Reducing Server and Network Load with Shared Buffering

Somaya Arianfar, Pasi Sarolahti, Jörg Ott
Aalto University, Department of Communications and Networking
{Somaya.Arianfar, Pasi.Sarolahti, Jorg.Ott}@aalto.fi

ABSTRACT

Today, content replication methods are common ways of reducing the network and servers load. Present content replication solutions have different problems, including the need for pre-planning and management, and they are ineffective in case of sudden traffic spikes. In spite of these problems, content replication methods are more popular today than ever, simply because of an increasing need for load reduction. In this paper, we propose a shared buffering model that, unlike current proxy-based content replication methods, is native to the network and can be used to alleviate the stress of sudden traffic spikes on servers and the network. We outline the characteristics of a new transport protocol that uses the shared buffers to offload the server work to the network or reduce the pressure on the overloaded links.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

Transport protocols, Resource management, Caching

1. INTRODUCTION

The Internet carries an ever-increasing amount and a wide diversity of traffic types. To keep up with the growing traffic, servers and network capacity need to be scaled up. The required capacity is usually determined by the highest *expected* traffic surge. However, unpredictable traffic surges may lead to congestion situations for the network or servers.

Unpredictability is not a rare phenomenon in the Internet. Traffic surges have happened before and they are happening now, growing both in number and size [11]. Many such traffic surges occur when a content item suddenly becomes popular and needs to be transferred to many clients in the absence of multicast distribution. Examples of such spikes could be the result of a tweet by someone with many followers[1] or mentions of web sites in the news.

[1]See: *http://www.techradar.com/news/internet/how-stephen-fry-takes-down-entire-websites-with-a-single-tweet-674170*

Fixed capacity servers and (access) networks are not able to handle sudden spikes of traffic on their own. One of the main mechanisms used to help in such situations is content replication, e.g., in the form of CDNs. In the CDN model of content replication, servers benefit from pre-planned content proxies through CDN agreements [14]. The majority of content providers, however, do not have CDN agreements. For their servers, sudden surges of traffic are very rare, but when they happen they may cause long periods of unavailability. Even servers with CDN agreements are not immune against the effects of sudden surges of traffic. Periods of server unavailability may happen as a result of the time it takes for an overloaded server to switch to less loaded CDN nodes or as a result of the time that CDN nodes require to adjust their resources [13]. Similar delayed reactions apply to other methods of load reduction such as cloud infrastructure [13].

More traditional forms of content replication as in network proxy caches [6] are not targeted at helping servers and thus cannot be specifically activated and used in case of server overload (servers won't even know of their existence). Even in case of network load reduction, such caching proxies easily loose their efficiency with increasing request rate and breadth of (unpopular) traffic.

A more fundamental issue regarding many of the content replication methods is their proneness to violate the end-to-end principle. Content replication below the application layer or in the form of transparent proxy caching is not necessarily in line with the end-to-end argument. Allowing a node in the middle of an end-to-end connection to send a content item or its subparts to a client causes issues regarding security, efficiency, and rate adaptation [9].

Despite the problems with different content replication methods, their usage is pushed even further into the network. A number of different proposals that treat content replication as a fundamental part of the network is growing. For example, proposals such as DECADE [2] or transparent caching for streaming files [9] try to benefit from in-network storage in a more native way compared to the overlay-based models discussed above. The new methods, however, remain application specific and limited.

In this paper, we propose an approach to reduce the effect of sudden traffic surges on servers (and thus implicitly on the network), especially for providers that cannot afford CDN agreements. We introduce an end-to-end friendly stream buffering mechanism in which content replication is a native part of the design rather than an exception to it. Our design, as we will explain later, essentially extends the server's local memory to the network for the purpose of content replication. We also outline a strawman for a new transport protocol that would interact smoothly with the buffering nodes, without being constrained to a specific (group of) application(s). This transport protocol can use the stream buffers to offload the workload of stressed servers to the network and thereby also assist in reducing the network load.

