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, 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 atleast 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. We believe these are very important for wide acceptance of the system.**

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, (c) A distributed hole-punching protocol for automatic NAT traversal and (d) Automatic rendering of webpages using a technique we call the transported frame hack. 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.

***Index Terms*— Peer-to-Peer, Web-Server, Content Distribution, Churn, Tit-For-Tat, Overlay Maintenance, Scalability**

Relevant Technical Area(s)
Peer-to-Peer, Web-Server, Overlay Protocol Design, Wide-Area Networks, Content Distribution

I. Introduction

Handling sudden spike or flash loads is an ubiquitous problem for web-site hosters. 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 [5]) to handle the distribution of 'rich media'. This is in addition to the web-caches (e.g. Squid[1]) 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 [13], Squirrel [16], etc.).

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

The underlying idea among all these approaches is to replicate or cache the data and shift the load away from the web-server 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 pathological case, the end-user latency actually increases as compared to when there are no intermediate cache nodes.

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.

The lure of a cache-less (free of cache nodes) approach has spurred ideas and techniques on how such a system might be developed ([17], [22], [21], [30]). 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 to setup a proxy or download 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. Secondary problems include that of cache-replication (how to replicate web-objects according to their demand) and cache-eviction (deciding which 'old' web-objects to evict to make room for the new and in-demand web-objects).

In a cache-less system, all end-user nodes that visit a particular web-site are interested in the same web-object. Thus the problems of cache-replication and cache-eviction disappear. Instead, the problem, now, is finding the set of other end-user nodes dynamically who might be able to supply 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 [17] proposal or even CoopNet [21]). 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]).

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.

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 the end-user should not be asked to download or configure any software when using Flashback, i.e., a seamless web-browsing experience. Many end-users are behind NAT (Network Address Translation) devices. These devices block uninitiated incoming connections. Thus, current P2P systems require the end-user to be fairly sophisticated in understanding this technology and require them to 'open up' certain ports on their NAT devices. Techniques to do automatic NAT traversal are therefore popular to remove this burden from the user. A technique called *hole-punching [12]* is particularly effective but it works best with UDP. TCP hole punching is also possible though much more unreliable. To maximize the utility and acceptance of Flashback, we decided the whole protocol would be based 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.

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.

We tie all the different components into one system that preserves end-user browsing experience. 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 and served up by the local web-server to the browser. All this works seamlessly through a technique we call the transported frame hack.

In summary, our main contributions are

- A fully functional and deployed system, Flashback, that can distribute web-pages scalably without intermediate caches
- 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 automatic NAT traversal
- A technique, Transported-frame hack, to seamlessly recruit browsers as web-servers and display web-pages without user intervention

The rest of the paper is organized as follows. In Sec-II we present an overview of Flashback, how it works and meets the requirements for a seamless browsing experience. In Sec-III, we describe the problem with extreme churn and the specific techniques in Roulette designed for handling it. In Sec-IV, we describe the Roulette protocol in full and compare it to BitTorrent and normal client-server system in Sec-V. We explore related work in Sec-VI and conclude in Sec-VII

II. Flashback Overview

Flashback is designed from the ground-up to be easy to use for the end-user while also being easy to deploy on the server side. We first describe the major goals of Flashback and an overview of how the whole system works.

Goals of Flashback: The two main goals of Flashback are : (1) Unchanged user web-browsing experience and (2) A ready-to-use system that can be deployed and used immediately. Taken together, these goals meant that Flashback needed to work across different operating systems and browsers unchanged (or a change to the HTTP protocol, like Overhaul). Secondly, the system should preserve the browsing experience and not require any more expertise than is required for simple browsing. For example, it should not require users to remember and append a URL to access another URL (like in Coral Cache) or install new software (like Dijjer [11]). We now describe how Flashback works and meets these two goals.

A. How Flashback Works

A user who needs content from an overloaded web-server must participate in Flashabck where he/she downloads and uploads data to other users. This is similar to P2P content distribution systems such as BitTorrent [1] (BT) where the peers trade data amongst each other to reconstruct the original file – in effect acting as servers themselves and reducing the bandwidth load of the original server. However, to maintain our constraint that *any* user be able to participate, the user should not have to download, setup or configure a third-party software. Thus, we dynamically recruit the browser itself to be part of the P2P system. We evaluated two alternatives to achieve this: (1) Write extensions to all popular browsers to incorporate the P2P software or (2) Dynamically load the functionality into the browser via Java Applets. We chose (2) for several reasons. First, using Java-applet technology is probably the most Browser/Platform independent way to load functionality into the browser. Second, loading of applets can be made transparent so that the user does not have to install any plugins or extensions and thus deployment of the system in independent of the end-users (of course, assuming that users already have Java plugin installed).

We explain the flow of data in Flashback using Fig-1. When the end-user browser visits a Flashback-enabled website (Step-1), say ‘http://abc.xyz.edu/hot.html’, a ‘two-frame’ web-page is served to the browser (Step-2). One frame is invisible due to zero pixel height and other frame (initially a blank page) takes up all the visible real estate on the browser. The invisible frame is instructed to download and initialize the Flashback applet. The Flashback applet is made up of two main components: (a) Code that runs a Flashback-peer and (b) A tiny web-server (called Pygmy[2])) that starts on the localhost (127.0.0.1:9000) (and thus not accessible from outside). After the applet initializes both the components, it instructs the large visible frame to display ‘http://127.0.0.1:9000/hot.html’(Step-3). Thus the applet automatically rewrites the initial URL to point to the local web server. This call from the applet is handled by the browser which makes a HTTP-request to pygmy (Step-4) for ‘hot.html. Pygmy in turn relays it to the Flashback-Peer (Step-5). The Flashback-peer then contacts the Flashback-seeder (Step-6) to get the meta-data for the file. Once it gets the meta-data it begins trading with other Flashback-peers (it gets ‘initiated’ into the Flashback overlay initially by the seeder) to download the file. Once the file is downloaded, it immediately stops trading and hands over the file (all files are stored in RAM) to pygmy (Step-8) which in turn marshalls it as a HTTP-response back to the browser

