Flashback: A Peer-to-Peer Web Server for Flash Crowds

Mayur Deshpande, Abhishek Amit, Mason Chang, Nalini Venkatasubramanian, and Sharad Mehrotra
University of California, Irvine
email: {mayur,aamit,nalini,sharad}@ics.uci.edu, {changm}@uci.edu

Abstract

We present Flashback [1], a ready-to-use system for scalably handling large unexpected traffic spikes on web-sites. Unlike previous systems, our approach does not rely on any intermediate nodes to cache content. Instead, the clients (browsers) create a dynamic, self-scaling Peer-to-Peer (P2P) web-server that grows and shrinks according to the load. This approach translates into a challenging problem – a P2P data exchange protocol that can operate in churn rates where more than 90% of peers can leave the overlay in under 10 seconds. This is at least an order of magnitude higher churn rate than previously addressed research. Additionally, our system operates under two strict constraints – users are assured that they upload only as much as they download and second, end-user browsing experience is preserved, i.e., low latency downloads and zero configuration or download of any software. Various innovations were required to meet these challenges. Key among them are (a) A TCP-friendly, UDP protocol (Roulette) for Tit-For-Tat data exchange under extreme churn, (b) A novel data structure (NOIS) for partial-data management and (c) A distributed hole-punching protocol for automatic NAT traversal. Experimental results show the effectiveness and near optimal scaling of Flashback. For a web-server (and clients) running on a DSL-like connection, end-user latency increases only one second for every doubling in web-server load.

1 Introduction

Handling sudden spike or flash loads is an ubiquitous problem for web-servers. High-traffic sites usually over-provision their bandwidth and CPU to handle the spike load. However, even these sites sometimes face unexpectedly high flashes. For example, on 9/11, many leading news sites buckled under the flash load and were forced to scale down the content on their sites. Other web-site hosters use paid third-party service providers (e.g. Akamai [3]) to handle the distribution of 'rich media'. This is in addition to the web-caches (e.g. Squid[2]) and proxies that many ISPs and organizations already maintain. Recently, Peer-to-Peer (P2P) content distribution systems based on volunteer machines have also been proposed and deployed (e.g. Coral Cache [11], Squirrel [12], etc.).

The underlying idea among all these approaches is to replicate or cache the data to a set of intermediate nodes in the network. End user browsers first contact these intermediate cache nodes (or proxies) after checking the local browser cache. If content is already present at these cache nodes and if it is *fresh*, then the content is served directly from the cache, saving the original web-server from the request. However, the end-user browsers still do not share, in any way, the load that they create in the first place. Apart from a philosophical fairness issue, cache-based approaches also suffer certain tangible drawbacks. First, the number of caching nodes (and their current load) dictates the scalability of the system. Second, some web-sites may not favor caching of their data – especially, if hit-count and end-user statistics directly translate to advertising related revenue. If the site sets *no-cache* on its web-pages, then the cache nodes have to get the web-page from the original server for each request from end users.

In this paper we explore the simple idea of distributing the *flash* load *back* to the end user browsers (hence the name Flashback for our system). Such a system is potentially self-scalable – as the load increases the system scales to meet the demand. Secondly, in such a system all end-user requests can be logged by the web-server if need be and third, this system would work well even if the web-page was set not to be cached.

Related Work The lure of a cache-less (free of cache nodes) approach has spurred ideas and techniques on how such a system might be developed ([13, 17, 16, 21]). However, these systems suffer certain drawbacks. First, they assume the users will be co-operative and stay in the P2P for a certain period of time. Second, they do not address the issue of users being behind Network Address Translation (NAT) devices which block incoming connections. Third, they are either not transparent to the user (require setup of proxy or download and configure some software) or require changes to HTTP. To the best of our knowledge, Flashback is the first out-of-the-box deployable and working system that is capable of preserving the user's browsing experience while making no assumptions that the peer will be co-operative.

Unlike a web-cache system, a cache-less system faces a different set of challenges. In a web-cache system (both infrastructure and P2P approach) a set of intermediate nodes maintain full copies of popular web objects. The main problem, here, then is that of finding the particular intermediate node fast enough that holds the needed web object.

In a cache-less system, all end-user nodes that visit a particular web-site are interested in the same web-object. The problem now, is finding the set of other end-user nodes dynamically who might be able to supply (parts of) the web-object. If the nodes are non-cooperative or selfish, however, then there are no nodes that posses the whole web-object – once a node gets the whole object, it can refuse to supply the object to anyone else (a problem not addressed by the original psuedoserving [13] proposal or even CoopNet [16]). The solution to this is to *chunk* the web object into smaller pieces and have the end-nodes exchange the pieces with each other. This kind of chunk-based, tit-for-tat incentive based policy is a popular technique used in large-file P2P content distribution systems (such as BitTorrent [1]).

[1]Due to space limitations this is a highly condensed version of the paper. Full details are available at [6]

[2]http://www.squid-cache.org/

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

The question then is, whether a protocol like BitTorrent can be used in the design of a cache-less system. We argue that while such a system is possible, it would not be very popular. BitTorrent is designed for dissemination of large files and where peers spend many hours in the system. In contrast, web-pages are usually small (ranging from tens of KBs to hundreds of KBs). Secondly, the web-page must be downloaded and displayed in the order of seconds for a normal web experience. Consequently, peers participate in the system in the order of seconds. Within this extremely small time frame, an end-user node must be able to find other nodes and successfully utilize its bandwidth to download the web-object as fast as possible. The crux of the problem in a cache-less system is therefore that of successfully being able to find other peers and exchange data in an extremely high rate of churn. BitTorrent is not designed for this extreme churn and, as we show in our experiments, this results in large end-user latency (to download a web page) that is way beyond the patience of an normal web-user.