2. CONTENT REPLICATION PRINCIPLES

Today's load reduction mechanisms based on content replication are external to the IP network. They are targeted at balancing predictable load, but are neither meant nor effective for alleviating the effect of sudden traffic spikes. In this paper, we propose a load reduction mechanism that utilizes active in-band content replication. Unlike other methods, we target a general-purpose network service with a focus on reducing the effect of sudden traffic spikes in short periods of time. Before turning to the details, we discuss the guiding principles for our design to emphasize those functional elements that make our proposal different from others.

Proper communication with the content replication nodes on caching decisions. Proxy caches cannot directly help the servers in case of overload. Additionally, normal ISP-level caches require diverse intelligent algorithms to second-guess which data is worthwhile caching. Because there is no clear way of deciding which traffic to the cache, such autonomous cache algorithms lose their effectiveness with the increased traffic rate and diversity. In an effective content replication system, a server should to be able to communicate with the content replication node at the time of overload and specify which content is worth caching so that the replication system can operate effectively with limited resources.

Minimal extra delay and management overhead. CDNs are one of the effective ways of reducing the server load. However, the need for pre-agreement with the CDNs can undermine their benefits when it comes to unexpected traffic spikes. An effective content replication and retrieval model needs to be light and fast.

Application independence. As the diversity of applications grows, so does the diversity in what causes traffic surges and thus their unpredictability. It would be cumbersome to design and deploy separate distribution techniques for different kinds of applications and making all applications use a single protocol (such as HTTP) may not be workable or efficient either. Therefore, a content replication and retrieval mechanism needs to be application-independent.

No limitation due to the application level object size. As the Internet is increasingly used to transfer audio and video, the size of the files to be replicated also grows. Large files or long media streams would take a long time to replicate, and live feeds may not have a known size in advance, so that application-level caching of entire objects is not applicable. Therefore, the traditional object-based hit/miss in the cache is not meaningful to an important set of applications. The content replication and retrieval mechanism needs to act independent of the application-level object size.

No limitation because of segmentation. Since we no longer rely on caching complete objects, file segmentation is used for effective replication [9]. The segmentation process can be quite inefficient and slow depending on the type of the application and segment size. For example, bigger segments of objects are preferable for limiting the number of requests that an application sends for an object, but at the same time bigger segments reduce hit probability in the cache and make cache management more difficult. An effective content replication and retrieval method should thus act independent from the segmentation process.

Reduce server load using content replication. There are many techniques today that try to reduce the load and traffic redundancy below the application layer, e.g., EndRE [1] and packet caching [3]. Such methods mainly focus on reducing the bandwidth usage, but do not significantly reduce the server work load. End-host based mechanisms such as EndRE [1] may even add to the server's workload by requiring complex computations. A useful content replication and retrieval mechanism for the purpose of reducing the server load should not add but reduce the server load.

No constraints from the underlying networks. Application-independent content replication sometimes is interpreted as caching independent object items as packets, e.g., in some caching proposals for information-centric networking [4]. While such mechanisms may reduce the server load, they are bound to the packet as a data unit and thus cannot easily address dynamic path characteristics (e.g., MTU sizes, unaligned packet boundaries). A content replication and retrieval model needs to avoid this limitations.

Individual or subsets of the above principles have been addressed in today's Internet or at a conceptual level in different proposals for information-centric networking, but we are not aware of a comprehensive approach taking all of them explicitly into account.

3. DESIGN COMPONENTS

In most operating systems, including Unix-like OSes and MS Windows, a common I/O abstraction for files, inter-process communication, and (to some extent) memory access is a byte stream, reflected in basic system calls, pipes, etc. This abstraction is application-independent (the applications add their own interpretation on top) and useful for dealing with files/streams of (yet) unknown sizes. A similar abstraction is also offered by the socket API for TCP: it also uses byte streams to send and receive data to/from the network. Packet boundaries are only introduced as a necessity of packet networks, but the exchanged messages and content items have their own demands on structure and size. Therefore, it is beneficial to be able to buffer and replicate the content in the network in a same way as they are accessed and sent by the server: as byte streams without predefined bounds or ends.

We propose adding a (byte) "stream buffering" capability to the network. Conceptually, this is a memory-based shared buffering approach that extends the local system memory to the network and reduces the redundant traffic. Using such shared buffers is invoked by a server on demand. To be able to prevent over-utilization, shared buffer usage by individual servers is controlled using an end-to-end load control architecture based on explicit signaling between the end hosts and the buffer nodes. We discuss the elements of our design in the following.

