The Cyclone Server Architecture: Streamlining Delivery of Popular Content

Stanislav Rost
stanrost@lcs.mit.edu
MIT Laboratory for Computer Science
Cambridge, Massachusetts

John Byers
byers@cs.bu.edu

Azer Bestavros
best@cs.bu.edu

Dept. of Computer Science
Boston University
Boston, Massachusetts

Abstract—

We propose a new webserver architecture optimized for delivery of large, popular files. Delivery of such files currently pose a scalability problem for conventional content providers, which must devote server-side resources in direct proportion to the high multiprogramming level induced by a set of these connections. While use of scalable multicast may remedy this problem some day, multicast is rarely supported in today's wide-area infrastructure.

Our approach alleviates many of the most serious scalability problems by developing new server-side mechanisms capable of managing a large set of TCP connections transporting the same content. The strategy we employ relies on the use of fast forward error correcting (FEC) codes to generate encodings of popular content, of which only a small sliding window is cached in memory at any time instant. The concurrent TCP connections then access content only from this shared window, which is globally useful to all clients. Our method hinges on eliminating unscalable TCP retransmission buffers, as we can "retransmit" fresh encoding packets in lieu of the originals with no performance degradation and with no modifications to client TCP stacks. Ultimately, our Cyclone server capitalizes on concurrency to maximize sharing of state across different request threads while minimizing context switching, disk I/O, thrashing under high load and the cache memory footprint. In this paper, we describe the design and prototype implementation of our approach as a Linux kernel subsystem.

Keywords: *TCP, FEC, webserver, popularity, concurrency, digital fountain, Tornado codes*

I. Introduction

Burgeoning demand for content on the world-wide web has led to radical changes in the manner in which popular content is delivered over the Internet. Today, through the use of various proactive replication and redirection schemes, requests for popular content are often routed to, and served from, dedicated servers. Whether such replication and redirection schemes are deployed over a LAN, such as within a local web server cluster, or in the wide-area, such as across a global Content Distribution Network (CDN), the implications are similar—namely, a server must be able to respond to a very large number of requests for a relatively small number of disproportionately popular files.

While various technologies and protocols that enable replication and redirection have been implemented and deployed (reverse proxies, content surrogates, etc.), the architectures of web servers have not undergone radical changes to accommodate the delivery of very popular content. In this paper, we argue for such a radical departure—namely, in-kernel support for buffer and network stack management that enables a server to efficiently service a massive number of concurrent requests for files which are both *large* and *frequently requested*. It is well known that such files exist on many popular web sites. Studies on Web traffic characterization indicate that the frequency of web requests follows a Zipf-like distribution, or in other words, the distribution of total requests for a given file increases according to a power-law relationship with respect to its popularity ranking [10], [4]. Similarly, file sizes served by web servers have also been shown to follow a similar heavy-tailed distribution [15], [4]. Taken together, when we have files which reside at the "tails" of both of these distributions, such as in the case of movies in a video jukebox or zipped files at a software distribution site, we must address the challenge of serving files which are both very large and very popular. Moreover, serving the large, popular files at the tails of these heavy-tailed distributions is often where scalable content delivery services have the most difficulty [32].

Our solution meets this challenge head-on, by dedicating servers to be responsible for delivery of the largest, most popular files, freeing up the other servers in a server farm for smaller, or less intensely popular files. We expect that our approach would naturally be coupled with traffic engineering solutions that concentrate requests for large, popular files to our specially optimized servers.

The premise of the server architecture we propose in this paper is to reduce the storage complexity of serving a massive number of requests for the same content to almost a constant. To achieve this requires (1) a near-optimal sharing of memory between concurrent requests, and (2) a near-stateless network stack. We achieve both of these goals through the conceptual use of an idealized digital fountain [12], which encodes n blocks of original content into $k \cdot n$ encoding blocks for $k >> 1$; receiving *any* n distinct encoding blocks allows the complete, efficient reconstruction of the source data. For large files, close approximations to an ideal digital fountain can be achieved with fast forward error correcting codes, such as Tornado codes [22], which we employ. Traditionally, encoded content has been most widely used in multicast applications, where transmission of a redundant symbol enables different receivers to recover from different packet losses. In our setting, encoding the content realizes a a new benefit, namely while it still enables each piece of encoded content to be useful to all clients simultaneously, in so doing, it facilitates the sharing of information and reduces per-client state at the server.

Compared to traditional architectures, the Cyclone Server Architecture (CSA) allows a significant reduction in resource requirements for delivery of content in situations in which a group of clients is concurrently downloading a particular file, without requiring similarity of transfer rates or synchronization of request arrivals among the clients. In CSA, files are encoded into circular arrays of Tornado blocks that are stored on disk prior to their delivery. The benefits result from the ability of CSA to service an arbitrarily large group of clients interested in the same file using a single, fixed-length sliding cache buffer. Such a buffer is replenished with data from the circular encoding on disk at the speed of the fastest client of the group of clients interested in the same content. Connections servicing the group's slower clients draw data from the same buffer at their respective speeds and thus may miss a number of contiguous encoding blocks. However, the slower clients are able to reconstruct the missed content from redundancy contained in other blocks that they receive successfully.