[2]http://pygmy-httpd.sourceforge.net/

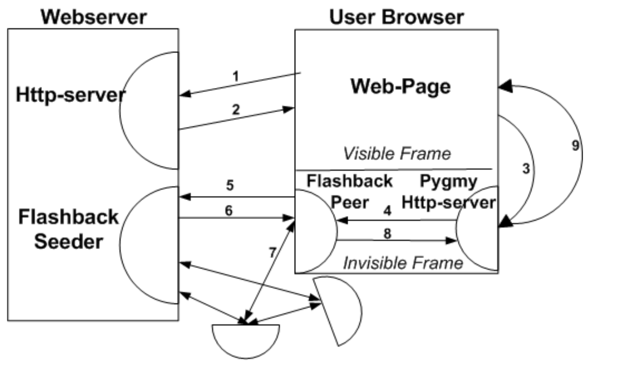


Fig. 1. Flashback: High level design showing the flow of data

(Step-9). The browser then displays the web-page in the larger frame.

If any of the links in ‘hot.html’ are relative links, they are all fetched through the local web-server (and in turn by the Flashback-Peer). This is because the browser thinks that the local host is the source for the web-page. An external link, however, is fetched by the browser as usual. Thus, if a site maintains links to advertising images, these are not served by flashback but directly from the third-party site. The owner of a site, therefore has full control on which web-pages[3] (and even which parts of the web-pages) are to be served via Flashback. The flashback-seeder is started at the root of the web-page hierarchy. When peers contact the seeder for a file, it only had to do a relative lookup for a file on the local machine. Thus, on the web-server side, no changes are required, except starting the flashback seeder and serving the modified split web-page. Further, the split web-page can be served only in case of a large load. Thus, during normal load, the original web-page is served as usual but as soon as high load is detected, the split web-page is served. On the end-user side, the user either gets the normal hot.html file or an applet that initializes itself and automatically downloads and displays the web-page, thus requiring nothing different for the user to do. The only extra step is that the applet requires permission to send and receive data from other peers. If the applet is signed by a certificate, this shows up as a pop-up dialog box. Once the user ‘accepts’ (clicking yes) the applet, the web-page automatically displays. Thus no download or configuring or extra typing is required from the users part. This design therefore satisfies all of the original design goals for Flashback.

Why serve two frames instead of one simple blank web-page? This is because of the subtlety in how applets are handled in a browser. If there is only one frame and the applet asks the browser to load the web-page there, the browser redirects to the local pygmy server, the applet is stopped because the browser has moved on to a new page – in effect stopping both the Flashback-peer and the pygmy server, and thus nothing will be displayed to the user. The two-frame design is what we term as the ‘Transported Frame Hack”

[3]We use the term web-page to mean a set of file-objects that are needed to display the web-page correctly, for example the embedded images, CSS files, javascript files, etc.

(Similar in spirit to the Transported-Man magic trick where the magician uses his twin brother to fake the illusion one person doing impossible tricks.)

B. Automatic NAT Traversal with Distributed Hole Punching

In designing Flashback, we wanted the use of the system to be seamless and also be accessible to a wide population. Thus, one of the high level requirements for Flashback is that the system should be able to run even without end-user intervention. Thus, peers must ideally be able to form connections with each other automatically, even if some them are behind NAT devices (NATs for short). NATs, in brief, multiplex many end-user nodes into one public IP(v4) address. This is useful for home users that have many computers but only one IP address or ISPs that are short on IP addresses. The main problem with NATs is that, by default, they do not allow new incoming connections. For a node behind a NAT to accept new incoming connections, the NAT must be explicitly configured to allow incoming packets on certain ports (opening ports on a a NAT). This issue is one of the major stumbling blocks for a P2P protocol. Most P2P systems require the end-user to be able to access their NAT-devices and change the settings to open ports.

Automatic NAT traversal is possible using a technique called “Hole Punching” [12]. Hole punching requires a well-known server that is not behind a NAT. Hole punching has certain subtleties, especially in the case of TCP and these are discussed in detail in [12]. UDP hole punching is relatively more straightforward and robust. We experimented with TCP hole punching as well but quickly realized that it was unreliable – especially when the NAT devices maintained TCP connection state. As [12] reported, UDP hole punching worked across 82% of NAT devices compared to 64% in TCP. Given these factors, we decided to design Roulette to work over UDP. A concern with hole-punching though, is the impact it would have on the seeder. If each Flashback-peer involved the seeder each time it wanted to do a hole punch, the overhead on the seeder may be too high. We tackle this using distributed hole punching.

For hole punching to work correctly, a “middle” server must know both endpoints of the two nodes that want to establish a full duplex connection to each other. When a Flashback-peer first contacts the seeder, it sends the IP and port that it sees. If the peer is behind a NAT, this is the IP and port that the NAT has assigned. The seeder replies back with the IP and port that it is seeing. This will be the IP and port of the NAT device. Taken together, these constitute the “identity” of the peer. Both the seeder and the peer now have this identity information. Once a peer discovers its own identity, it begins to search for other neighbors. It starts be asking the seeder to refer it to another peer. The seeder then acts as the intermediary in the hole punching process. It sends the new peer’s identity to one its random neighbors. It also sends the neighbor’s identity to the new peer. The two peers then attempt hole punching. Note that each peer has the other’s both endpoints. If the hole punch succeeds, packets start flowing in both directions and a *full duplex* connection now exists between the two peers. This process is repeated and the peer now obtains two neighbors. Now, this peer knows both endpoints of its two neighbors and can act as the intermediary in a hole punch process between them. The server is not needed in this hole punch. In effect, peers start acting as intermediaries for their neighbors’ hole punch process, i.e., a distributed hole punching process. The beauty of this is that the seeder is now in no way special as far as getting more neighbors are concerned and just another neighbor peer.

III. Roulette: 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.

1) 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 [26] and thus we would expect churn 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 [20] 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 ([29], [7], [9]). Fast download under extreme churn is therefore the primary design goal and research challenge for Roulette.