3.1 Stream buffering

Stream buffering is a way for routers to buffer data such that multiple destinations or connections can re-use the same data. Stream buffers are the core enablers in offloading transfers with multiple (simultaneous or time-shifted) receivers. Stream buffering has strong similarities to caching, with a couple of major differences: stream buffers do not cache application-layer objects or network packets, but just opaque (pieces of) data streams taken from the transport payload as identified by an identifier carried in the transport headers. Stream buffers are equipped only with very simple (and fast) stateless operations for storing and fetching data based on incoming packets. The additional intelligence comes from the interactions with the sender and the load control architecture, as will be explained later in this section. The concept of stream buffers is more similar to shared memory in Unix system than the traditional web caches of packet-level caches as discussed in some recent research [3]. To support stream buffering, a new transport protocol is needed, as we will discuss in more detail in the next section.

The basic enabler for shared stream buffering is proper byte stream identification. With TCP, a byte stream is only meaningful in the context of an end-to-end flow. End-to-end identification of byte streams makes it difficult to share the (contents of a) byte streams across different connections because a flow uses random initial

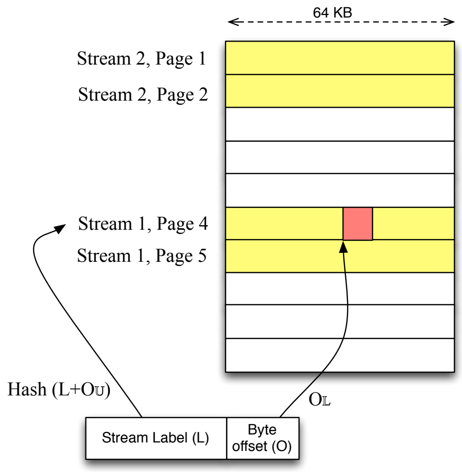

Figure 1: Stream buffer structure

sequence numbers, several objects (including often variable size headers) are sent within the same sequence number space, and each object is *implicitly* identified within the message exchange.

Therefore, as a primary element of our design we use content-based stream identifiers to identify different byte streams. Such a stream identifier could simply be the same as an application level file name or an specific encoding of that. For example, the identifier can be built by hashing the URI and content modification timestamp in HTTP, based on the piece index used in BitTorrent, or simply calculating a hash over the content.

Stream buffers assume that the transport headers come with a pair of fields (L, O) identifying the data in the payload of packet. L is the stream label derived from the byte stream identifier, and O is the stream offset, similar to a TCP sequence number, except that it is not random and relative to the beginning of each transmitted content item. Unlike TCP, we assume that 64 bits can be used for the offset, allowing maximum of 16 EB of data for one label. Should this not suffice, larger streams can be split and use multiple labels. These two fields are used for placing the data in the memory of a stream buffering router or other network intermediary.

The memory of a stream buffer is organized as a hash table, that stores memory pages of B bytes. The choice of B is an implementation detail for the buffer and could be optimized for the file size distribution; as an example, we use 64 KB as the page size, because it allows easy operations with 16 bits addressing the data within the page. The page size does not need to be standardized, and different buffer nodes may choose different page sizes independently.

As shown in Figure. 1, when a packet arrives at a stream buffer, it chooses the memory page as $Hash(L+O_U)$ and offset within page from O_L, the lower part of offset based on the page length (e.g., the lowest 16 bits). With 64-bit offsets, O_U therefore stands for the upper 48 bits of the offset. Then the transport payload is copied to this position in the memory. Note that, at this point, we do not apply any special heuristics for deciding about cache insertion or replacement policies: any new packet can be accepted and replace an old one if they are both assigned to the same location in memory.

A set of new arriving bytes could replace buffered ones, if $Hash(L_{new} + O_U)$ points to a page that has already been used by another stream. We note that each page contains the necessary meta-data about the stream including the stream identifier to which the page belongs and the valid byte range. Erasing a complete page and updating its meta data occurs with the arrival of the first bytes of a new stream to that page. Events such as packet loss or packet reordering may cause unused "holes" to appear in the stream buffer. Therefore the page-specific meta-data keeps track of ranges of valid bytes in each page.