An almost stateless network stack is achieved by elimination of bulky TCP retransmission queues, made possible by high utility of transmission of any block of the erasure encoding not previously sent. CSA employs a modified version of TCP retransmission semantics (no changes to standard client TCP network stacks are needed) that retransmits the next unsent block of the encoding from the shared sliding cache buffer as opposed to drawing the data from a retransmission queue.

Finally, the need for in-kernel implementation is driven by the necessity of precise synchronization of the speed of progression of the shared buffer to that of the connection to the fastest client of the group, and desire to capitalize on the benefits of elimination of the retransmission queues. Although OS kernel support is required in the server, the decoding functionality of the client can be implemented in a straightforward manner at the application level, for instance as a browser plug-in. Based on the above architectural features, we have implemented a web server subsystem (under Linux), named Cyclone, for use by server applications.

The rest of the paper is organized as follows. Section 2 outlines related work. In Section 3, we describe the design objectives for our method of content delivery. In Section 4, we present an overview of the design of the Cyclone architecture. Section 5 describes our implementation of the Cyclone subsystem in the Linux kernel. Section 7 contains analytical models ans results of simulations. And finally, in Sections 7 and 8 we elaborate on future work and provide acknowledgments.

II. Related Work

Considerable work has focused on optimizing the delivery of popular content from a *single* server. A common theme of this related work is to address issues of scale by minimizing resource consumption, often by reducing the marginal cost of serving an additional concurrent request.

Multicast-based Techniques: One widely researched method for delivering popular files with a minimal footprint on a single server leverages network mechanisms such as native multicast. Researchers have built scalable solutions which use non-adaptive, cyclic transmissions over multicast or broadcast channels to provide eventual reliability [2], [1], and more sophisticated solutions which employ forward error correction to achieve scalable reliability without a significant performance penalty [12], [26]. These solutions scale to large audiences, as the marginal cost of adding an additional client as perceived by the server is near zero.

The main disadvantage of multicast-based content delivery techniques is their reliance on the existence of an end-to-end multicast-enabled infrastructure. As a result, these techniques are better suited for enterprise network environments, as opposed to WAN environments. This is clearly evident in the slow deployment/adoption of such techniques on the Internet, and the emergence of alternative distribution protocols that "emulate" multicast using unicast-based overlay networks [20], [14].

Unicast-based Techniques: To improve the scalability of Internet servers[1] in a unicast environment, recent research efforts have focused on operating system optimizations (e.g. memory subsystem, file system, network stack, etc.) [29], [28], [25], [19], [18]. Generally, these optimizations fall under two categories: (1) optimizations that improve resource allocation decisions, and (2) optimizations that boost resource utilization.

[1] In this paper, we use the term Internet servers to refer to web servers, caches, proxies, reverse proxies, surrogates, etc.

Examples of approaches that aim to improve resource allocation decisions include the use of better cache management [13], [10], [9], [8], [24], [21] or the use of prefetching [16], [27]. Examples of approaches that aim to improve resource utilization include the elimination of unnecessary memory transfers between the various layers in the system (user space, kernel space, network buffers, etc.) [29] and avoiding overloading through proper admission control [33].

Scalable Content Delivery Engines: To put our work in context, we compare it to recent projects that parallel *some* aspects of our two-pronged approach to the construction of popular content delivery servers. Recall that our strategy relies on (1) an (almost) perfect sharing of memory between concurrent requests to a server, and (2) an (almost) stateless network stack implementation on the server.

The IO-Lite system [29] is a good example of a recent effort that aim to achieve the first of the above two goals in the context of Internet servers [28]. In that work, Pai, Druschel and Zwaenepoel demonstrate that repeated copying and multiple buffering of data is a major detriment to system performance. Specifically, their work exposed the unnecessary overhead of copying file blocks from the file system in the kernel layer to the application process' memory space, and then duplicating the very same data buffers in the network layers, as done in conventional operating systems. In [29], they propose a unified cache architecture to remedy this problem in which buffers containing file data are shared across the layers of the operating system. This work differs from ours in that it focuses on eliminating redundant data copying among a single service thread and various OS subsystems; in addition to that, our approach enables sharing *across* threads as well.

The Digital Fountain approach described in the introduction is an example of a recent effort that aims to achieve the second of the above two goals in the context of Internet servers. In one instantiation of that approach, Byers, Luby, Mitzenmacher, and Rege propose the use of fast error correcting codes [22] to alleviate the need for per-client state information at the source of a reliable multicast transmission [12]. Another instantiation of this approach uses encoded content to facilitate stateless downloads from multiple mirror sites in parallel [11]. Our work also employs this paradigm. However, while previous work primarily attempts to improve the utilization of network resources; our approach targets servers in a unicast environment and aims to improve utilization of a single server's resources.