Assuming users will be selfish and leave immediately is not unrealistic and in fact might be the right thing to do. In P2P file sharing systems (such as Gnutella [2]), most users tend to act selfishly. In the context of web-content, a minute or two of altruism would probably not hurt the user. However, we feel that if the user is given strong guarantees that he/she will only upload as much as they download, they would be more accepting of such a system. Further, for certain ISPs where users are charged according to bytes transferred, this requirement becomes especially critical.

Why should extreme churn be a problem? To answer this question we first study BitTorrent, a popular P2P Tit-For-Tat, content dissemination protocol that works very well in practice. BitTorrent is primarily designed for dissemination of large content and where peers stay in the system typically for hours. We then 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 also a 'tracker' that co-ordinates the whole download 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 them. 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. Peers also regularly inform the tracker of their progress and the tracker also continually checks if a peer is still in the system or has left.

Drawback of BitTorrent under Extreme Churn: BitTorrent is designed to scalably distribute large content (hundreds of MBs) 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.

A. 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 also the name Roulette). A separate cache is kept for each file is that is traded. 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.

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.

Detecting and Repairing Overlay Partitions: Occasionally, a group of peers can become partitioned from the main overlay due to high churn. Then, after some time, everyone has all the chunks in the group and there are no more 'compatible' peers. Thus, peers must watch out for partitions actively and find compatible peers again. We use a simple and completely local technique to guard against partitions. A peer continually tracks how much new file-data it receives every 500 ms. If it does not receive any new data, it suspects it may have become partitioned. It then asks the seeder for a new neighbor, since the neighbors of the seeder are, by definition, not partitioned. Once the peer is connected to a neighbor of the seeder, it is back in the main overlay.

B. Low Overhead Data Exchange in Extreme Churn

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) may be quite appropriate because a peer may be doing 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, initially, when a peer has a small amount of data, it also 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 reduced in the initial and final stages of data download[4]. To aid in meta-data exchange of intervals, we developed an extension to SkipLists [23] that can maintain and search for non-overlapping intervals in $O(Logn(N))$ time. We describe this lightweight and easy to implement data structure next.

NOIS: NOIS stands for Non-Overlapping Interval Skiplist. We use NOIS to store the intervals of chunks (not bytes) that have been downloaded for a file. Since the Roulette data exchange protocol uses intervals extensively, it is important to have a data structure that supports addition, deletion and search for intervals efficiently. A skiplist [23] is a probabilistic data-structure that approximates the functionality of a balanced tree (providing $O(Log(N))$ search, insert and delete) w.h.p.

Skiplists are most used to store single point values. However, for Roulette, we need to store and manipulate intervals. Generalized intervals-skiplits [14] have been proposed for stabbing queries that function similar to an augmented interval red-black tree but fully-general interval skiplists are an overkill for our application. We know that the intervals we need to maintain are non-overlapping (if duplicate data chunks are received, they are discarded) and hence a simpler data structure can be built. In NOIS, each skip-node is similar to a node in a normal skiplist, except that, a range is maintained in each node (Fig-3. Further, the operations (search, insert, delete) in NOIS are all range based. The search operation is similar to normal skiplist search (with checks of overlapping or containment). The case of interval deletion and insertion are more special since an insertion may lead to merging of intervals and deleting a small portion of a large range leads to creation of more intervals. NOIS has been released as open-source software[5].

[4]Various other alternatives to design an efficient handshake method are possible. For example, sending a fixed-length bit-vector using Bloom Filters. We are currently exploring this and other alternatives

[5]A Java implementation is available at http://www.ics.uci.edu/ mayur/software/nois.jar

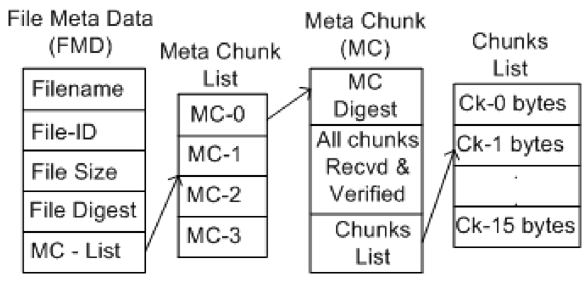


Fig. 2. FileMetaData (FMD) Data-structure

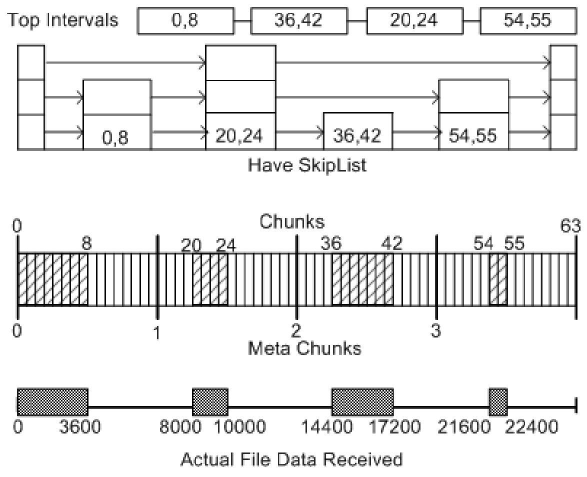


Fig. 3. Data Management Layer Cake

IV. 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. Finally, we explain the role and functionalities provided by the Content Management Subsystem (CMS).

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 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 chunks one peer will transmit to another. After the FMD is successfully by the peer, various data structures which together constitute the representation for the file are initialized (shown in Fig-3.

A. 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. An algorithmic description of both sides are shown in Fig 4 and Fig 5 respectively[6]. 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.

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 (lines 4-7 in 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 (line 9 in Fig-4). 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 (lines 10-12 in Fig-4).

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 (line 14, 17 in Fig-4). 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 (lines 19-23 in Fig-4) where

[6]While the figure shows the use of Goto statements, we have not actually implemented Roulette that way. We use the *State Design Pattern* to implement the transition among the various states in the receive process.

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 (line 16 in Fig-4). 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.