Our Contributions: To address this research challenge, we propose Roulette, a UDP based P2P content distribution protocol that is able to operate under extreme churn to distribute even small files (couple of KBs to hundreds of KBs) with low latency and in a Tit-For-Tat manner. Roulette employs a unique overlay construction and maintenance mechanism using a stochastic revolving neighbor cache (hence its name) that is strongly tied to data transfer. Roulette uses UDP to solve another critical design constraint – that of a seamless web-browsing experience. Flashback seamlessly handles NAT by doing hole-punching. Thus, the end-user is not bothered with 'opening up ports' on their NATs. While TCP hole punching has been explored, UDP hole punching is most reliable [10] and thus our decision to base Roulette on UDP. In addition, we also came up with a new distributed hole punching protocol to relieve one central server from participating in each hole punch request (full details are available in [6]).

The decision to base Roulette on UDP had a cascading effect on the design of Flashback. Roulette now had to be explicitly designed to do flow and congestion control (we skipped error recovery) to be friendly to other TCP traffic. Further, this decision catalyzed a design for a more flexible and compact chunk management sub-system. Flashback, therefore manages chunk information in intervals using a novel data structure we call NOIS (Non-Overlapping Interval Skiplist). NOIS allows efficient data exchange in an almost stateless manner and facilitates easy flow control (full details available in [6])..

We tie all the different components into one system that preserves end-user browsing experience (as shown in Fig-1). When a user visits an overloaded site that is running Flashback, she is served a modified web-page and a small applet that contains the code for the Flashback peer and a stripped down web-server. The original web-page is then downloaded by the flashback peer by contacting other flashback peers and served up by the local web-server to the browser. All this works seamlessly through a technique we call the transported frame hack (Full details along with how relative links and absolute links are handled are explained in [6]). Thus, when a user visits a 'Flashback-enabled' site, the experience is no different from visiting any other web-site – there is no download of any special software or configuring of NATs – it just works.

In summary, our main contributions are

- A fully functional and deployed system, Flashback, that can distribute web-pages scalably without intermediate caches

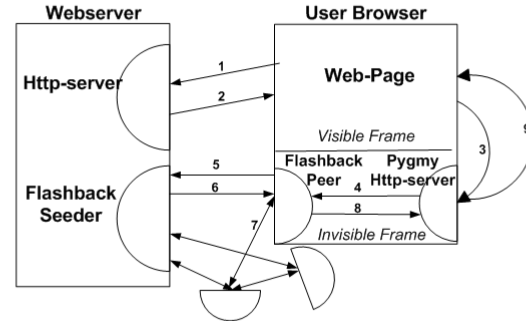


Figure 1. Flashback: High level design showing the flow of data

- Roulette: A UDP based content dissemination P2P protocol that works in extreme churn
- A novel data structure, Non-Overlapping Interval Skiplist (NOIS) for chunk data management
- A distributed hole punching protocol for NAT traversal
- A technique, Transported-frame hack, to display web-pages without user intervention

The rest of the paper is organized as follows. In Sec-2, we describe the problem with extreme churn and the techniques we designed for handling it. In Sec-3, we describe the Roulette protocol in full. We evaluate the performance of Roulette in Sec-4 and conclude in Sec-5. A full comparison to related work and more performance results can be found in [6].

2 Handling Extreme Churn

The primary requirement of Roulette is that it can operate under extreme churn. We term extreme churn as a 50% or more change in the P2P overlay in under 10 seconds. In this highly dynamic setting, normal P2P content distribution approaches either fail or degrade significantly. We use the case study of BitTorrent to explain why. We describe our approach to tackle extreme churn and the two specific requirements that arise out of that approach. How these are tackled in Roulette are described last.

The Problem of Fast Download Under Extreme Churn

The nature of P2P web-page distribution requires that a Flashback-peer be able to download the requested web-page as fast as possible and in an extreme churn environment. Ironically, the faster peers are able to download the web-page, the more churn they create (if it is assumed that they are selfish and leave immediately after the download). The average end-user patience for a web-page to load is around 10 seconds [19] and thus we would expect churn to be in the same time range, i.e., the overlay network can completely change in under 10 seconds. In this time frame, peers must be able to trade and download a web-page. Previous research has addressed P2P data exchange in high churn environments where peers have a life-time of couple of minutes [15] but Roulette faces an order of magnitude different churn rate leading us to term it as **extreme churn**. Further, unlike other P2P system, we assume the worst – i.e. peers can leave as soon as they have all the data. Thus there are no long term peers to take advantage of ([20, 4, 7]). Fast

IEEE COMPUTER SOCIETY

download under extreme churn is therefore the primary design goal and research challenge for Roulette.

Why should extreme churn be a problem? To answer this question we first study BitTorrent, a popular P2P Tit-For-Tat, content dissemination protocol for large (100 of MBs) data that works very well in practice. We examine why simple modifications or tuning to BitTorrent are not sufficient for it to be applicable for small file dissemination under extreme churn. We then present a key insight that is the driving factor behind most of Roulette's design.