III. Design Objectives: An Ideal Compact Cache

In the scenario described in the introduction, our objective is to efficiently satisfy a massive number of concurrent requests to a small number of popular files. In this scenario, we expect the server's main memory cache space to be a scarce resource whose usage is at a premium, i.e. it is infeasible to cache the entire working set. Nevertheless, the cost of I/O operations, which is orders of magnitude slower than operations to the cache, make optimizing cache utilization essential to delivering the highest performance. Our objective is to eliminate cache misses in the constrained service environment in which a set S of popular files of variable

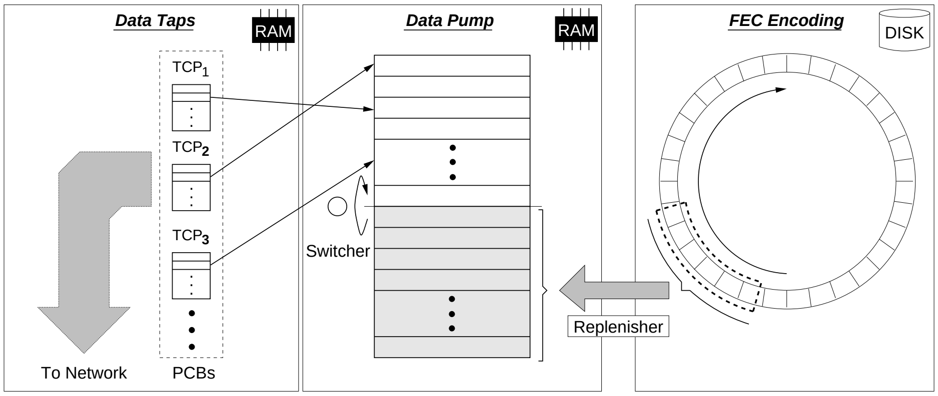


Fig. 1. Delivery of a file in the Cyclone server architecture

length is fixed in advance, but where the cache memory of size M is of sufficient size to store only a fraction of S. Thus the challenge is to partition the cache across files and efficiently utilize the *compact* cache space apportioned to file F in a manner which is *globally useful* to all request threads for F even as the multiprogramming level grows large. Issues of scale are central to our design, so apportionment should not necessarily be based on the size or popularity of a file. Expressed in another way, the challenge is to design a scalable system in which the marginal cost of serving an additional client requesting a file from the set S is as close to zero as possible.

From the perspective of managing cache resources in the context of delivering a particular large file, an ideal solution would therefore:

- Consume a fixed amount of cache memory regardless of the number of clients interested in the file.
- Allow the amount of cache memory to be considerably smaller than the size of the entire file.
- Provide performance benefits comparable to that which can be achieved by conventionally caching the entire file.
- Admit clients which arrive asynchronously.
- Accommodate heterogeneous client transfer rates without a performance penalty.

To recap, the properties of the ideal solution can be achieved by a delivery method utilizing a sliding buffer that is globally useful. Global usefulness of the buffer implies that its contents always contribute to the transfer progress of all clients downloading the file whose data is in the buffer. We describe our technique for achieving global usefulness and other components of our design in the subsequent section.

IV. Design Overview

A. Compact Caching via Fast Erasure Codes

Our strategy for achieving global usefulness of the sliding buffer employs a powerful procedure originally described by Rabin [30]. His Information Dispersal Algorithm (IDA), disperses a file F of length ℓ into n pieces F_i, $1 \leq i \leq n$, each of length ℓ/m such that the original file F can be reconstructed from *any* m pieces, where $n > m$. This technique, which can be realized in practice with forward error correcting codes, has been widely used to enable a dispersed encoding of a file to be transmitted over a lossy channel to one or many receivers. For this application, even if up to $n - m$ pieces of the encoding are lost in transit to a given receiver, that receiver may recover from the remaining pieces which arrive intact.

In our application, use of a dispersed encoding gives our server considerable freedom, and enables it to maintain cache contents which are globally useful with very high probability, as we shall now describe. First, for a given file F, the server chooses m so that ℓ/m is equivalent in size to maximum packet payload and generates an information dispersed encoding of the file of length $n = cm$, where we refer to c as the *stretch factor* of the encoding, as in [12]. Note that this encoding procedure need only be done once for each file as its output, the encoded content, could be stored on disk for future use. Also note that, since dispersed encoding contains redundancy, strict ordering of its blocks is unnecessary and the encoding can be considered *circular*.

Secondly, the server allocates a *sliding cache buffer* able to hold a fixed quantity (k) of contiguous pieces of the encoding and fills it with initial k blocks of encoding of F from disk. All connections transferring F draw data from that cache buffer. In the steady state, the server continually furnishes the next k blocks of the encoding from disk into the cache buffer whenever the fastest connection delivering F exhausts all of the blocks in the buffer. When the server reaches the end of the file storing the dispersed encoding of F, it may wrap around to draw data from the beginning of the file, thus rotating the cache buffer in the circular encoding.

Using this procedure with high concurrency levels, a typical client would arrive asynchronously, access pieces of encoded content from the cache as the sliding window rotates, and ideally receive sufficiently many pieces of the encoding to recover the file before a full rotation through the encoding completes. Note that the replenishment of pieces of the encoding will occur far faster than their retrieval by the slower clients, thus slow clients will typically "miss" many pieces of the encoding, tolerance to which is one of the key points of our design. Another important design consideration is the use of a sufficiently large stretch factor to ensure that most clients do not experience a full rotation, which can induce redundant transmissions and commensurate performance degradation. Details of the fast error correcting codes we use for encoding are discussed in the implementation section.

B. The Cyclone Subsystem

We now provide an overview of the design decisions we have made in building our Cyclone content delivery architecture which achieves the design objectives specified in the previous section. A high-level depiction of the mechanisms this architecture uses to deliver a single source file is provided in Figure 1.

