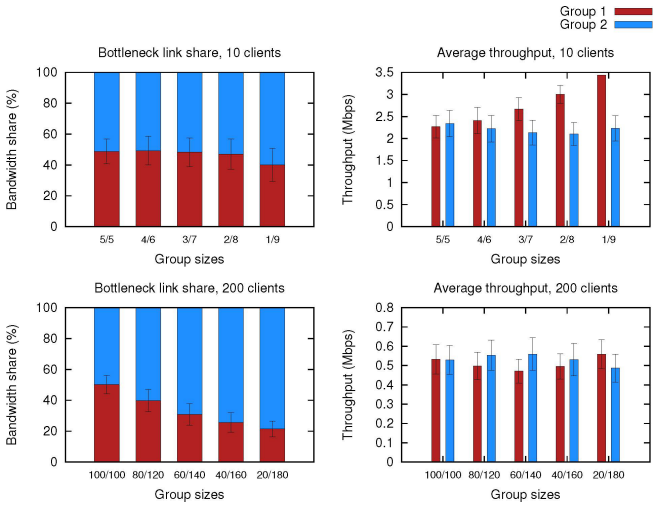

Fig. 6. Bandwidth share on first link and end-to-end throughput

The average bandwidth share on the server uplink and the average client throughput group is shown for the two client populations in Figure 6. A small client population leads to increasing unfairness in the uplink utilization and end-to-end throughput as the group sizes differ more. However, the results from the experiment with 200 clients show that with larger groups the system achieves more even resource utilization per client. The nature of CacheCast that the ratio between the number of packets sent and the number of bytes sent is unequal leads to the small diversion from the perfect fair share.

4.5 Effect of Small Files

The performance gains with CacheCast are achieved by removing redundant payload in packets that are sent during a short time window. Either block reordering or fountain codes is used to send to all concurrent clients the same data block. The more clients concurrently download the same file the higher the performance gains through redundancy removal. This number of clients depends on the arrival rate and the time they spend to download the file. Given an arbitrary arrival distribution, the less time the clients need for the download the less clients download the file at the same time. The download time is determined by the file size and the available bandwidth between client and server.

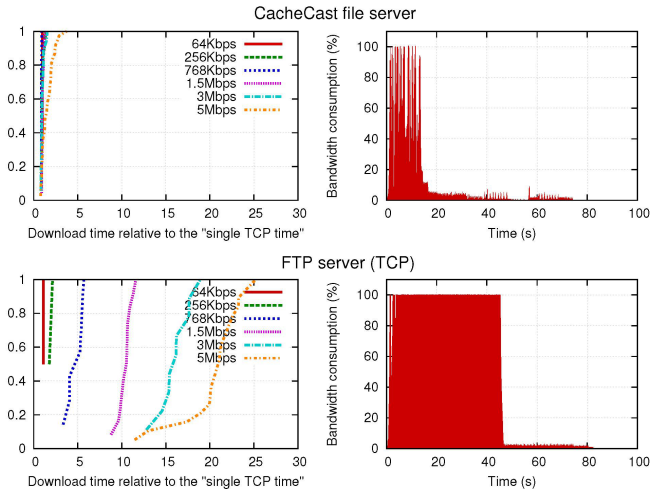

Fig. 7. Download time and bandwidth consumption for 500 kB file

To illustrate this effect, we show in Figure 7 an example of the download time and bandwidth consumption for a small file (500 kB). The decreased file size results in less overlapping, so there is less redundancy to remove. The performance gain of the CacheCast file server has decreased compared to the 18 MB file. The average packet train length has decreased from 14 to 6.6, thus the CacheCast file server is not able to benefit as much from CacheCast as in the original experiment.

4.6 Effects of Bandwidth Overprovisioning

So far all experiments have studied cases in which congestion occurs on the first link. To study also the effects of overprovisioning of bandwidth on the first link we set the bandwidth of this link to 300 Mb/s. The results presented in Figure 8 show that the clients of the FTP and CacheCast file server experience similar

download times. This is obvious, since there is no need to remove redundant network traffic if the network is overprovisioned. However, it shows also that CacheCast leads to a much lower bandwidth consumption on the first link.