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 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 rerequests. 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. The peer however does not remove the neighbor from its neighbor list. The peer is present in the neighbor cache until it is garbage collected much later by the overlay management layer. During that time, if the neighbor connects back again, the 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 the 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 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

```
1)  WHILE all data for all pending files are not received
2)    FOR EACH neighbor $X$ DO
3)  INIT:
4)        IF $X.File$ is $NULL$
5)            Initialize $X.File$ to a mutually interesting file
6)            IF no such file exists
7)                Remove $X$ from Neighbor-List;
8)  FIRST HANDSHAKE:
9)        Initialize $\{X.File.Intervals\}$ to missing data intervals
10)       IF $X\{.File.Intervals\}$ is $\emptyset$
11)           Remove $FILE$ from mutually interesting set
12)           Set $X.File$ to NULL; GOTO: INIT
13) SECOND HANDSHAKE:
14)       Select $\{Chunk\_IDs\}$ from $X.FileIntervals$
15)       IF $\{Chunk\_IDs\}$ is $\emptyset$
16)           GOTO: FIRST HANDSHAKE
17)       Ask $X$ for $\{Chunk\_IDs\}$ data chunks
18) WAIT FOR DATA:
19)       As data chunks are received
20)           Update data intervals for $File$
21)           Add $X$ to Roulette-Cache
22)           IF All chunks have been received or TIMEOUT
23)               GOTO: SECOND HANDSHAKE
```

Fig. 4. Roulette Data Exchange Protocol:Receiver Side

```
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
```

Fig. 5. Roulette Data Exchange Protocol: Sender Side

intervals are sent 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 the peer has the data for those chunk-ids and the burst of the data that the peer has asked for is not violating the TFT policy).

3) **Remote Neighbor RTT and BW Estimation:** Estimating a packet’s Round Trip Time (RTT) and the bandwidth to a remote neighbor is important to take timely decisions. For example, if the RTT is known, then a node can decide whether it should retransmit a request when no reply is received within a particular amount of time. Since we use UDP as the transport protocol, packet loss detection and reliable transmission must be handled explicitly at the application layer. Most operations at the application layer are of ‘Request-Reply’ nature, for example, random neighbor, first-handshake and second handshake operations. When a remote neighbor is sent one of these messages, it is expected to reply back with an appropriate reply message. If it does not reply within a certain amount of time, we resend the request. The default RTT estimate between two peers is initially set to 100 msec. Every request message is tagged with a sequence number. The reply to the request message contains the sequence number enabling a node to estimate the RTT for the particular request (which is “smoothed” with each successful RTT estimate). For data chunks however, this estimate is revised to take into account the bandwidth intensive nature of the data chunks. Initially, the bandwidth between two peers is set to zero. The bandwidth value is only updated when multiple data chunks are received in sequence. The inter-arrival time between the data chunks is used to calculate the bandwidth as $ChunkSize/InterArrivalTime$. Bandwidth estimate is also updated smoothly. Using this bandwidth estimate, a node expecting multiple chunks calculates the timeout for the arrival of the next chunk as $2*CurrentRTT+Bandiwdth/ChunkSize$. The factor of 2 for $CurrentRTT$ is to make the timeout conservative to account for sporadic congestion in the network and delays within Roulette's read and write queues.

4) **Flow and Congestion Control:** The application layer also has to provide for flow and congestion control. Flow control and congestion control are implemented in a single elegant *stop-and-go* scheme in Roulette. 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. 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 the data transfer rate and then greedily try to scale it back up slowly. This scheme of linear increase and stop-and-go means that the maximum available bandwidth between two nodes is not exploited fully at first. However, the large number of neighbors a peer has compensates for this.

B. Content Management Subsystem

The content management subsystem in Roulette is highly sophisticated with multiple representations for file-data (as shown in Fig-3. The core of this system is the *Have Intervals* that are stored in a Non-Overlapping Interval Skiplist (NOIS). Apart from the Have Skiplist (HSL), the CMS also maintains three other data structures. A sorted list (by decreasing size) of the chunks intervals is maintained. This list is updated whenever the HSL is changed. Two bit-vectors, one each for the meta-chunks and the chunks are maintained. A set bit in the chunks bit vector indicates that the chunks has been downloaded. A set bit in the meta-chunks bit vector indicates that the meta-chunk has been verified.

Functionalities of the CMS: Here we consider the main functionalities that the content management subsystem is required to provide and the asymptotic cost of each operation. For example, consider a file (of size 25,600 bytes) that is currently being downloaded by a peer as illustrated in Fig-3. This file has four MetaChunks and 64 chunks of 400 bytes each. The data that has been downloaded so far is shown in shaded-grey. When a remote neighbor sends a first handshake message to ask for intervals of data a local peer can provide, the local peer first makes a 'clone' of the local *Have-Skip-List* (HSL). The remote intervals are then deleted from this cloned HSL. The intervals that are left over are the intervals of the data that the local peer has but which the remote peer does not have. The top intervals in this cloned and deleted skiplist are then sent to the remote neighbor. Each deletion is a $O(Log(N)$ operation, where N is the total number of intervals in the skip-list. The remote peer is only allowed to send a certain fixed number of intervals, so this whole operation is O(Log(N)). The cloning operation can also be implemented efficiently with a memory copy.

When a remote neighbor sends a second handshake message to ask data for particular chunk-ids, the local peer makes a direct lookup into the chunk bit-vector. If the bit is set (chunk downloaded), it sends the data to the remote peer. This is a $O(1)$ operation.