3.2 Store-me Bits

Our shared buffering model allows for identifying and organizing streams of bytes at the transport level. It is known that only a fraction of the traffic that enters a content replication node is going to be transmitted repeatedly [7]. In our case, the volume of the bytes that arrive at a buffering node could easily be in the order of gigabits per second. In such situations, the efficiency of the replication system may easily suffer if the buffering nodes end up filling their memory with content that won't be accessed again while stored. To alleviate this problem we propose using the stream buffers in combination with *store-me* bits in the packet headers.

A server that is sending the same byte stream repeatedly within a short period of time, is the first entity to notice the potential redundancy at the stream (transport) level. At the server, these redundant transfers incur load and, e.g., in case of a flash crowd, may lead to temporary overload. Therefore, the server itself can assist buffering nodes in deciding which streams of byte are worth buffering by giving corresponding hints: the server sets *store-me* bits in the transport headers during its overload period.[2] The stream buffering nodes on the path from the server to different clients will only consider buffering those bytes that are marked with the store-me bit. In this way, the amount of traffic that the buffering node has to deal with is reduced to a small proportion of what could have entered the memory otherwise.

Servers sending store-me bits will instantiate state in buffering nodes and thus consume resources. Buffering nodes may apply local admission and replication policies when seeing new packets with store-me bits set. Stored data is available in the buffer for a limited time, to avoid delivery of out-dated content. We consider the shared buffer mainly as a short-term storage used to relieve load spikes, with lifetimes on the order of minutes rather than hours.

3.3 Load Control Architecture

Putting a server into a position to allocate downstream resources obviously raises the issue that every server could simply set the store-me bits for every stream, which would lead to buffer pollution and the buffering nodes would be back to second-guessing which streams to store. Therefore, we need a mechanism that penalizes for excess store-me requests and gives incentives for the senders to be selective about when to send them. In the following we outline such load control architecture, loosely inspired by Explicit Congestion Notification and the Re-feedback architecture [5].

In addition to the store-me bit, the packets contain an "*overload*" bit that is always initialized to zero by the sender. If the rate of incoming store-me bits at a stream buffer exceeds an threshold beyond which the buffer node considers itself overloaded by the store-me requests, it sets the overload bit in the packet. The state of the overload bit is echoed back in an "overload echo" to the sender in the packets flowing to the reverse direction. As the sender receives overload echoes, it reduces its allowed rate of outgoing store-me bits, similarly as TCP reduces the congestion window in response to congestion notifications.

In addition, there can be a policer near the sender (e.g., run by a network operator) that monitors the overload echoes and store-me bits for each source, and clears any excess store-me bits that the sender might try to send. Because the policer would reset the bits from more or less random packets leading to reduced caching efficiency, the sender has the incentive to remain within the allocated

[2]We assume that server farms can be organized such they share load information.

quota of outgoing store-me bits, and retain the control over which pieces of data are more important to store, e.g., based on popularity.

A possible underlying business scheme would be that an ISP agrees with an application service provider a basic rate of store-me bits, possibly with the option of getting more at an extra cost once they exceed a threshold.

3.4 Receiver-based Transport Protocol

We have discussed how the buffering node can identify and write the re-usable bytes to the stream buffer. Accessing such a stream buffer below the application layer, nevertheless, needs re-visiting the underlying transport protocol logic. We have mentioned that this requires the stream identifier (label) and offset to be carried inside the transport header as well as the store-me bits and feedback about resource usage. However, a more fundamental issue regarding accessing the stream buffers is that the source-driven logic of the transport protocols such as TCP does not match the receiver driven pattern of accessing a cache or in this case byte stream buffer. We outline a strawman of such a receiver-driven transport protocol in the next section.

4. A STRAWMAN PROTOCOL: CTP

One of the design goals of stream buffers is that they should be able to operate at high-speed with minimal operational logic with low number of instructions per processed packet. For such efficient operation the currently used extensible text-based protocols, particularly HTTP, is not a good fit. Therefore in the following, we outline a strawman proposal for a receiver-driven transport protocol optimized to interact with stream buffers in a stateless manner: *Content Transport Protocol (CTP)*. In our outline, we focus on those elements of the protocol relevant for interacting with stream buffers, be it at the server or a buffering node, and leave other aspects aside for now.