A Brief Primer on BitTorrent

In BitTorrent (BT), the content distributor first creates a 'torrent' file (MetaData about the file) which has to be downloaded first by each BT peer. The torrent file contains information how many 'pieces' a file has been chunked into and a SHA hash for each piece. The piece size is decided initially by the content distributor and is usually in the range of 128KB-1MB. When a peer downloads a piece, it verifies the downloaded piece against the hash and finally when the whole file is downloaded, verifies that as well. A seeder peer is also created that has the whole contents. Additionally, there is a 'tracker' that co-ordinates the whole process. Peers contact the tracker to obtain the list of other peers who are currently downloading the file and establish connections to them. When a peer first contacts another peer, they exchange a bit-vector indicating the pieces they already have. This allows each peer to figure out what missing pieces the other peer can provide. After that, a peer updates each of its neighbors with the piece-id of every piece that it successfully downloads and verifies. This allows each peer to maintain a 'stream' of requests for pieces to ask from neighbors. Peers therefore maintain piece 'state' about their neighbors. It is worth nothing that a peer, at a certain time, is only trading with 4-5 of its neighbors even though it pre-opens TCP connections to as many as 20 other peers. Using a technique called 'optimistic unchoking' a peer slowly moves towards trading with those peers that give it the maximum utilization of its bandwidth.

Drawback of BitTorrent under Extreme Churn

BitTorrent is designed to scalably distribute large content (100s MB) and where peers stay in the system for hours. The design choices and default paramater values of BT reflect this. However, a deeper problem with trying to use BitTorrent to trade small files in an extreme churn environment is its philosophy of doing business – **choose a few but 'rich' neighbors** (choosing the 4 peers out of 50 to do data exchange with). A BT peer implements this philosophy using 'optimistic unchoking' to find richer and richer peers (peers with more bandwidth). This however, will be ineffective under extreme churn. First, it may be extremely hard to get an accurate estimate of the bandwidth in the short time frame. Thus it will be hard to discern how rich a peer really is. Second, the extreme churn rate implies that the chosen few neighbors may leave quickly reducing a peers throughput until it finds other peers to ramp up its bandwidth. By the time it finds other trading neighbors, some of the current neighbors may leave. Thus a peer may never be able to utilize its bandwidth fully, resulting in a slow download.

Our Approach to Tackle Extreme Churn

We make the observation that the key to handling extreme churn might infact be to use the opposite philosophy of BitTorrent, i.e., **choose many but 'compatible' neighbors**. When neighbors are disappearing fast, it helps to have a large set of them with whom data can be exchanged. Second, due to the large number of neighbors it will not matter what bandwidth one particular neighbor is providing; the large quantity of them will result in overall effective bandwidth utilization. However, the neighbors should be such that data can be traded with them, i.e., they are compatible.

This solution however, is not efficient in BitTorrent. First, in BT a peer updates all its neighbors on each chunk download. This overhead becomes large when there are a large set of neighbors. Second, since neighbors are arriving so frequently, a handshake of the chunks possessed must be done frequently adding further to the overhead of the protocol. Third, there is anecdotal evidence that TCP congestion control starts to behave erratically when data transfer happens simultaneously over a large number of connections resulting in poor throughput.

In Roulette, we use a two-pronged approach to handle extreme churn. First, we implement a stochastic neighbor recommendation policy that is tied to data transfer. This allows peers to recommend compatible neighbors for other peers. Second, we reduce the overhead of meta-data exchange by eliminating the need for a peer to send updates to its neighbors on each chunk download. We describe these in more detail now.

2.1 Finding Many Compatible Peers

How peers find, keep and delete neighbors has a large impact on the type of the overlay formed and consequently on the data exchange between peers. In Roulette, we have designed a new overlay construction protocol that is explicitly tied to data transfer so that peers can find compatible peers fast. In a sense, we have merged a decentralized heart-beat protocol into the data exchange process and use it for overlay construction. Further, the seeder does not explicitly try to construct or maintain any particular type of overlay resulting in a fully decentralized overlay construction and maintenance.

Keeping up With Lost Neighbors

Due to extreme churn, neighbors disappear quickly and thus it is important to keep a good fill of neighbors. Each peer is initialized with two important parameters, $MinDegree$ and $MaxDegree$ (default of 4 and 32 respectively). When a peer has less than $MinDegree$ neighbors, it continually seeks new neighbors by trying to add a new neighbor every 100ms. Once, it has the minimum required number of neighbors, it still continues to acquire more neighbors (to compensate for leaving neighbors), but the neighbor-seeking rate slows according to its degree. If a peer has more than $MaxDegree$ neighbors, it stops acquiring neighbors. New neighbors are sought by randomly choosing an existing neighbor and asking it for a 'recommendation'.

Referring Neighbors Using the Roulette Cache

The key intuition behind this is that a peer keeps a 'revolving cache' of the most recent neighbors with whom it has exchanged data. When it has to recommend a neighbor to another peer, it chooses stochastically from this revolving-cache (hence the name Roulette). When a peer wants to find new neighbors it asks its current neighbors for recommendations. The recommended peers are one that are most likely to be still active and also possess some data for the file(s) that the requesting-peer is interested in. Since peers are leaving fast, it is essential that a peer find compatible peers that are also active.