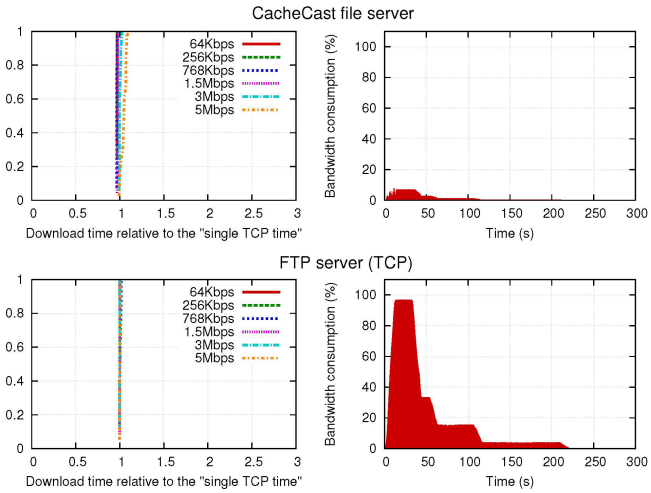

Fig. 8. Download time and bandwidth consumption for 300 Mb/s first link

5 Conclusions

This paper presents an approach to increase the scalability of file servers through link level caching with CacheCast. The main difference between the original application domain of CacheCast and file servers is that clients do not all arrive at the same time. This means that they will in a traditional file server be served with different data blocks at any given point in time, which in turn does not allow to create packet trains with the *msend()* system call. The core idea of our solution to this is to re-arrange the data blocks such that the individual data blocks are send with *msend()* to several clients. We implemented and evaluated one version based on round-robin distribution and one on fountain encoded files. The main conclusions from the evaluation are: (1) both approaches can substantially reduce the download time for clients and the bandwidth consumption of the first link, (2) the version with fountain encoded files clearly outperforms the round-robin version, mainly due to problems related to rate control, and it simplifies management tasks of the server; (3) clients of a CacheCast server get a fair bandwidth share; and (4) small files and bandwidth overprovisioning reduces the performance gain of CacheCast over FTP, but still provides benefits in terms of shorter download times (for small files) and lower bandwidth consumption. Thus, if there is more than one client downloading the same file, CacheCast

increases the scalability of file servers, both on the server itself through the *msend()* call and the first link; and it reduces traffic related costs. The more concurrent clients the higher the gain without relying on other resources than the host and CacheCast enabled routers.

This paper presents the first study to use CacheCast for asynchronous file transfers. The use of fountain codes has been inspired by previous work combining fountain codes with IP multicast [6]. Many early works in this area assume that all clients have the same amount of bandwidth available. Byers et al. address this unrealistic assumption through layered multicast with several multicast groups [2]. The clients are responsible to subscribe to an appropriate subset of these layers. However, the leave latency of the Internet group management protocol makes it very hard to efficiently adapt to changing bandwidth. The strategy of dynamic layering is introduced in [1] to avoid this bottleneck. However, this results in unfair bandwidth sharing with TCP, especially when the drop tail queue size increases. Gill et al. [3] introduce a client work ahead policy to determine how reception bandwidth is allocated among the layers to protect again short-term bandwidth fluctuations. The major difference between our approach and previous works is the strength of CacheCast to achieve near IP multicast performance and maintain at the same time the end-to-end relation between client and server. In this way DCCP can adjust quickly transmission rate for each individual client to achieve full TCP friendliness.

References

1. Byers, J., Horn, G., Luby, M., Mitzenmacher, M., Shaver, W.: Flid-dl: congestion control for layered multicast. Selected Areas in Communications, IEEE Journal on 20(8), 1558–1570 (Oct 2002)
2. Byers, J., Luby, M., Mitzenmacher, M.: A digital fountain approach to asynchronous reliable multicast. Selected Areas in Communications, IEEE Journal on 20(8), 1528–1540 (Oct 2002)
3. Gill, P., Shi, L., Mahanti, A., Li, Z., Eager, D.L.: Scalable on-demand media streaming for heterogeneous clients. ACM Trans. Multimedia Comput. Commun. Appl. 5(1), 8:1–8:24 (Oct 2008), http://doi.acm.org/10.1145/1404880.1404888
4. Huang, C., Li, J., Ross, K.W.: Can internet video-on-demand be profitable? SIGCOMM Comput. Commun. Rev. 37(4), 133–144 (Aug 2007)
5. MacKay, D.: Fountain codes. Communications, IEE Proceedings- 152(6), 1062–1068 (Dec 2005)
6. Mitzenmacher, M.: Digital fountains: a survey and look forward. In: Information Theory Workshop, 2004. IEEE. pp. 271–276 (Oct 2004)
7. Postel, J., Reynolds, J.: File Transfer Protocol. RFC 959 (Standard) (Oct 1985), http://www.ietf.org/rfc/rfc959.txt, updated by RFCs 2228, 2640, 2773, 3659, 5797
8. Srebrny, P., Plagemann, T., Goebel, V., Mauthe, A.: CacheCast: Eliminating Redundant Link Traffic for Single Source Multiple Destination Transfer. In: Proceedings of the 2010 30th IEEE International Conference on Distributed Computing Systems (ICDCS). IEEE Computer Society (June 2010)
9. Srebrny, P., Plagemann, T., Goebel, V., Mauthe, A.: No more déjà vu - eliminating redundancy with cachecast: Feasibility and performance gains. Networking, IEEE/ACM Transactions on 21(6), 1736–1749 (Dec 2013)