The functionality of the Cyclone server architecture is made available to server applications by the Cyclone subsystem. The subsystem houses centralized collections of state uniquely pertinent to delivery of a particular file, the *data pumps*. Data pumps are responsible for maintenance of the file-specific cache buffer. Conceptually, a data pump ensures that fresh content from the FEC encoding is serviced to outbound transport connections, or data taps. As depicted in Figure 1, the data pump employs a sliding window strategy to ensure that fresh content is replenished into the cache. To avoid costs of memory locking while replenishment is occurring, we divide the cache buffer used by a data pump into two *half-spaces*. Each half-space contains pieces of the encoding which are atomically furnished to the data taps (for simplicity, recall that we set the encoding granularity equal to the size of a packet payload). The active half-space is the source of encoding packets to be sent out to the clients. The loading half-space (shaded in Figure 1) is inaccessible to service threads and can only be accessed by the data pump's *replenisher* thread, whose task is to load encoding packets from disk while packets are being concurrently furnished from the active half-space.

Collections of state which allow connections to interface with the data pump are referred to as *data taps*. A data tap tracks how much of the active half-space content has been transmitted by its parent connection. Service threads, each progressing at the speed driven by the congestion-controlled connection to the client, consult the data taps to determine which encoding blocks to read next and update them to reflect the progress. As soon as any thread exhausts all of the blocks in the active half-space, it requests that the half-spaces be switched. When the switch occurs, the loading and active half-spaces reverse roles, the data pump directs the replenisher thread to fill the new loading half-space, all data taps attached to the pump reset to the first block, and the service threads proceed to seamlessly serve encoding blocks from the new active half-space.

One integrity constraint is imposed on the system to preserve logical separation of the half-spaces. The half-space switch will not be performed until the replenisher thread has finished loading all of the encoding blocks into the non-active half-space. In practice, this constraint does not impose a significant performance cost since there are typically orders-of-magnitude difference between the time required to replenish a half-space and the time needed by a client to exhaust a half-space.

C. Interfacing with Reliable Transport Protocols

Of central importance to the design of the Cyclone server is the choice of, and interface to, the transport protocol we use to deliver the content. TCP is the most natural choice, as it is a standard for reliable unicast transfers and it is desirable to employ its congestion control functionality. However, it is highly undesirable to use TCP's stateful retransmission semantics, first because explicit retransmission of data is unnecessary with the encoding framework we use, but more significantly, because retransmission of packets requires undesirable retention of state in the form of a retransmission buffer, which must necessarily be of size proportional to the bandwidth-delay product of the TCP connection.

We propose two methods for circumventing this problem. The conceptually simpler version relies on UDP coupled with TCP-friendly congestion-control mechanisms, as could be provided by the Congestion Manager (CM) architecture [3]. In this scenario, packet retransmissions would be unnecessary for providing reliability, as receipt of a sufficient number of packets from the cyclic encoding would ensure eventual reliability. However, in the contemporary Internet, use of UDP is undesirable, both because many firewalls are configured to block UDP traffic and because many UDP transfers are not congestion controlled and are therefore discriminated against.

Therefore, we rely on a second transport-level approach, in which we make a server-side modification to the TCP packet retransmission algorithms which does not alter TCP timing semantics, but does alter the *content* of TCP retransmissions. In practice, we issue retransmissions which contain a different piece of the encoding than the original transmission. We emphasize that this alteration of TCP, which we call TCP-ERC, or TCP for erasure-resilient content, requires no client modifications, as TCP clients never examine discrepancies between content of an original transmission and a retransmission. Moreover, since our changes do not impact the timing semantics nor the critical path [6] of a TCP transfer, there is no negative performance impact.

Delving into the details of TCP-ERC, all of the flow control and congestion control mechanisms of TCP are retained in full. Conceptually, TCP-ERC differs from regular TCP in only one respect. In a situation in which TCP would retransmit a packet due to loss, TCP-ERC proceeds to transmit a fresh encoding packet (in the Cyclone server, that means simply transmitting the next Tornado block from the active half-space). On the packet level, the sequence number of the retransmitted packet is the same as that of the original lost packet, but a different portion of the encoding is inserted as the retransmitted packet's payload. This approach has the primary advantage that cumbersome (and superfluous) retransmission buffers do not have to be maintained in memory, substantially streamlining the management of TCP connections. At the client side, a client could conceivably observe the fact that TCP-ERC is being used at the sender, i.e. when it receives an original transmission and a retransmission with identical sequence numbers which contain different payloads. However, the TCP specification indicates that duplicate packets (i.e. packets with identical sequence numbers to packets which have already arrived) should be dropped. Thus clients can retain and use their existing TCP stack implementations to receive TCP-ERC transmissions.

D. Client Support for FEC-encoded Content

Transmission of encoded content implies that clients must have the capability to reconstruct the file from the encoded transmission. A client may advertise such a capability in the header of its HTTP request. The ability to reconstruct files from FEC encodings is entrusted to a library of subroutines positioned between the application and the communications protocol. The reconstruction procedure takes over the processing of incoming blocks of the encoding for the application, and supplies the application with a fully reconstructed file. A potential cost of this procedure is that this layer must be able to buffer the entire file prior to decoding. It is also the responsibility of the client to notify the server that it has received enough data to reconstruct the file by closing the connection or transmitting a stop message. The functionality of the reconstruction library can be made available to existing web browsers as a plug-in for processing of a MIME data type designating erasure-resilient encoding.