IEEE COMPUTER SOCIETY

Whenever a peer sends or receives data (chunks) from its neighbors, it adds them to the *Roulette-Cache (RC)*. The RC is a variable sized cache with the number of slots varying by the peer's current degree ($curDegree$). A neighbor is added to the end of the cache (higher slot number). The cache is then trimmed back, if necessary, to $curDegree$ slots. The neighbors trimmed are at the front of the cache (lower slot numbers). To recommend a neighbor, a peer probabilistically selects a neighbor from the RC. The probability of selecting a neighbor from the cache is $slotNumber/\sum_{k=1}^{s} k$, where s is size of the cache. Thus, the probability of choosing is directly proportional to the slot number, i.e., the neighbors most likely picked from this cache is one with whom the peer has most recently sent or received data from. Secondly, the more number of times a node has traded data from a neighbor, the more likely its recommendation since a neighbor can be present multiple times in the RC. [6] also discusses how overlay partitions are repaired using a simple and elegant mechanism.

2.2 Low Overhead Data Exchange

Keeping overhead low is an important requirement in order to maximize the 'useful' file-data that a peer transfers. In most P2P system, peers exchange meta-data about what parts of the file they have. This allows each peer to know what actual file data to ask or give to another peer. The meta-data is usually the chunk-ids of the chunks a peer has downloaded, encoded in some fashion. In Bit-torrent, two peers initially exchange a bit-vector where the bit number corresponding to the downloaded chunks are set. After this, a peer explicitly updates each neighbor with the chunk-id of each new chunk that it downloads. When the files being traded are small, the chunks have to be also small to allow for parallelism in the system. Web-objects can range from sizes as small as couple of KBs (simple HTML page text) to tens of KBs (CSS, javascript) to hundreds of KBs (images) to tens of megabytes (music and video files). For small file sizes, say tens of KBs, data chunks may need to be as small as 512bytes. Updating a large number of neighbors on each of these small chunk download can easily become a significant fraction of actual data downloaded.

In Roulette, we eliminate these updates to neighbors. Instead peers do a an explicit handshake each time they need meta-data information from their neighbors. In an extreme churn environment, this scheme (no update, explicit handshake) is appropriate because a peer does a lot of handshakes anyways due to the extreme churn rate. Handshakes are costlier than updates and thus must be made efficient. Roulette uses an interval-based approach to tackle this. In a handshake, a peer sends the top intervals of data that it has. An interval-based representation allows for a compact representation of chunk information and thus a lower-overhead data exchange protocol. For example, consider Fig-3 where a peer has downloaded certain portions of a file. If intervals are used, the peer can encode the full information of what chunks it currently has. The number of *non overlapping intervals* of data a peer has downloaded dictates the amount of handshake data it must transmit. The handshake information can potentially reduce as a peer 'fills in the gaps' and also initially, when a peer has a small amount of data, it has very few intervals. The main advantage of non-overlapping interval representation is that the handshake overhead is not constant (unlike a bit-vector representation) with every handshake but is significantly small in the initial and final stages of data download. To aid in meta-data exchange of intervals, we developed an extension to SkipLists [18], called NOIS, that can maintain and search for non-overlapping intervals in $O(Logn(N))$ time. [6] explains NOIS in more detail; NOIS has also been released as open-source software (http://www.ics.uci.edu/ mayur/software/nois.jar).

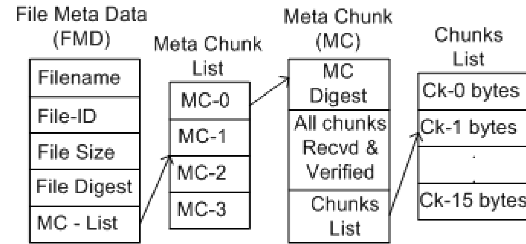


Figure 2. FileMetaData (FMD) Data-structure

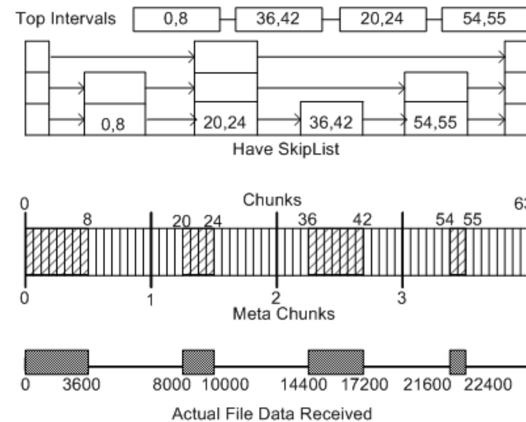


Figure 3. Data Management Layer Cake

3 Roulette Protocol

In this section we describe in detail how Roulette actually does data transfer. Since we use UDP as the transport protocol, much of TCP's functionality has to be emulated. However, we have designed Roulette to maximally utilize UDP without being unfriendly to other TCP traffic. We describe these issues after explaining the data transfer protocol.

After the flashback peer starts in the browser, it joins the P2P network and waits for file request from the local web server (pygmy). The CMS (Content Management System) is initialized for a particular file when the flashback peer receives a request for a file from the pygmy local webserver. The peer then asks the flashback seeder for the *FileMetaData* (FMD for short) associated with this file. The structure of the FMD is shown in Fig2. The seeder replies back with the FMD. The FMD consists of two important pieces of information: (a) The File-ID which is a unique id that is seeder generated. This File-ID is used by peers to refer to the file with each other, since the actual filename may be quite long and impose unnecessary overhead. The FMD also contains an SHA hash of all the file's contents so that when a peer downloads all content of the file, it can verify the integrity of the file.