An alternative to specifying a new protocol could be to extend TCP to support stream buffering [12]. That approach would allow easier deployment but it comes with additional challenges, not least related to the limited extensibility of TCP. CTP has, however, some commonalities with TCP: it transfers an opaque reliable byte streams, and is connection-oriented. Applications that transfer content using TCP should find it convenient to use CTP for transmitting the same content. However, there are some major differences: CTP is receiver-controlled and does not multiplex flows between two end hosts based on port numbers. Rather, the main identification method is the content-based stream identifiers for the payload. Similar to TCP, CTP runs on top of IP, whose forwarding and routing operations work as usual. The differences come with nodes that support shared buffering of CTP packets, and with sharing data between different destinations. The CTP header does not have any dependencies on the underlying layers, allowing replication of entire CTP packet for different destinations. This design is also NAT-friendly: as long as a NAT device does not make assumptions on the transport header structure, changing the IP address does not change the semantics of the CTP message.

CTP also has some commonalities with the approach taken in *information-centric networks* ([10, 8] among many others): The packets are identified based on content-based labels instead of traditional four-tuple of IP addresses and port numbers. However, CTP still runs on top of normal IP network with its traditional routing infrastructure, and operates on sessions between one sender and one receiver.

In CTP mode of operation, the receiver can request a window of bytes identified with the content-based stream identifier and the byte range. This is done using the REQUEST packet. Any stream buffering node on the route between the server and the client can interpret this request and reply to it without having to consult the application layer. Only the server would do the latter to, e.g., fetch the required data from the source (e.g., a file). Data is carried in DATA packets. If intermediate nodes are able to supply part of the data on behalf of the sender, they update the offset and byte range in the REQUEST packet accordingly.

Packets with the same content identifier and byte range are interchangeable, regardless of the application that has triggered them. Due to the stream buffering nature, also overlapping fractions of packets with partially intersecting byte ranges may be stored. This leads to the following pieces of information carried in a CTP header of a DATA packet:

- **Stream Label.** A 20-byte statistically unique content-based stream label. This could be constructed, e.g., from a hash over HTTP URI and other meta-data necessary for identifying a particular version of content, from a BitTorrent piece identifier, or a hash calculated over the content itself. Although the stream buffer operation is agnostic to application-layer data units, applications can use stream labels to differentiate between different application data units to enforce consistent handling of different packets of the same application data unit.

- **Sequence Number.** A 64-bit value indicating the relative offset to the beginning of stream indicated by label. Unlike TCP, the sequence number is not initialized randomly by an end host, because the data needs to be sharable between multiple hosts.

- **Checksum.** A cumulative checksum over the payload. To offer additional (although not very strong) security, cumulative checksum does not only cover the packet itself, but is added on top of the data from the beginning of the stream. Before the first byte of the stream is transmitted, the cumulative checksum is initialized to a random value (shared by all receivers of the same content (stream)), and the checksum of a new packet is added on top of the sum from the earlier data.

REQUEST packets also include the number of bytes requested (which is adjusted by the receiver based upon a congestion control algorithm) and they do not carry a checksum. As noted in the previous section, the header will also include the *store-me* bits and the overload information for the load control feedback loop.

Application Interaction

Because CTP is a receiver-based protocol, it needs a different API than the traditional sockets, even though they start out similarly. A server application creates a listening "socket" by binding to an IP address and CTP service name to support demultiplexing different application classes.[3]

The receiving (client) application initiates the socket state by calling *Get (server address, stream identifier, meta-data buffer)*. After this the receiving application reads bytes from the socket similarly to the normal *read()* call semantics with TCP. A single read call stores a number of bytes to an indicated buffer and may need to be repeated multiple times before a full content object is received.

The write semantics are a little different because parts of the stream may have been served by buffering nodes so that the server may not need to write continuous byte sequences nor start at the

[3]Modeling at least partly hierarchical names could serve a similar purpose and then the application would use wildcards to register its responsibility for a certain set of names.