E. Design Summary

At the core of the Cyclone server architecture are files stored as erasure-resilient encodings. The advantage given by such an encoding is the ability to serve pre-encoded file blocks in any order, yet still contribute to the transfer progress of each of its clients with high probability. It is then possible to advance the sliding cache buffer at the speed of the *fastest* client in the group of clients interested in the same file, without waiting for slower clients. Slower clients who receive encoding blocks out of sequence due to an advancing sliding buffer can nevertheless reconstruct the file efficiently. In the event that a client witnesses the sliding buffer traverse the entire encoding, it is likely that this client will receive some redundant transmissions before recovering the file. However, large stretch factors can mitigate the effects caused by this contingency at the expense of increased secondary storage requirements at the server and increased decoding times (for detailed treatment, please see the Section VI-C).

The description of the design is now complete. The architecture discussed above achieves both maximization of sharing and cache compactness. However, more details are necessary to provide a full understanding of the Cyclone server architecture implementation.

V. Implementation Overview

This section of the paper presents the overview of a prototype implementation of the Cyclone server architecture as an in-kernel subsystem.

A. Codes

While the general IDA approach proposed by Rabin is very powerful, a substantial practical limitation is imposed by the computational complexity of encoding and decoding operations. Most authors applying IDA approaches to networking applications have used variants of Reed-Solomon codes (as described in [23]) as their forward error correcting primitive [7], [31], [26]. These codes rely on cumbersome finite field operations and have $O(n^2)$ encoding and decoding times (where n is the number of source blocks). Thus, while IDA with Reed-Solomon codes works well for reconstructing a file stored as n pieces distributed over n remote sites when n is 14 and m (the number of pieces needed to recover the file) is 10, its decoding inefficiency is prohibitive when transmitting a file spanning 5000 packets encoded with a stretch factor of 10. A partial remedy to this problem, which improves decoding time, is to divide a large file into blocks and to apply Reed-Solomon forward error correction over those smaller blocks [26]. In order to reconstruct the file, this approach necessitates recovery of a set of encoded pieces which together enable the decoding of all blocks. While this approach improves decoding, it limits scalability, as the reconstructor typically must wait a considerable amount of time before receiving the last codeword needed to reconstruct the final incomplete block, especially as the number of blocks grows large, as detailed in [12].

A newly available alternative to Reed-Solomon codes are sparse codes such as Tornado codes [22], which are closely related to low-density parity-check codes. By relaxing the decoding guarantee, i.e. by requiring slightly more than n encoding packets to recover an original file of n packets, and by using randomized encoding and decoding algorithms with fast XOR operations, these codes achieve linear encoding and decoding times. In practice, Tornado codes require reception of $(1 + \epsilon)n$ distinct encoding packets to reconstruct the source data, where ϵ is on the order of 0.05 for files of tens of megabytes or more. We employ these codes for our application, specifically, the family of Tornado Z codes, described in [12], which have an average overhead of roughly 5%, low overhead variance, and decoding times of at most a few seconds for files of tens of MB. A crucial parameter in selecting an appropriate Tornado code is the *stretch factor*, or the ratio between the length of the encoding and the length of the file. Employing a large stretch factor is highly desirable as it enables us to ideally accomodate a wider range of heterogeneous client transfer rates, as we describe momentarily. On the other hand, for our application, increasing the stretch factor causes a commensurate (linear) increase in (a) the disk space needed to store the encoding at the server, (b) the space needed to reconstruct the file at the client, and (c) the decoding time at the client. Unlike blocked Reed-Solomon codes, Tornado codes admit large stretch factors with only a linear degradation in encoding and decoding times and no degradation in overhead, measured as useless packets.

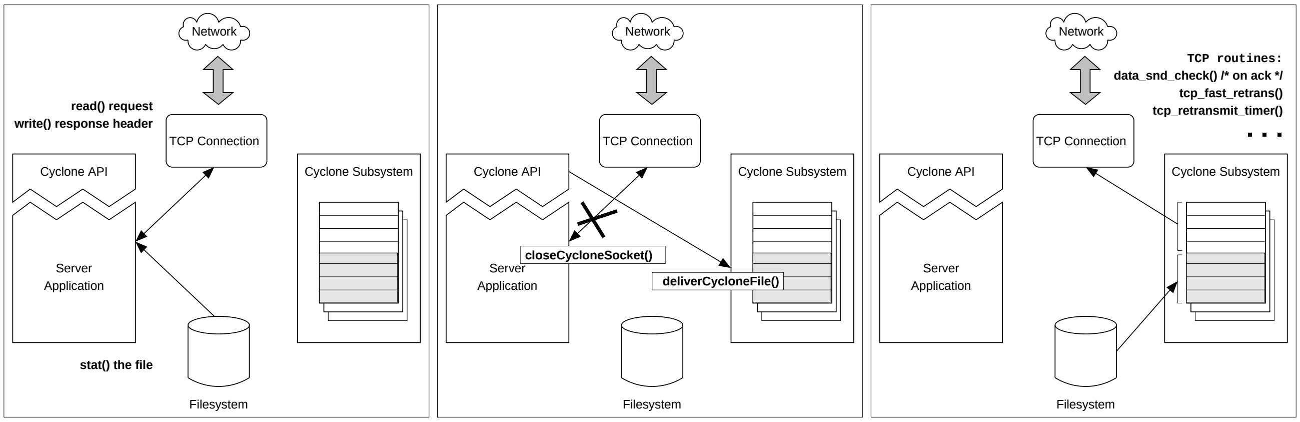
As an example of the tradeoffs, if clients are downloading a 10MB file at rates varying from 50Kbps to 1Mbps, the refresh rate of the cache will occur at a rate fast enough to accomodate the fastest client, while a slow client will transfer only about $\frac{1}{20}$ of the packets in a given cache block. In this scenario, a stretch factor of 20 would be needed to ensure that a slow client did not observe a full rotation of the encoding. If a full rotation does occur, this causes performance degradation, as it implies that clients may receive duplicate transmissions. However, the incidence of duplicate transmission will remain low – even with a stretch factor of 10, and assuming pure random transmission of encoding packets (with replacement), the incidence of duplicate packets will remain below 6% using Tornado Z codes. In practice, a stretch factor of 10 is the parameter setting we employ. For further discussion on duplicate transmissions, refer to Section VI-C.