When a new data chunk is received from a remote neighbor in the receive side of the data exchange protocol, various changes to the data structures are required. The chunk bit-vector is first checked for duplicate chunk. If it is duplicate, the chunk is discarded and no further changes are made. Else, the chunk bit-vector corresponding to the Chunk-ID of the chunk is set to 1. A check is then made to see if this chunk completes any meta-chunk. If so, the data corresponding to all chunks in the meta-chunk is checked for data integrity with the SHA-hash that was originally provided by the seeder. If the check passes, the bit in the meta-chunk is set to 1. Else, the bits of all chunks belonging to meta-chunk are reset to 0 and the data for the chunks is discarded and the HSL is updated. If the chunk does not complete a meta-chunk, the data for the chunk is conditionally added to the Have-Skip-List. All bit-vector operations finish in $O(1)$ time as does a hash-check. In case of hash-check fail one interval (corresponding to 16 ids) has to be deleted and in case of adding a data chunk, one interval has to be added to HSL. Both are $O(Log(N))$ operations. In case the HSL is updated, the sorted interval data structure also has to be updated. This requires adding one new interval, removing two intervals (due to removal of bad chunks) or updating one interval. With a binary search on the list, this can also be done in $O(N)$ time, since the list is sorted to begin with (and stays sorted even after the operation).

V. 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.

A. 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 [4] – 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[7]. Next, we describe our experimental testbed and the network topologies that we used.

1) Testbed: The testbed consists of a FreeBSD machine as an emulator and 13 clustered Debian Linux hosts. The emulator supports a Gigabit ethernet interfaces while the Linux hosts have 100Mbps ethernet interfaces. All machines are connected by a dedicated Gigabit router. The emulator is a dual processor 2.6Ghz machine with 2GB of RAM while the hosts are dual processor machines running at 900Mhz with 500MB of RAM. The emulator machine runs a custom FreeBSD Kernel configured with a system clock running at 1000Hz (as required by Modelnet). The hosts run Linux with a customized 2.6 version kernel [8]. The hosts support Java version 1.5 and Python version 2.3.5. All hosts are synchronized to within two milliseconds through NTP (Network Time Protocol).

To model the vagaries of the underlying Internet, we used the *Inet* [3] topology generator tool to generate Internet router topologies of 5000 routers. Inet generates topologies on a XY plane which Modelnet then uses to emulate inter-router (and hence inter-node) latencies. On this router backbone, 50 subnets are created and each subnet is allocated 5 hosts for a total of 250 emulated hosts. Bandwidth constraints and network packet loss rates are specified separately in Modelnet. 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. We randomly picked 1000 node-to-node endpoints (out of 250*250 possible node-to-node combinations) and plotted the latencies between them. This approximates the latencies peers will experience when communicating with each other. The latency-graph is shown in Fig-6.

Our choice of bandwidths for nodes requires some explanation since the testbed imposes certain restrictions. First, the maximum bandwidth generated in the testbed cannot exceed 1 Gbps (bottleneck of emulator NIC card and router). Second, to keep the emulator from being overloaded, we did not want to generate data at such a rate that the emulator CPU usage went above 10%. Third, while Modelnet provides for running many virtual nodes in one physical host, we did not want to create so many processes that the swap space was being used. Under these constraints, we would still like to simulate reasonable bandwidth assumptions. In our use-case we assume many end-users will have broadband. 400Kbps and 800Kbps are close enough approximations of DSL connections available today. Users may also be behind higher bandwidth connections (such as cable-modem that can provide data rates in the Mbps range) but the primary objective is to study the scalability patterns and not exhaustively test all possible bandwidth scenarios. We feel our choice of bandwidths for end-users is a reasonable approximation that can provide good insights into the real-world behavior of the tested protocols.

[7]Another testbed choice we considered was PlanetLab (http://www.planet-lab.org/) but most nodes there have high bandwidth. Secondly, designing custom overlays with different end-user bandwidths would have been very difficult.

[8]This version supports NPTL (New Posix Threading Library), to efficiently support multiple threads.

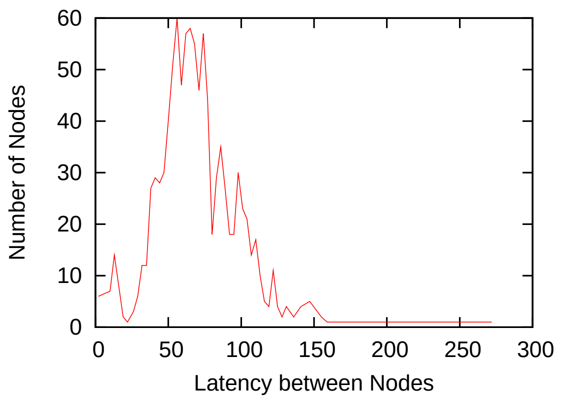


Fig. 6. Distribution of latency among nodes in the Modelnet Internet emulator testbed

2) 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 [8], Bullet [18] or CREW [10]) because they are either designed for streaming content and/or do not perform Tit-For-Tat. We tried to setup Dijjer [11] 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. We did not make any other changes, for example changing it to enforce stricter it-For-Tat. A BT peer can therefore keep (and serve) upto 50 pending requests if it needs to (as is default in BT) and thus Tit-For-Tat can be lax.

Dijjer: Dijjer is a freely available P2P web-cache system developed using core concepts from Freenet (reference). Dijjer nodes form an Distributed Hash Table (reference) and web-files are also hashed so that a node can find a file from the 'closest' node using DHT routing. During the searching process, the file is replicated along the search path. Similar to other P2P web-cache system, Dijjer nodes are long lived (compared to Flashback) who co-operate with each other to cache popular content and serve it to end nodes. We tried to set up Dijjer so that nodes would behave more selfishly, i.e. join to receive a file and then leave as soon as they got the file. Trying to set up Dijjer this way turned out to be quite difficult. When there is large churn, new incoming nodes turn out to be 'closer' to the file that nodes in the system; thus they get the file straight from the web-server rather than from other peers. Thus, a Dijjer system set up so that when nodes leave immediately performs as bad (or good) as a normal client-server system. Due to this, we do not use it in our comparison test.

HTTP Client-Server: For a baseline comparison, we set up a Apache[9]-2 web-server. Clients to this web-server are emulated using the UNIX command line program wget (version 1.9.1).