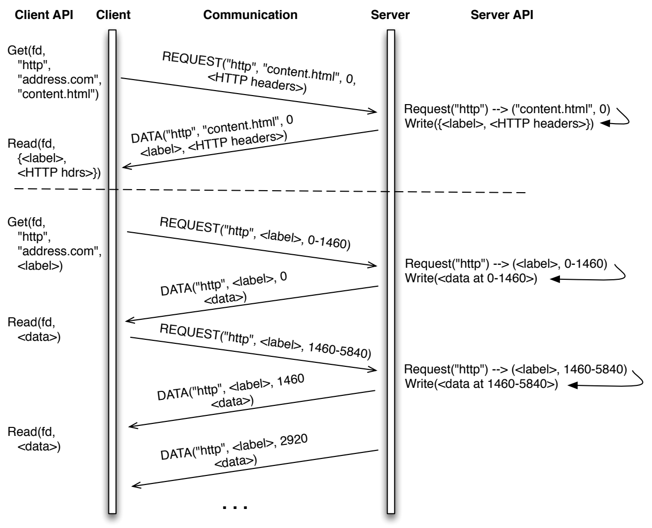
Client API
Client
Communication
Server
Server API
Get(fd, "http", "address.com", "content.html")
REQUEST("http", "content.html", 0, <HTTP headers>)
Request("http") --> ("content.html", 0)
Write({<label>, <HTTP headers>})
DATA("http", "content.html", 0 <label>, <HTTP headers>)
Read(fd, {<label>, <HTTP hdrs>})
Get(fd, "http", "address.com", <label>)
REQUEST("http", <label>, 0-1460)
Request("http") --> (<label>, 0-1460)
Write(<data at 0-1460>)
DATA("http", <label>, 0 <data>)
Read(fd, <data>)
REQUEST("http", <label>, 1460-5840)
Request("http") --> (<label>, 1460-5840)
Write(<data at 1460-5840>)
DATA("http", <label>, 1460 <data>)
Read(fd, <data>)
DATA("http", <label>, 2920 <data>)
. . .

Figure 2: Protocol exchange scenario and application interactions.

beginning of the stream. Hence, the write operation works as a callback: when a REQUEST message is received, the system indicates to the application which stream label and sequence number range is requested. This is shown as "Request" callback in the example in Figure 2. Then, the application writes the data in question to the socket. With system calls such as *sendfile()*, the entire interaction on the server side could be delegated to the kernel.

With many operating system implementations, it may be difficult to introduce new API calls between the kernel and applications. However, using the standard socket API is entirely possible with CTP. In client side, the *connect()* call would replace the Get call, with a new kind of address structure that would indicate the stream label. On the server side, the Request callback could be replaced by *accept()* call, with similar update on the address structure, that is originally returned with the client's address information. With CTP this structure includes the stream label and sequence number range for the data that the server is expected to send.

Example scenario

Figure 2 illustrates a CTP protocol exchange scenario in the case of a HTTP session, and the related API interactions at both client and the server. In the example, communication is divided into two phases: in the beginning the client does not yet know the actual stream label to be used. Therefore it simply uses the HTTP URL (or its hash) in place of the label, to ask the server for the actual stream label. The request can contain HTTP headers from client as its payload, and the DATA response, along with the correct stream label, may also contain HTTP headers from the server. This data is never marked for caching, so it is always sent by the server.

After the initial message exchange, the client opens a new CTP connection, but now requesting the stream label received from the server. This data could potentially be stored at stream buffers, and therefore the DATA packets may have been transmitted by the intermediate network nodes instead of server.

Each REQUEST packet contains the range of bytes that can be sent in response to the packet. If an intermediate stream buffer sends the requested data or portion of it in response to the REQUEST, it modifies the byte range in REQUEST headers accordingly, so that the upstream nodes do not send the same data again.

The CTP receiver acts as a client, opening a connection to an indicated server using the *Get()* call, giving the requested stream label and service name as a parameter. Additionally, the client can specify meta-data to be included as a payload of the REQUEST packet, such as HTTP headers. The connect call triggers transmission of the first REQUEST packet, but if the data flow is longer, the client protocol implementation continues send further REQUESTs as the client reads the data, at the rate managed by the client-side flow and congestion control algorithms.