The FMD is made up of a list of MetaChunks. A MetaChunk represents a set of chunks (by default 16) and contains the SHA hash of the contents of the chunks that it represents. When a peer downloads all chunks of a MetaChunk, it can immediately check if the MetaChunk is valid, If not, it discards all data for the particular MetaChunk. A MetaChunk is an umbrella for faster validation of downloaded content; it has no influence on what or how many

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

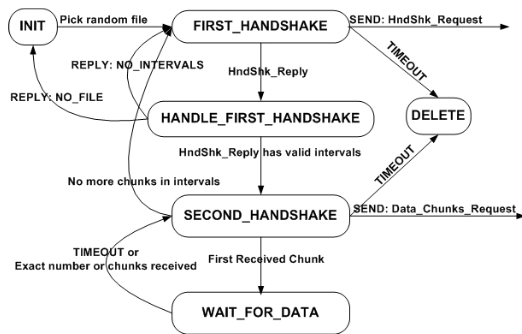


Figure 4. Roulette Protocol: Receive Side

chunks one peer will transmit to another. After the FMD is successfully downloaded, various data structures representing the file are initialized (shown in Fig-3).

3.1 The Data Exchange Process

At a high level, Roulette data exchange can be broken down into two parts, the receiver side and the sender side as shown in Fig 4 and Fig 5 respectively [3]. For each neighbor that a peer has, it continually tries to download data from them. It also has to upload data to them, else the neighbor may cut off download to it (Tit-For-Tat policy). Once the file representation is initialized with the FileMetaData, the peer is ready to start exchanging data for it. We describe the receive side first followed by the sending side.

3.1.1 Receive Side

On the receive side of the protocol, a peer is continually trying to download data for files. The first step is to figure out a file for which the remote neighbor has data. This is the **INIT** phase Fig-4). If the remote neighbor has no data for any file that the peer is interested in, the neighbor connection is terminated.

Once a file is found for which the neighbor is willing to provide data, the peer enters into the **FIRST HANDSHAKE** (FH) mode. Here, the peer sends the top-intervals of data that it has for the file to the remote neighbor. For example, if the peer were trying to download data for the file in Fig-3, and it is allowed to send only 2 intervals, it would send {[0,8], [36,42]}. Since we use UDP, the set of intervals have to fit into one message and thus the restriction on the number of intervals a peer can send. In Roulette, the default maximum number of intervals in a message is 8. The remote neighbor uses these intervals to figure out what intervals of data that it can provide which the peer is missing. If the remote neighbor says it cannot provide any missing data, the file is removed from the consideration set and the per goes back to try and find another file.

If the remote neighbor, however, says that it has data in certain intervals that the peer needs (for e.g., let us assume that it sent back intervals {[12,16], [28,30]}), then the peer transitions into the **SECOND HANDSHAKE** (SH) mode. In the state, the peer sends requests to the neighbor for specific chunks from the intervals of data that the neighbor can provide. The chunks to get are selected at random. To continue the example, let's say the peer asks for chunks {16, 29} (How many chunks a peer asks for is controlled by the *Burst Size* – an important parameter in congestion control).

After sending this second handshake out, the peer waits to get the chunks from the neighbor, i.e., it moves into the **WAIT FOR DATA** (WFD) mode where it waits for data chunks to arrive and then handles them. Handling a chunk involves updating data structures in the content management subsystem and also updating the Roulette Cache. If the peer gets only some (or none) of the chunks, it timesout and goes back to second handshake mode again (how long to wait for timeout is a function or RTT and bandwidth estimate). If it gets all requested chunks, it immediately goes back to second handshake. This time around it has a fewer range of chunks to choose from ({[12,15], [30]})). This cycle from second handshake to wait for data continues until the peer gets all the chunks in intervals that the neighbor originally promised. Once this is exhausted, the peer goes back to first handshake to try and get more intervals from the remote neighbor. The process is continued until the peer gets all the data for the file or the remote neighbor can no longer provide any data. In the latter case, a different file is then chosen or if no such file is available, the remote neighbor connection is severed.

Since the whole protocol runs over UDP, there is no reliable delivery. Thus, with every request message that needs a reply (First Handshake and Second Handshake), a timeout is associated. If the timeout expires, the message is resent. This happens a fixed number of times (default is 4). If there is still no reply, the remote neighbor is assumed dead and removed from the neighbor set.

3.1.2 Send Side

A peer also has to reply to handshake requests from its neighbors and send them data chunks. The reply side is much simpler as compared to the request side.

Upon getting a first handshake request from a neighbor, a peer checks the **Tit-For-Tat (TFT)** policy first. If more data has been transferred to the neighbor than has been received, the peer silently drops the first handshake request. The TFT policy in Roulette is strict. A remote neighbor is only allowed to 'run up a tab' of 4K in data or 90% of received data, whichever is higher. Note that the TFT policy is not restricted to a file but all data transfers for all files for that particular neighbor. Dropping the request when TFT fails, rather than sending a denial message back to the neighbor is an explicit design mechanism. When the request is dropped, the remote neighbor will be forced to send re-requests. This is 'grace period' during which the remote neighbor must make up to this peer. Else, as per the request side protocol, the remote neighbor thinks this peer is dead and removes this peer from its neighbor list. This peer however does not remove the neighbor from its neighbor list. The remote neighbor is present in the neighbor cache until it is garbage collected much later by the overlay management layer. During that time, if the remote neighbor connects back again, this peer will still not respond to any requests. This combination of policies ensures that peers have no incentive to cheat or 'leech' off other peers.