B. Basic scalability

The goal of the basic scalability experiment is to test how the systems perform under 'one-shot' loads for various file sizes and with different client bandwidths. For each system, a server (or seeder) is started first. A certain number of clients (BitTorrent peer, Flashback peer or wget process) are then created concurrently and are asked to download a particular file. When a client/peer downloads the file, it immediately quits. The number of simultaneous clients are varied from 4 to 96. The upper limit is because that creates close to 8 peers on each of the linux cluster machines and CPU utilization when running Flashback nears close to 80%. Beyond this, the test results for Flashback start to get skewed and hence the limit. Peers are asked to download files varying in size from 4KB to 128KB. This again reflects types of files that might constitute a web-page, for example, simple HTML files are usually a couple of KBs, CSS and javascript files are around 20-30KBs and images are usually between 40-150KB. All clients and the server have a maximum (symmetric) bandwidth. We test with two network topologies – 400Kbps and 800Kbps. As noted before, these are approximations to DSL like connections. The requirement that peers exit immediately is to mimic total selfish behavior. Even in single-shot loads, this creates some churn since all peers do not finish at once. The peers that finish later have to deal with exiting peers.

Experiment results: The results of the basic scalability experiment are shown in Fig-7. The Y-axis plot the time in milliseconds and the X-axis (in logscale) shows the number of nodes that participated in the one-shot load. We run each test 5 times and plot both average and 90% values. For each protocol we tested the time for different file sizes (from 4KB upto 128KB). The performance results for normal HTTP, BitTorrent and Flashback are shown in Fig-7(a), Fig-7(b) and Fig-7(c) respectively. In Fig-7(d) we compare the three systems side-by-side for one file size, 128KB. The spike in latency seen at 96 nodes for some file sizes for BitTorrent and Flashback is due to the CPU on the cluster machines becoming the bottleneck running the large number of processes. Thus, we did not experiment with a larger number of nodes than 96.

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

From the Figures, we can make the following observations:

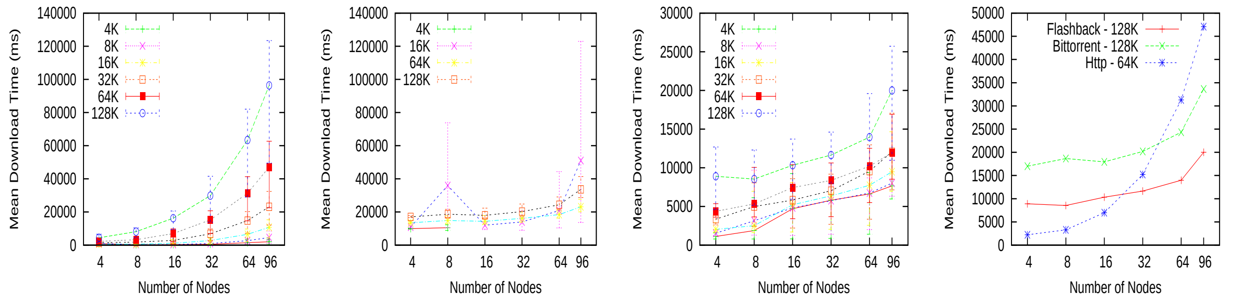
- In normal webserver (HTTP-Apache) the end user latency grows linearly as a function of both increasing load and increasing file size. In comparison, both BitTorrent and Flashback scale logarithmically with increasing load. A P2P approach, therefore, offers superior scalability and end-user latency benefits.
- Flashback handles load better than BitTorrent, both in lower end-user latency as well as the variability. For e.g., in BT, the 90^{th}ile value can be almost three times the average value while in Flashback it varies less than 30%. Note that the Y-Axis of BitTorrent and Flashback are different.
- When the protocols are compared side-by-side, the difference in the protocols become clear. For example, a BitTorrent peer takes 25 seconds on average to download a 128KB file when 64 peers are started at once. A Flashback peer takes less than 15 seconds. This is because, in BitTorrent, many peers leave at the same time and the remaining peers take time to reconnect and ramp up their bandwidth. In contrast, Flashback peers are continually trading with many peers and thus are less susceptible to exiting peers.

Even with one-shot loads, the difference between the three systems is easily perceptible. We know study the effects of even increased churn and its effects in the next section.

C. 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' 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.



(a) HTTP Scalability (b) BitTorrent Scalability (c) Flashback Scalability (d) Three Systems Comparison

Fig. 7. Basic scalability test w.r.t. network size and file size with 800kbps bandwidth nodes. X-axis is logscale.

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 assume the pool has stabilized and stop the test. The average total time recorded by *all* the peers that participated in this test is also recorded (in some tests we averaged over 800 peers).

Experiment results: The results of the dynamic stability experiment are shown in Fig-8. 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-8(a), Fig-8(b) and Fig-8(c) respectively. In Fig-8(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.

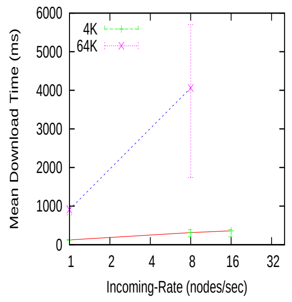
From the Figures, we can make the following observations:

- 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. We are currently studying why this happens. 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-8(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.

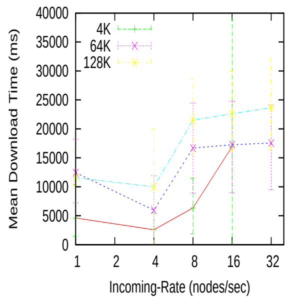
1) Effect of end-user bandwidths: In this experiment we evaluate the effect of end-user latency when the end-user machines have a lower bandwidth capacity. Intuitively, the latency must increase since the system throughput as a whole as reduced. We ignore HTTP’s performance since its behavior is easily predictable. We compare BitTorrent and Flashback and show the results in Fig-9(a), Fig-9(b), Fig-9(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.