The sending application acts as a server, that manages multiple short-lived connections for sending the data. There is a master listening socket for each application or service, such as "http" in this example. The server waits for incoming REQUEST packets for specified service, and when one arrives, the *Request()* callback at server application is invoked, resulting in a new short-lived child socket that is used to send the data with the usual write or send calls. After this, the child socket is automatically closed, and the next REQUEST is processed in a new *Request()* call. The main benefit of this approach is that is allows implementing stateless server, that gets all required information for sending data as part of the Request() call. This way, delegating the transmissions to downstream stream buffers becomes also possible.

5. DISCUSSION

The combination of stream buffering, store-me bits with the load control mechanisms, and CTP covers the principles mentioned in section 2. Our application and network independent design allows for alleviating the server and network load under sudden traffic surges and benefiting from in-network storage in a partly end-to-end-controlled manner.

We have done some back of the envelope estimation to check the storage requirements and affectivity of this model for the most expected situations. In our estimations we have used numbers that represent some flash crowd cases [13, 15] as a good representative of sudden spikes of traffic. Based on these numbers, most of the flash crowd events have at least around 1000 requests per minute from more than 100 ASes. The session length usually does not exceed a few 100 seconds. Very often flash crowd traffic caused by one content item triggers further request crowds for 4–5 other correlated content items [13, 15].

Based on these numbers, a shared stream buffer should be able to replicate the first (1–10) minutes of at least 4 or 5 files that might occur in the form of flash crowd at approximately same time. Assuming videos at a rate of 1 Mbit/s yields 7.5–75 MB per video and expected storage capacity of some 250 MB, with a target life time of 100 min. Since sudden spikes of traffic are not that common not much more than this memory space would be needed to handle most of traffic surges effectively, without worrying about details such as different access rates and inter-arrival gap between different requests. Advanced algorithms for managing the stream buffer and its overload signaling are subject of future work.

We also ran initial simulations using ns-3, for a simple topology with 6 traffic sources and 1000 receivers routed through a common shared buffer. Between the sources and the shared buffer there also is a policer that monitors the rate of store-me bits per source against the overload signals, and eliminates excess store-me bits based on this information. The senders have 100 files that have zipf popularity and are requested according to Poisson distribution with mean 50. The file sizes follow Pareto (mean:40 KB, shape: 1.5) distribution. When there is no overload, the store-me bit is set probabilistically for different files, so that the popular files are more likely to be transmitted with the store-me bits. Overloads results to no store-me bits from these sources.

In addition to a basic simulation scenario, we will also run a scenario that has 5 malicious hosts, that send files that are never

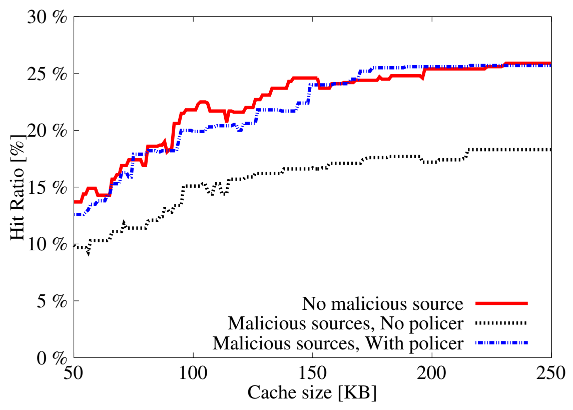

Figure 3: The request hit-rate distribution with 1000 file requests and variable size of shared buffer.

requested but always have the store-me bit set. This may be somewhat artificial scenario, but the purpose of our initial simulation study is to illustrate the effect of malicious hosts on the request hit rate at shared buffers, and how a simple load control mechanism can improve the situation.

For these initial evaluation scenarios we use the reduced hit-rate to identify the overload in the cache. The more the hit-rate goes down the more probable it is for the cache to mark the packets with the overload bits, which are later echoed back to the source. The policer acts based on the information that it keeps about the sources and their number of received overload marked packets. The policer checks and erases the store-me bits for the sources that their number of marked packets is more than 1. Note that here for every source the number of marked packets are reduced every time that the source itself sends new packets with the store-me bits unset.