If the remote neighbor passes the TFT test, then this peer must respond appropriately. This involves figuring out if it can supply any missing data intervals for the file. First, the peer checks if it has the file. If not, it responds with a *NO_FILE* reply. The remote neighbors will then try with another file. If the file exists, then with the help of the content management subsystem (CMS), a list of intervals that this peer can provide and which are not present at the remote neighbor are created. This list of intervals are sent

[3] We have designed the protocol using the *State Design Pattern* to implement the transition among the various states in the receive process.

IEEE COMPUTER SOCIETY

```
1)    WHEN message from Y arrives
2)      IF Y has violated Tit-For-Tat
3)        DO NOT send any reply to Y; return;
4)      IF message is asking to supply missing data intervals
5)        SEND intervals we have that Y is missing
6)      IF message is asking for data in chunk-ids
7)        SEND data and update Tit-For-Tat
8)        Add Y to Roulette-Cache
```

Figure 5. Roulette Protocol: Sender Side

back as a **FIRST HANDSHAKE REPLY**. In reply to a second handshake request, again, a check for Tit-For-Tat is first carried out. If this passes, the peer sends the chunks for the chunk-ids that are requested (checking, of course, that this peer has the data for those chunk-ids).

Roulette also performs bandwidth and RTT estimation but due to space constraints, we do not describe it here. A detailed explanation is provided in [6].

3.1.3 Flow and Congestion Control

The application layer also has to provide for flow and congestion control since the transport protocol is UDP. Flow control and congestion control are implemented in a single elegant *stop-and-go* scheme. After the second handshake, a peer waits to receive all chunks. Only after it has received all chunks, does it send out a second handshake again (to get more chunks). This results in automatic flow control. Secondly, the receiver-peer controls the burst of chunks that the sender may send. Initially the burst-size is set to a default of 16. This is increased by 1 for every successful reception of the whole burst. Thus, data transfer between peers ramps up linearly. If a whole burst is not received, the burst size is set to the number of chunks received in the current burst. Thus, Roulette does not follow any mathematical function in reducing the burst but rather bases it on the actual number of data chunks that actually made it all the way from the sender to the receiver. We call this scheme 'linear increase with precise decrease'. This is possible only because of the stop-and-go nature of data transfer. Thus, when there is congestion in the network, the peers automatically scale back to the correct data transfer rate and then greedily try to scale it up slowly.

4 Performance Measurement

In this section, we quantitatively analyze the performance and scalability of Flashback, in particular the Roulette protocol. Preserving end-user browsing experience is the primary goal of Flashback and a key parameter is the latency to download a web-page, even when the web-server is under high load. Thus our experiments are specifically designed keeping this in mind. We perform two major sets of experiments to test whether Roulette can consistently provide low latency for downloads. First, we perform basic scalability tests under 'one-shot' flash loads. Second, we generate consistently high loads on the web-server and test whether Roulette can perform well under high churn. Due to space constraints we do not present the results of the basic scalability test here. The results for that are presented in [6].

4.1 Experiments Framework

To measure the performance of Roulette and be confident that the results would be a good indication of what one could expect in a real deployment, we setup an Internet emulation testbed using Modelnet [2] – a real-time network traffic shaper and provides an ideal base to test various systems without modifying them. Further, Modelnet allows for customized setup of various network topologies. Primarily, we used two main network topologies: (1) a network where all end nodes have bandwidth of 400Kbps and (2) another where all nodes have bandwidth of 800Kbps. For all network topologies, the latency between nodes is always heterogenous, as dictated by the router backbone generated by Inet. On an average, inter-node latency is around 60ms. Detailed notes on the testbed and experiment setup can be found in [6].

4.1.1 Comparison Systems

To the best of our knowledge, Flashback is the first incentive-based system that provides cache-less flash dissemination capability. Thus, the experiments are primarily geared towards testing it. BitTorrent, however, can be a potential replacement protocol for Roulette (since BitTorrent also provides for chunks based dissemination in a Tit-For-Tat manner). Thus we compare with BitTorrent. We did not include other P2P content dissemination protocols (such as Splitstream [5], Bullet [14] or CREW [8]) because they are either designed for streaming content and/or do not perform Tit-For-Tat. We tried to setup Dijjer [9] as a comparison point for a cache-based system but faced many problems. For a baseline comparison, we also test a normal client-server approach using Apache and 'wget'. We describe the specifics of the comparison systems below.

BitTorrent We downloaded and used the python source code for BitTorrent (BT) version 4.0.2. Out of the box, BT is configured for dissemination of large files and for nodes to seed as long as possible. Thus, we made certain changes to it. First, we changed it so that when a BT peer downloads the required file, it immediately exists. Thus, apart from the initial seeder, there are no extra seeders at any time. Next, we changed the piece size. The default is 256KB. With this default, BT performed very poorly for small files. This is easy to understand. When the file is less than 256KB, a peer does not trade with others at all since it waits for the single piece to download and then immediately exits. To compare suitably with Roulette, we changed the default piece size of BitTorrent to be similar to Roulette, i.e., the piece size is now 512Bytes. We also changed the source code so that we could accurately measure the exact time a BT peer took to download a file.

HTTP Client-Server For a baseline comparison, we set up a Apache[4]-2 web-server. Clients to this web-server are emulated using the UNIX command line program wget (version 1.9.1).

