The Cyclone Server Architecture: Streamlining Delivery of Popular Content

Stanislav Rost
prgrssor@cs.bu.edu

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 technique for efficiently delivering popular content from information repositories with bounded file caches. Our strategy relies on the use of fast erasure codes (a.k.a. forward error correcting codes) to generate encodings of popular files, of which only a small sliding window is cached at any time instant, even to satisfy an unbounded number of asynchronous requests for the file. Our approach capitalizes on concurrency to maximize sharing of state across different request threads while minimizing cache memory utilization. Additional reduction in resource requirements arises from providing for a lightweight version of the network stack.

In this paper, we describe the design and implementation of our approach as a Linux kernel subsystem.

I. Introduction

The 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. Specifically, through the use of proactive replication and various redirection schemes, requests for popular content are routed to, and served from, dedicated servers—either in a local-area setting (within a web server cluster) or in a wide-area setting (within a Content Distribution Network). Whether such replication and redirection schemes are deployed over a LAN (e.g. layer-7 TCP routing [40]) or a WAN (e.g. content "Akamaization" [24]), the implications are similar—namely, a server must be able to respond to a very large number of requests for a relatively small amount of disproportionately popular data.

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 frequently requested and relatively large. Files which are much more frequently requested than average exist on many popular web sites. For example, studies on Web traffic characterization [12], [6] indicate that web requests follow a Zipf-like distribution, which leads to the distribution of total requests for a given file increasing according to a power-law relationship with respect to its popularity ranking. Similarly, large files also occur naturally, such as in the case of a server hosting video clips or a software distribution site. In general, file sizes have been shown to follow a heavy-tailed distribution [16], [6], which provides for the possibility of many sites featuring large content. It is also noteworthy that serving the large, popular files at the tails of these two heavy-tailed distributions is precisely where scalable content delivery services have the most difficulty. Our solution for serving these files can be coupled with traffic engineering solutions that concentrate requests for large, popular files to our specially optimized servers. Such a solution would dedicate servers to be responsible for delivery of the "tails" [16] of the file size distributions, and would free up the other servers in a server farm for smaller, or less intensely popular files.

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 use of Tornado codes [29], [28], erasure codes which encode n blocks of original content into $k \cdot n$ blocks of the encoding that contain redundant information; receiving any $n + \epsilon$ distinct encoding blocks allows the complete reconstruction of the source data, where ϵ is on the order of $0.05n$ for sufficiently large files, i.e. tens of megabytes or more.

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 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 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.[1]

The rest of the paper is organized as follows. In the remainder of this section, we discuss related work. Section 2 outlines related work. In Section 3, we describe the design objectives for our method for 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. And finally, in Sections 6 and 7 we elaborate on future work and provide acknowledgments.

[1]The Cyclone server architecture requires its clients to perform application-level decoding, but does not require or introduce any changes to the clients' kernel or network stack implementations.

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 [13], [34]. 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 [26], [23].

Unicast-based Techniques: To improve the scalability of Internet servers[2] in a unicast environment, recent research efforts have focused on operating system optimizations (e.g. memory subsystem, file system, network stack, etc.) [37], [36], [33], [22], [21]. Generally, these optimizations fall under two categories: (1) optimizations that improve resource allocation decisions, and (2) optimizations that boost resource utilization.

Examples of approaches that aim to improve resource allocation decisions include the use of better cache management [14], [12], [11], [10], [31], [27] or the use of prefetching [17], [35], [19]. 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.) [37] and avoiding overloading through proper admission control [41].

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

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

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 [37] 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 [36]. 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 [37], 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; whereas our approach enables sharing *across* threads as well.

The Digital Fountain approach [13] is a good example of a recent effort that aims to achieve the second of the above two goals in the context of Internet servers in a *multicast* environment. In that work, Byers, Luby, Mitzenmacher, and Rege propose the use of fast error correcting codes [29], [28] to alleviate the need for per-client state information at the source of a reliable multicast transmission. This work differs from ours in that it is aimed at multicast environments; whereas our approach targets servers in a unicast environment.