Figure 3 shows the effect of malicious hosts on the data hit rate on requests arriving at the shared buffer. The graph shows that the presence of malicious host significantly hurts the request hit rate at shared buffer, but how the presence of policer helps to repair the situation. More thorough simulations on the load dynamics with various network parameters are currently in progress and more detailed evaluation is provided in a future paper.

We have left various protocol details, such as MTU discovery, unspecified in this paper. We believe that the request-response cycle of CTP allows ways for implementing MTU discovery and the required feedback, and plan to investigate MTU discovery and other specific protocol details in more detail as future work.

Finally, one issue not discussed in this paper is security. Authentication is one of the main issues in traditional caching systems. Problems such as content cache pollution due to lack of content authentication also exist in our model, and since we operate on arbitrary segments, authentication becomes even trickier. As a first step, our design provides the possibility of having a cumulative checksum over the whole or a window of bytes in a byte stream so that at least name clashes can be detected. Stronger authentication could be added on top. In addition, our strawman protocol has no setup phase in which a server could enforce a handshake with a client to protect against DoS attacks (cf. TCP SYN cookies). Such a phase could be introduced for the first REQUEST packet hitting the server. Our receiver oriented design results to less serious damage to the server in such scenarios compared to any source-driven transport protocol. However, more elaborated work is needed to address this issue.

6. CONCLUSIONS

In this work, we have proposed a content replication model with a focus on reducing the effect of sudden traffic spikes. Our proposal is not meant to replace the existing content replication methods but its goal is to cover a gap that exist between current solutions and the need for alleviating the sudden traffic surges in short periods of time. To reach this goal, we have introduced a network-based stream-buffering model and a content-based transport protocol that could assist in reducing the server workload and, as a secondary effect, redundant network traffic. To make shared buffering effective, servers may indicate to the network which byte streams to buffer while feedback loops and accounting countermeasures are used to ensure that servers don't misuse this capability. An initial assessment suggests that such buffering would be feasible at least in terms of memory requirements to deal with rare flash crowds. Detailed evaluation of our design is subject to future studies.

Acknowledgments

This research was partially funded by the EU FP7 PURSUIT project (FP7-INFSO-ICT-257217). We would also like to thank the anonymous CSWS'12 reviewers for their valuable feedback.

7. REFERENCES

[1] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: an end-system redundancy elimination service for enterprises. In *Proc. NSDI*, 2010.

[2] R. Alimi, A. Rahman, D. Kutscher, and Y. Yang. DECADE Architecture. Internet Draft draft-ietf-decade-arch-07.

[3] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *SIGCOMM '08*, 2008.

[4] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. ReARCH, 2010.

[5] R. Briscoe. *Re-feedback: Freedom with Accountability for Causing Congestion in a Connectionless Internetwork*. PhD thesis, UCL, 2009.

[6] P. B. Danzig, R. S. Hall, and M. F. Schwartz. A case for caching file objects inside internetworks. *SIGCOMM Comput. Commun. Rev.*, 23(4):239–248, 1993.

[7] S. Ihm and V. S. Pai. Towards understanding modern web traffic. *SIGMETRICS Perform. Eval. Rev.*, 39(1):335–336, June 2011.

[8] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. ACM CoNEXT*, 2009.

[9] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. Technical report, CMU, 2012.

[10] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM '07*, 2007.

[11] C. Labovitz, D. McPherson, S. Iekel-Johnson, J. Oberheide, F. Jahanian, and M. Karir. Internet Observatory Report. *Proc. NANOG-47*, 2009.

[12] P. Sarolahti, J. Ott, K. Budigere, and C. Perkins. Poor man's content centric networking (with TCP). Technical Report Publication series SCIENCE + TECHNOLOGY, 5/2011, Aalto University, 2011.

[13] P. Wendell and M. J. Freedman. Going viral: flash crowds in an open CDN. In *Proc. IMC*, 2011.

[14] R. Wetzel. CDN Business Models - Not All Cast from the Same Mold . `http://www.netforecast.com/Articles/CDNArticleSameMold.pdf`, 2003.

[15] H. Yin, X. Liu, F. Qiu, N. Xia, C. Lin, H. Zhang, V. Sekar, and G. Min. Inside the bird's nest: measurements of large-scale live VoD from the 2008 olympics. In *Proc. IMC*, pages 442–455, New York, NY, USA, 2009. ACM.