4.2 Dynamic Stability Test for Flash Crowds

To test the performance of the various protocols as they would perform under flash traffic we designed a novel experiment called the *Stabilized Pool* test. There are two main goals of the experiment: (a) To test whether a protocol is **stable** under a particular load and (b) To calculate the average end-user delay that a client may experience when the server is under a particular load.

The main goal of this test is whether the system *self-stabilizes*. Peers are introduced at a particular rate to 'hit' the server. If the system is self-stabilizing, then the number of 'incomplete'

[4]http://www.apache.org

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

peers does not keep growing but stabilizes around a certain number. Incomplete peer are those which haven't got the complete file yet. If the throughput of the system is lower than the demand placed on new incoming peers, then the pool of incomplete peers keeps growing. However, if the system is self-stabilizing, then the throughput of the system grows along with the hit-rate and thus the pool of incomplete peers stabilizes. The self-stabilizing characteristic is extremely important for the self-scaling property of a cache-less P2P dissemination system.

In the experiment, we varied the incoming input rate from 1/second to 32/second (3.6K hits/hour to 115K hits/hour). The files that the peers requested were also varied. Every second, we checked the number of currently running peers or the pool-size and stored the value. Every 10 seconds, we evaluated the maximum pool size in last 10 seconds. If this value was higher than the maximum in the previous 10 seconds, the maximum pool size value was updated and the test continued. Else, we assumed that the pool had stabilized and stopped the test. The average total time recorded by *all* the peers that participated in this test was also recorded (in some tests we averaged over 800 peers).

Experiment results The results of the dynamic stability experiment are shown in Fig-6. The Y-axis plot the time in milliseconds and the X-axis (in logscale) shows the rate at which nodes arrive continually at the webserver. For each protocol we tested the time for three different representative file sizes (4KB (text), 64KB(small images) and 128KB (large images)). The performance results for normal HTTP, BitTorrent and Flashback are shown in Fig-6(a), Fig-6(b) and Fig-6(c) respectively. In Fig-6(d) we compare Flashback and BitTorrent side-by-side for one file size, 128KB. We tried running the experiment for 64 nodes/sec but again the CPU on the cluster machines became the bottleneck and skewed the results. We thus show results only for incoming rates upto 32 nodes/sec.

From the Figures, we can make the following observations:

- We did not show all rates for HTTP because it did not stabilize when the input rate was more than 8nodes/sec for 64KB files or larger. We show the latency for input rate for 64KB file as a reference to latency time for 4KB file. The difference in latency time is dramatic and shows why webservers can so easily start "trashing".
- Both Flashback and BitTorrent stabilize under loads upto 32nodes/sec. For BitTorrent, however, there is a sharp increase when the load changes from 4 nodes/sec to 8 nodes/sec but this increase is more smooth for larger incoming rates. There are no such dramatic jumps in Flashback. In general, the end user latency grows logarithmically with increasing load for both BitTorrent and Flashback. Again, this shows the superior scaling of recruiting end users to act as a distributed, self-scaling web-server.
- The difference in end-user latency between BitTorrent and Flashback is quite significant as shown in Fig-6(c). For low load (upto 4 nodes/sec) BitTorrent and Flashback have almost equal latency. However, there is sharp rise in latency time for BitTorrent after that. We conjecture this is due to BitTorrent's inability to handle high churn effectively. The end-user latency for 32 nodes/sec for BitTorrent is over 20 seconds whereas it is less than 12 seconds in Flashback. Clearly, if BitTorrent were to be used as the data exchange protocol it would test many users' patience.

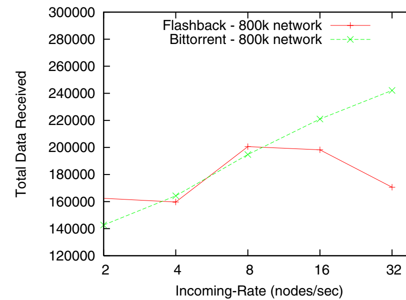


Figure 8. Average data received by each peer in BitTorrent versus Flashback to download a 128KB file

4.2.1 Effect of end-user bandwidths

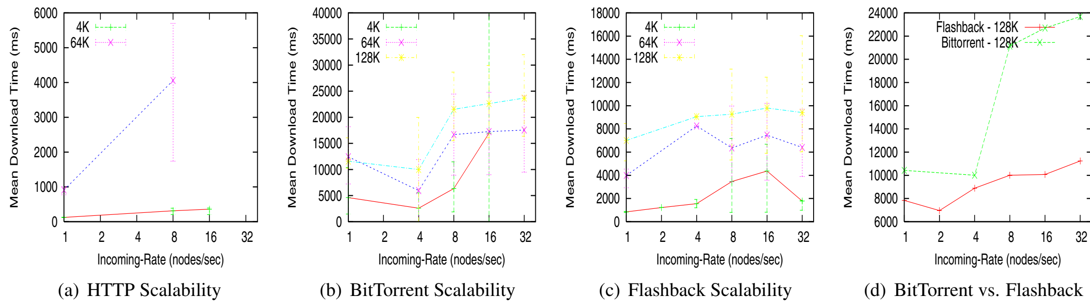
In this experiment we evaluate the effect of end-user latency when the end-user machines have a lower bandwidth capacity. Intuitively, the latency must increase since the system throughput as a whole as reduced. We ignore HTTP's performance since its behavior is easily predictable. We compare BitTorrent and Flashback and show the results in Fig-7(a), Fig-7(b), Fig-7(c).