B. Overview: Delegation of Processing

The Cyclone server subsystem performs the majority of the processing associated with content delivery. The task of the server application is reduced to processing a client's request and then invoking the Cyclone subsystem to deliver the requested content via the existing connection to the client. In the course of content delivery by the Cyclone subsystem, a replenisher thread loads data into the data pump corresponding to the file, and the in-kernel network layer event-driven routines transmit the content and invoke the switching of half-spaces. The process of satisfying a request for content using the Cyclone architecture is depicted in Figure 2, which we describe now.

The functionality of the Cyclone subsystem is provided to server applications through a system call. After transmitting a response header to the client (Figure 2(a)), the server application invokes the Cyclone system call (Figure 2(b)), passing the descriptor to the open socket as well as the name of the file to be delivered as parameters. The system call resolves the file name to the corresponding data pump (creating one if it does not exist yet), create a data tap, integrate it into the socket state, and connect it to the data pump. Once the system call returns, the thread of the server application that invoked it may entirely disregard the remainder of the delivery process and process the next request. Furthermore, it is recommended that the server application should close its socket to conserve per-process resources.

The Cyclone subsystem begins the delivery of encoded content after the system call returns control to the application and the re-



(a) (b) (c)

Fig. 2. Satisfying a request for content using Cyclone/Linux

sponse header has been successfully transmitted. The event-driven kernel routines (Figure 2(c)) responsible for the transmission of data in conventional delivery schemes also handle the special-case processing for Cyclone connections, discerning them from regular connections by presence of data taps. Such special-case processing involves procuring data from the pump as opposed to network buffers whenever the kernel routines can transmit or must retransmit.

C. Data Pump Management

As described in the Section IV-B, data pumps contain centralized cache buffers, for simultaneous access by multiple threads of service. They must then exist outside of the scope and lifetime of any particular service thread. Additionally, data in half-spaces will be accessed by time-critical kernel routines, and thus must be situated in memory which will not be swapped out to disk. These constraints compelled us to allocate data pumps as well as the hash table for resolving data pumps from file names within kernel-space memory.

Destruction of a data pump is invoked by the server application, and may either be immediate or delayed. In either case, the actual destruction only happens after all of the data taps detach from the pump, but in the case of delayed destruction the existing clients tapped into the data pump are first allowed to finish receiving the file. Note that the costs of creating and destroying data pumps are significant and often it is more beneficial to allow a data pump to idle than to destroy it.

The role of a replenisher thread is to furnish data to a data pump's loading half-space. In our implementation, one replenisher thread is created for every data pump, however, it is possible to improve upon that scheme. Once a replenisher thread fills the loading half-space, it will block until connections exhaust the active half-space and switch the half-spaces to resume loading the data. In practice, due to the difference between the throughput of secondary storage and that of network, the replenisher will often wait for ample periods of time. Instead of blocking, a single thread may be able to assume the role of multiple replenishers for a number of data pumps.

VI. Analytical Models

The primary benefits of our design result from unified cache and network stack management. Centralized, globally useful cache buffers prevent the amount of cache memory consumed by concurrent service threads from growing with the number of threads. Additionally, using such buffers reduces the cost of operations such as data copying, disk I/O and related system calls which would otherwise be performed by each service thread individually. Aggregation of the network stack with the globally useful cache alleviates the need for bulky send buffers and retransmission queues.

To analyze the resource savings of the Cyclone architecture over a conventional architecture, we will apply the following model. We assume that at an instant in time, a server (either conventional or Cyclone) is serving n asynchronous connections, $C_1 ... C_n$, which concurrently deliver a file F of length l bytes to their respective clients. Each connection transmits data at a heterogeneous rate R_i, over a path whose one-way latency is L_i.

A. Cache Buffer Scalability

A conventional server architecture delivers a file F by repeatedly loading a block of size B into a buffer and transmitting its contents.[2] Subdivision of a file into blocks of size B yields $N_B = \frac{l}{B}$ blocks.

At any instant of time, a connection C_i demands data from one of those blocks, and since connections are asynchronous and independent, we model the mapping of requests to blocks as being mapped independently and uniformly at random. In the event that multiple connections access a common block, we assume that a single instance of the block can be stored in memory and can be shared, as in case of memory-mapped disk access.