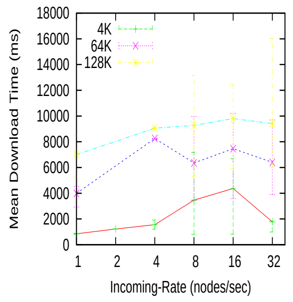
2) Data Overhead: Here, we compare the average total data received by a peer in BitTorrent and Flashback when downloading a file. The amount of data that a peer actually receives during the download process is greater than the actual file because of the overhead of meta-data exchange. In HTTP, the data downloaded is the same as the file size (just a little bigger accounting for TCP and IP header overhead). Fig-10 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



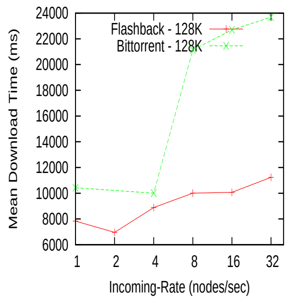
(a) HTTP Scalability



(b) BitTorrent Scalability

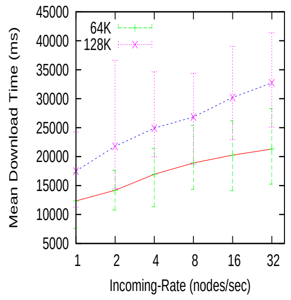


(c) Flashback Scalability

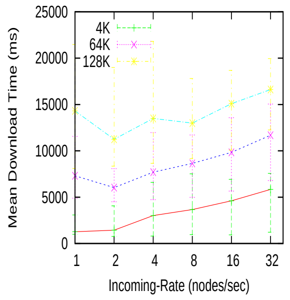


(d) BitTorrent vs. Flashback

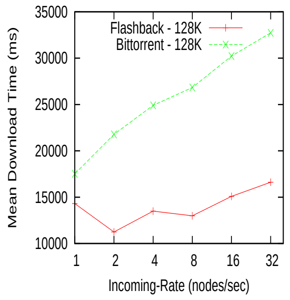
Fig. 8. Dynamic stabilization test w.r.t. increasing incoming rates of 800kbps bandwidth nodes. X-axis is logscale. HTTP does not stabilize with more than 8nodes/sec for files over 64KB.



(a) BitTorrent with 400Kbps nodes

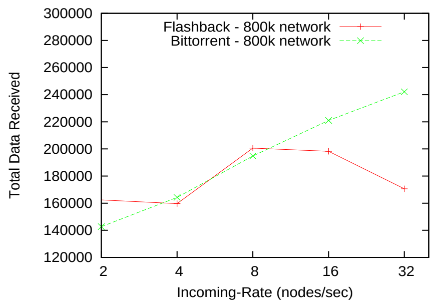


(b) Flashback with 400Kbps nodes



(c) BitTorrent vs. Flashback

Fig. 9. The effect of peer bandwidth nodes. HTTP is ignored from comparison. X-axis is logscale.



(a) Data Overhead

Fig. 10. Average data received by each peer in BitTorrent versus Flashback to download a 128KB file

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.

Summary: P2P approaches are highly scalable and are ideal for building self-scaling web-servers. However, the protocol must be explicitly optimized for small files and extreme churn. While BitTorrent shows good scaling trends the absolute latency is far higher than what an end user might like. Flashback, in comparison is both scalable with low end-user latency.

VI. Related Work and Future Directions

Flashback is a cache-less approach to handle flash crowds. This approach was proposed as psuedoserving in [17] where clients have to agree to share content with other clients. However, there is no mechanism (such as Tit-For-Tat) to actually enforce this and no working system with this technique. In CoopNet [21], the authors propose and implement a cache-less approach. However, clients get whole files from others and clients are also assumed to be co-operative and stay in the system for a few minutes. Flashback works scalably even without these assumptions. The churn rate we handle is in the order of seconds, a magnitude higher than what is assumed in CoopNet. Further, since clients usually get whole files in CoopNet, the authors present techniques on how to find the closest or best peer using IP address prefix matching. We do not employ this in Flashback because the primary objective in Flashback is not to find a few best peers but to get a lot of peers and download parts of file from them. In Overhaul [22], the authors also explore a cache-less approach and wherein the file is split into chunks and clients get parts of the file from each other. However, their approach requires a change to the HTTP protocol in order to achieve client-side transparency. Even if the proposed changes are accepted, it would take years before browsers and servers reflected this change. Additionally, the problem of clients behind NATs would still need to be addressed, probably requiring HTTP to run over UDP (like DHTTP [24]. Overhaul clients are also assumed to be co-operative and secondly, it is not clear how well Overhaul would work in an extreme churn environment. However, the experimental results presented strongly validate the the scalability benefits of a cache-less approach. [30] is a simulation based study that explored the benefits and usefulness of a cache-less approach and showed the potential bandwidth savings that may be gained. The study also highlighted the problem of implementing a cache-less approach due to peers being behind NATs. We deal with this issue explicitly in Flashback and the design of an UDP based protocol and interval based chunk representation is a direct ramification of this constraint.

There have also been many other Peer-to-Peer approaches to solve the flash crowd problem but these operate with the peers acting as caches, i.e., a cache-based approach, for example, Coral [13], Squirrel [16], Kache [20], Backslash [27], PROOFS [28], and Dijjer [11]. Squirrel is designed to exploit organizational level peers while Coral and Kache are more generic. All four approaches use a distributed hash table (DHT) ([6]) as their fundamental data structure. In these P2P web-cache approaches, volunteer nodes form a distributed cache. When a end-user needs some content, it has to first contact one of the peers in the cache. This is accomplished either by requiring the user to rewrite the URL (as in Coral and Dijjer) or installing a proxy (as in Squirrel). Once a peer is contacted and the URL request made to it, the peer performs a DHT 'lookup' to see if any other peer already has the web-page. If no other peer is deemed to have the web-page, the peer gets the web-page from the main web-site and this is then returned to the end user. The web-page may also replicated during this search process so that the popular a web page becomes, the larger number of nodes it is cached in. The P2P web-cache is usually comprised of volunteer machines and not the end clients that are requesting a particular page. Therefore, even though the web-cache may be large it is not self-scaling and may still become a bottleneck. Secondly, if the web-page is set not to be cached by the web-site the web-cache just adds to the latency in getting the web-page. Third, in the design of these systems, it is assumed that the volunteer machines are relatively long lived. Kache is unique in this respect that it explicitly tackles the issue of high churn when volunteer machines are short lived. The lifetimes assumed in Kache, though, are still an order of magnitude larger than that assumed in Flashback. In Kache, the authors show that their systems performs well even when the churn rate is 10%-25% change to the P2P overlay in 200 seconds. When the churn rate is increased so that 10%-25% of nodes leave in under 40s, the system starts to 'trash'. In comparison, 90% of the overlay can change in under 10 seconds in Flashback and nodes are still able to get the download the file with low latency *and* in a tit-for-tat manner. The issue of dynamicity or churn in P2P networks has also been studied in itself [19], [7], [9], [29].