What is interesting is the difference in the trends in the two protocols for the 400Kbps network. The difference in latency grows bigger with increasing load (number of nodes/sec) and the subsequent increased churn. Flashback scales extremely well in this case. The average end user latency grew only 4 seconds from a load 2 nodes/sec to 32 nodes/sec (an increase in 1 second of latency for every doubling of load)whereas in BitTorrent the latency increased by almost 15 seconds. The absolute latency in BitTorrent at 32nodes/sec is almost too long for a good web experience – at more than half a minute. In comparison, the latency is just above 15 seconds in Flashback.

4.2.2 Data Overhead

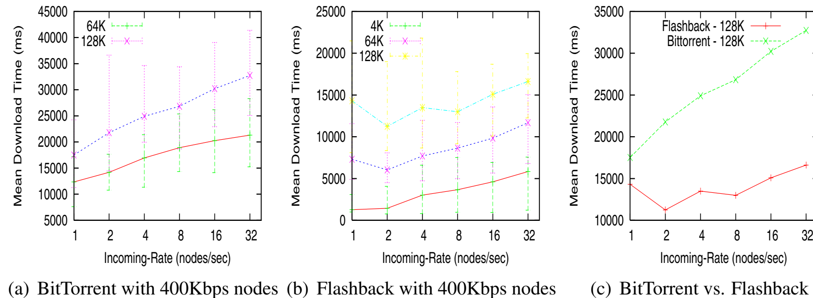
Here, we compare the average total data received by a peer in BitTorrent and Flashback when downloading a file. The amount of data that a peer actually receives during the download process is greater than the actual file because of the overhead of meta-data exchange. In HTTP, the data downloaded is the same as the file size (just a little bigger accounting for TCP and IP header overhead). Fig-8 shows the average data received across increasing load on server to get a 128KB file. The overhead in Flashback is almost constant (and in fact decreases with load) but in BitTorrent it is a steady increase. Note that the varying parameter is the load and not the file size, so changing the chunk size in BitTorrent will not change the trend of this graph. During high churn, the overhead in a BitTorrent like protocol is high due to large number of handshake messages. Flashback has almost constant overhead in spite of increasing load due to the novel interval-based approach to exchanging and maintaining meta-data. This also explains why the end-user latency trends between Flashback and BitTorrent diverge. With the same end-user bandwidth, Flashback is able to provide better 'system throughput'. Why the overhead drops at large load is something we are investigating closely.

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY



(a) HTTP Scalability (b) BitTorrent Scalability (c) Flashback Scalability (d) BitTorrent vs. Flashback

Figure 6. Dynamic stabilization test w.r.t. increasing incoming rates of 800kbps bandwidth nodes. X-axis is logscale. HTTP does not stabilize with more than 8nodes/sec for files over 64KB.



(a) BitTorrent with 400Kbps nodes (b) Flashback with 400Kbps nodes (c) BitTorrent vs. Flashback

Figure 7. The effect of peer bandwidth nodes. HTTP is ignored from comparison. X-axis is logscale.

5 Conclusions

In this paper we introduced a cache-less approach to handle flash crowds at web-sites using a novel P2P data exchange protocol, Roulette that works well in distributing small files in an extreme churn environment. However, we see Flashback not as a replacement for web-caches but as a supplementary mechanism that is useful when Flash crowds appear inspite of web-caches or simply because a web-site does not want its pages cached. Though Flashback has been designed from the ground up to maintain a seamless user experience, some firewalls can still block P2P connections leading to explicit user intervention. Flashback is also currently designed only to distribute static web-pages. We are currently exploring the use of Flashback for more dynamic data use-cases.

References

[1] Bittorrent: http://bitconjurer.org/bittorrent/.
[2] Modelnet: http://issg.cs.duke.edu/modelnet.html.
[3] Akamai. http://www.akamai.com.
[4] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.
[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.
[6] M. Deshpande, A. Amit, M. Chang, N. Venkatasubramanian, and S. Mehrotra. Flashback: A peer-to-peer webserver for flash crowds, http://www.ics.uci.edu/~mayur/flashback_ics_tr_2006.pdf, 2006.
[7] M. Deshpande and N. Venkatasubramanian. The different dimensions of dynamicity. In *P2P*, 2004.
[8] M. Deshpande, B. Xing, I. Lazardis, B. Hore, N. Venkatasubramanian, and S. Mehrotra. Crew: A gossip-based flash-dissemination system. In *ICDCS*, 2006.
[9] Dijjer. http://dijjer.org.
[10] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *USENIX*, 2005.
[11] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *NSDI*, 2004.
[12] S. Iyer, A. Rowstrom, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, 2002.
[13] K. Kong and D. Ghosal. Mitigating server-side congestion in the internet through psuedoserving. *IEEE/ACM Transactions on Networking*, 7(4), 1999.
[14] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Usenix Symposium on Operating Systems Principles (SOSP)*, 2003.
[15] P. Linga, I. Gupta, and K. Birman. Kache: Peer-to-peer web caching using kelips. *ACM Transactions on Information Systems (under submission)*, 2004.
[16] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *IPTPS*, 2001.
[17] J. A. Patel and I. Gupta. Overhaul: Extending http to combact flash crowds. In *WCW*, 2004.
[18] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM. Vol 33.*, 1990.
[19] P. Selvidge. How long is too long to wait for a website to load?
[20] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC*, 2006.
[21] D. Stutzbach, D. Zappala, and R. Rejaie. Swarming: Scalable content delivery for the masses. In *Techincal Report, University of Oregon*, 2004.