Then, for a file of N_B blocks and a multiprogramming level n, total memory consumption in this model is the following occupancy problem: How many blocks are covered by the set of n requests? The probability of a block *not* being covered is $\left(1 - \frac{1}{N_B}\right)^n$. Therefore the expected number of blocks which *are* covered is $N_B\left(1 - \left(1 - \frac{1}{N_B}\right)^n\right)$. When n is small relative to N_B, the expected fraction of blocks covered is closely approximated by $\frac{n}{N_B}$, or, coverage grows linearly with the multiprogram-

[2] Such an assumption is valid regardless of whether the conventional server relies on *write()* or *mmap()* system calls for file access.

ming level. As the multiprogramming level becomes large relative to the number of blocks, the limiting behavior is that the entire file is covered. On many modern operating systems, server architectures which do not take advantage of memory mapping cannot share file blocks beyond the boundaries of the file system cache. In such cases, the memory requirements are more simply expressed by $n \cdot B$. On the other hand, in the Cyclone server architecture, total memory consumption is only $2 \cdot H$, where H is the size of a half-space, independent of the number of clients n.

We next analyze the rate of I/O traffic in the two systems. Because of the regulation of TCP send queue sizes by the operating system, a service thread servicing a connection C_i will access the file at a rate ultimately determined by R_i. Thus, in a conventional system, the total rate of disk access for n service threads can be modeled by $\sum_i \frac{R_i}{B}$. Then, assuming transmission rates R_i are independent and identically distributed from some distribution $\mathcal{D}$, where $\mu = \mathbf{E}[R_i]$, the average rate of disk access in the conventional server can be represented as

$$\frac{n \cdot \mu}{B}.$$

In contrast, the Cyclone server's disk access to a file is driven by the rate of transmission of the *fastest* connection servicing F. The rate of disk access in the Cyclone server architecture can then be expressed as

$$\frac{\max_i R_i}{H}.$$

B. Network Stack Scalability

While our arguments apply to any transmission protocol that buffers content in the server, we also discuss the issues of network stack scalability within the framework of TCP.

The scalability of a server's TCP layer is limited by potentially large quantities of data temporarily residing in send buffers and retransmission queues of TCP stacks of concurrent connections. A TCP buffer can conceptually be subdivided into a send queue and retransmission queue. Content supplied by the application, but not yet transmitted onto the network, resides in the send queue. Content already transmitted, but not yet acknowledged, resides in the retransmission queue. The difference between the application's fill rate and the network connection's transmission rate dictates the size and behavior of the send queue. Many operating systems impose a limit on the send queue size and decelerate the fill rates of applications which exceed such a limit. TCP's sliding window algorithm and retransmission policy limits the size of the retransmission queue of a connection C_i to be at most $2 \cdot R_i \cdot L_i$, with the steady-state limit on the lower bound of the queue size being $R_i \cdot L_i$.

Typical memory demands imposed on a server by a TCP stack can be illustrated with a back-of-the-envelope example. As the Internet technologies evolve, the available bandwidth will increase while propagation delays, dictated by bounds on the speed of light, will likely remain the same. A connection with 100 ms RTT and available bandwidth of tens of Mbps will yield retransmission queue sizes in the range of hundreds of kilobytes per connection.

A TCP-ERC stack coupled with the Cyclone server does not feature a send buffer, but instead draws data from the shared cache buffer. Unlike in the case of TCP retransmission queues, the impact of connection properties on memory requirements of a TCP-ERC stack is negligible. Connection-influenced memory consumption in a TCP-ERC stack occurs primarily due to the need to store per-packet timestamps and selective acknowledgment flags in a lightweight retransmission queue, to ensure proper functioning of TCP. Under Linux, TCP-ERC's lightweight retransmission buffers require 16 bytes per packet as opposed to regular TCP's requirements of ≈ 1600 per socket buffer. By removing timestamps from the TCP-ERC stack and performing timeouts using a constant amount of state, it may be possible to eliminate the need for per-packet state, providing even better memory scalability.

C. Duplicate Transmissions

One of the factors affecting the performance of the Cyclone server architecture is the potential transmission of duplicate encoding packets. The cyclical manner in which the replenisher retrieves encoding from disk provides for the possibility of slower connections transmitting such packets. Connections may transmit redundant encoding blocks after persisting for more than one rotation of the sliding buffer over the entire length of the encoding.

The performance issues associated with the finite length of the encoding can be mitigated by selecting a large enough value of the stretch factor prior to encoding the content. The choice of the stretch factor's value is crucial for eliminating redundant transmissions, but also impacts storage costs and decoding time.