While P2P web-cache approaches to solve the flash crowd problem are somewhat new, the design of WWW has long supported infrastructure-based caches. These are machines that are maintained by organizations or ISPs and they cache web-objects that are accessed by the members of the organization. Infrastructure caches have many benefits including low latency, bandwidth savings to both the organization and the primary web-server and also importantly, transparency to the end user. These cache machines can be either standalone or participate in a larger cache network. How to form these cache-networks, the performance issues involved, etc. has received a large amount of research study (we defer to the survey in [31]). In this context, we wish to say that Flashback is not intended as a replacement for these infrastructure caches but rather 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.

Future Work: Flash crowds can appear inspite of web-caches due to low hit-rate, a wide and distributed set of people who want the content or low capacity of the web-server itself. Further, this is exacerbated with growing dynamic and customized content. Bandwidth bottleneck is only part of the larger problem of a web site being unable to disseminate information to a large audience. If the web-page is generated dynamically, the CPU can also become a bottleneck and there is active research on ways to tackle this [15], [25], [32].

Flashback currently does not address either dynamic content or streaming content. Since the end user browser becomes a tiny web-server in Flashback, it is not inconceiveable that Flashback can be used to handle dynamic data but the nature of the problem changes dramatically since all peers are no longer interested in the same content. Handling streaming content may be easier, especially since peers may now stay longer in the system thereby causing less churn. We are currently exploring both these ideas.

VII. 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 and use-cases.

References

[1] Bittorrent: http://bitconjurer.org/bittorrent/.

[2] Gnutella: http://gnutella.wego.com.

[3] Inet: http://topology.eecs.umich.edu/inet/.

[4] Modelnet: http://issg.cs.duke.edu/modelnet.html.

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

[6] Balakrishnan, H. Looking up data in p2p systems. In *Communications of the ACM (CACM)* (2002).

[7] Bhagwan, R., Savage, S., and Voelker, G. M. Understanding availability. In *International Workshop on Peer-to-Peer Systems (IPTPS)* (2003).

[8] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP* (2003).

[9] DESHPANDE, M., AND VENKATASUBRAMANIAN, N. The different dimensions of dynamicity. In *P2P* (2004).

[10] DESHPANDE, M., XING, B., LAZARDIS, I., HORE, B., VENKATASUBRAMANIAN, N., AND MEHROTRA, S. Crew: A gossip-based flash-dissemination system. In *ICDCS* (2006).

[11] DIJJER. http://dijjer.org.

[12] FORD, B., SRISURESH, P., AND KEGEL, D. Peer-to-peer communication across network address translators. In *USENIX* (2005).

[13] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIERES, D. Democratizing content publication with coral. In *NSDI* (2004).

[14] HANSON, E. N., AND JOHNSON, T. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures* (1992).

[15] IYENGAR, A., RAMASWAMY, L., AND SCHROEDER, B. Techniques for efficiently serving and caching dynamic web content. In *Book chapter in Web Content Delivery, Springer* (2005).

[16] IYER, S., ROWSTROM, A., AND DRUSCHEL, P. Squirrel: A decentralized peer-to-peer web cache. In *PODC* (2002).

[17] KONG, K., AND GHOSAL, D. Mitigating server-side congestion in the internet through psuedoserving. *IEEE/ACM Transactions on Networking 7*, 4 (1999).

[18] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Usenix Symposium on Operating Systems Principles (SOSP)* (2003).

[19] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *PODC* (2002), pp. 233–242.

[20] LINGA, P., GUPTA, I., AND BIRMAN, K. Kache: Peer-to-peer web caching using kelips. *ACM Transactions on Information Systems (under submission)* (2004).

[21] PADMANABHAN, V. N., AND SRIPANIDKULCHAI, K. The case for cooperative networking. In *IPTPS* (2001).

[22] PATEL, J. A., AND GUPTA, I. Overhaul: Extending http to combact flash crowds. In *WCW* (2004).

[23] PUGH, W. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM. Vol 33.* (1990).

[24] RABINOVICH, M., AND WANG, H. Dhttp: An efficient and cache-friendly transfer protocol for web traffic. In *INFOCOM* (2001).

[25] RAMASWAMY, L., LIU, L., AND IYENGAR, A. Cache clouds: Cooperative caching of dynamic documents in edge networks. In *ICDCS* (2005).

[26] SELVIDGE, P. How long is too long to wait for a website to load?

[27] STADING, T., MANIATIS, P., AND BAKER, M. Peer-to-peer caching schemes to address flash crowds. In *IPTPS* (2002).

[28] STAVROU, A., AND RUBENSTEIN, D. A lightweight, robust p2p system to handle flash crowds. *IEEE Journal on Selected Areas in Communications 22*, 1 (2004).

[29] STUTZBACH, D., AND REJAIE, R. Understanding churn in peer-to-peer networks. In *IMC* (2006).

[30] STUTZBACH, D., ZAPPALA, D., AND REJAIE, R. Swarming: Scalable content delivery for the masses. In *Techincal Report, University of Oregon* (2004).

[31] WANG, J. A survey of web caching schemes for the internet. In *ACM Computer Communications Review* (1999).

[32] ZHAO, W., AND SCHULZRINNE, H. Dotslash: handling web hotspots at dynamic content web sites. In *INFOCOM* (2005).