It is straightforward to calculate the conditions under which a client can reconstruct the source content before a full rotation completes, namely if it has a hit rate of at least $\frac{1+\epsilon}{c}$ (where $1+\epsilon$ is the reception inefficiency of the Tornado code, and c is the stretch factor) in the first rotation. In the Cyclone server, situations occur in which slower clients miss blocks due to a faster connection's faster advancement of the shared sliding buffer. In the course of delivery of the content in a given half-space, the slower connection C_i will only be able to transmit an $\frac{R_i}{\max_j R_j}$ fraction of the half-space content before the fastest connection will switch the half-space. In fact, on average, C_i will miss a $1 - \frac{R_i}{\max_j R_j}$ fraction of the entire encoding during every revolution of the sliding buffer. Thus it is possible to guarantee that C_i will transmit a sufficient fraction of the encoding to reconstruct the source file before the rotation completes if the following condition holds:

$$\frac{R_i}{\max_j R_j} > \frac{1+\epsilon}{c}.$$

The stretch factors needed to realize this guarantee for a wide range of transmission rates are impractically large. In practice, considerably smaller stretch factors can be employed in combination with scalable heuristics aimed to reduce redundant transmissions. One such heuristic ensures that slow clients do not always deterministically send from the front of half-spaces, but rather choose the

$$r = \frac{R_i}{\max_j R_j} \cdot N_h$$

blocks that they send during every "round" of transmission (with such rounds beginning and ending in a half-space switch) at random. Alternatively, clients might choose a random initial offset from within each half-space. Other, more complicated heuristics may provide better protection against repeated transmissions of identical blocks at the expense of additional memory.

VII. Conclusions and Future Work

We have described an architecture for a scalable content delivery engine dedicated to transmission of large, popular files to a broad audience using TCP transport. The novelty in our design stems from erasure-resilient encoding of the content, which facilitates maximal sharing of transmitted packet payloads across concurrent request threads and keeps per-connection state to a minimum.

We have implemented this Cyclone subsystem in the Linux kernel and have validated the correctness of our version of TCP-ERC implemented in the Linux TCP stack. We have developed server and client APIs suitable for testing the performance of Cyclone transfers. These APIs are integrated into Apache web server [17] and SURGE, a representative web workload generator [5].

Currently, we are conducting a series of experiments to demonstrate the performance advantages of Cyclone over traditional server architectures. The initial experiments involve the scenario of a single relatively large, popular file being requested by multiple clients. Such tests allow us to debug, profile and optimize our implementation.

In the future, we would like to test our architecture in multi-file scenarios, in the settings of both local and wide-area networks. We are also exploring various avenues for application of globally useful shared buffers, lightweight network stacks and other components of the Cyclone server architecture.

VIII. Acknowledgments

Many thanks to Mark Crovella, Paul Barford, and Bob Frangioso for helpful conversations, useful advice and technical suggestions.

References

[1] S. Acharya, M. Franklin, and S. Zdonik. Dissemination based data delivery using broadcast disks. In *IEEE Personal Communications*, pages 50–60, December 1995.

[2] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of web pages using cyclic best-effort (UDP) multicast. In *Proceedings of IEEE INFOCOM*, March 1998.

[3] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *Proceedings of ACM SIGCOMM '99*, September 1999.

[4] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web, Special Issue on Characterization and Performance Evaluation*, 2:15–28, 1999.

[5] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *ACM Sigmetrics*, 1998.

[6] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[7] A. Bestavros. AIDA-based Real-Time Fault-Tolerant Broadcast Disks. In *Proceedings of the 16th Real Time System Symposium*, June 1996.

[8] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad. Application level document caching in the internet. In *IEEE SDNE'96*, June 1996.

[9] J-C. Bolot, S. Lamblot, and A. Simonian. Design of efficient caching schemes for the world wide web. In *ITC 15*, Washington, DC, June 1997.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM*, New York, NY, March 1999.

[11] J. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *Proceedings of IEEE INFOCOM*, New York, March 1999.

[12] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM*, Vancouver, Canada, 1998.

[13] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Annual Technical Conference*, December 1997.

[14] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics '2000*, pages 1–12, Santa Clara, CA, June 2000.

[15] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, December 1997.

[16] L. Fan, Q. Jacobson, and P. Cao. Potential and limits of web prefetching between low-bandwidth clients and proxies. In *ACM Sigmetrics*, 1999.

[17] Apache Foundation. Apache web server.
URL = **http://www.apache.org**.

[18] J. Hu, S. Mungee, and D. C. Schmidt. Techniques for developing and measuring high-performance web servers over ATM networks. In *Proceedings of IEEE INFOCOM*, San Francisco, CA, March 1998.

[19] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the Apache web server. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC'99)*, Phoenix/Scottsdale, Arizona, February 1999.

[20] J. Jannotti, D. Gifford, K. Johnson, M. F. Kaashoek, and J. O'Toole. Reliable multicasting with an overlay network. In *Proceedings of OSDI*, San Diego, CA, October 2000.

[21] S. Jin and A. Bestavros. Greedydual* web caching algorithm: Exploiting the two sources of temporal locality in web request stream. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[22] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th STOC*, May 1997.

[23] F. J. Macwilliams and N. Sloane. *The Theory of Error-Correcting Codes*. North Holland, Amsterdam, 1977.

[24] E. P. Markatos. Main memory caching of web documents. In *Proc. of the 5th World-Wide Web Conference*, May 1996.

[25] E. N. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. In *ACM Sigmetrics*, 1999.

[26] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of ACM SIGCOMM '97*, September 1997.

[27] V. Padmanabhan and J. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, July 1996.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[29] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified

I/O buffering and caching system. In *Proceedings of OSDI*, 1999.

[30] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 38:335–348, 1989.

[31] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review*, 2(27):24–36, April 1997.

[32] E. Schooler and J. Gemmell. Using multicast FEC to solve the midnight madness problem. Technical report, Microsoft Research, September 1997.

[33] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation for cluster-based network servers. In *Proceedings of IEEE INFOCOM*, 2001.