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 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, popularity of content and I/O operations, which are 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 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 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 [38]. 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 [13]. Note that this encoding procedure need only be done

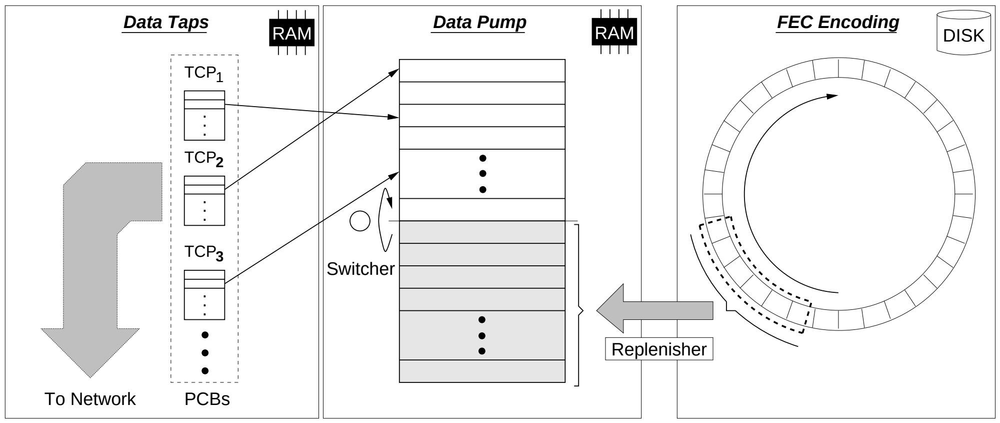

Fig. 1. Delivery of a file in the Cyclone server architecture

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 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 replenishing 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. The 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-space 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 [5]. 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 [8] 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 [25] 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 [20] 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 unusual event that a client witnesses the sliding buffer traverse the entire encoding, then 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.

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

This section of the paper presents the details of implementation of the proposed 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 [30]) as their forward error correcting primitive [9], [39], [34]. These codes rely on cumbersome finite field operations and have O(n^2) encoding and decoding times. 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 [34]. 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 [13].

A newly available alternative to Reed-Solomon codes are sparse codes such as Tornado codes [29], which are closely related to parity-check codes. By relaxing the decoding guarantee, i.e. by requiring slightly more than m encoding packets to recover an original file of m packets, and by using randomized encoding and decoding algorithms with fast XOR operations, these codes achieve linear encoding and decoding times. We employ these codes for our application, specifically, the family of Tornado Z codes, described in [13], 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.

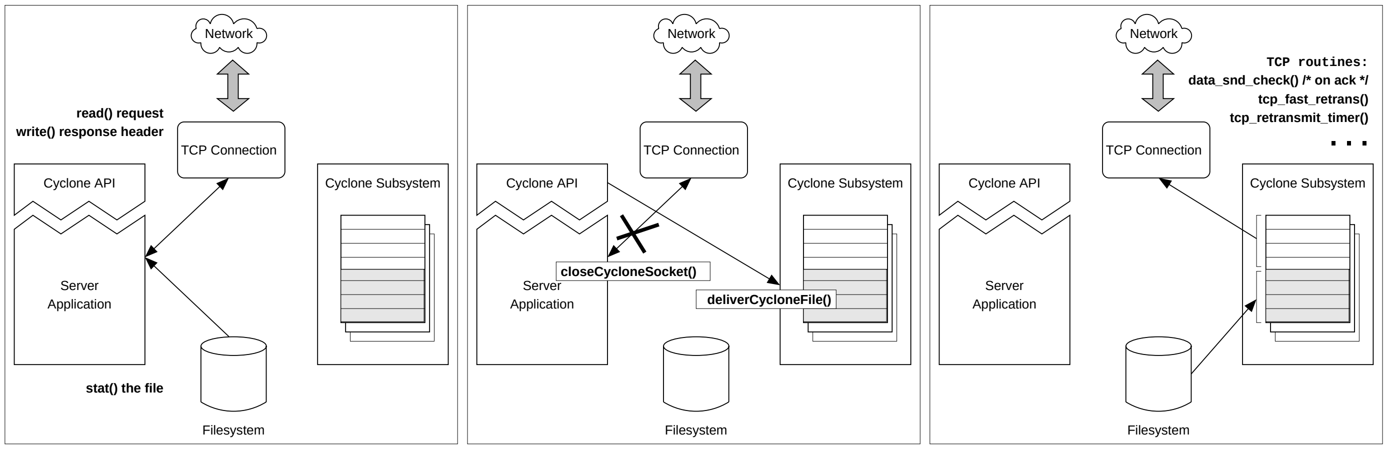

Fig. 2. Satisfying a request for content using Cyclone/Linux

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 will invoke 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 will then resolve 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 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 response 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 the 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.

D. Synchronization

The following constraints pertain to switching and must be preserved via synchronization:

- Half-space switch must not occur until the replenisher is finished loading data into the non-active half-space.
- The half-space switch must be atomic, i.e. no connections may draw data from the active half-space when the switch occurs.
- The replenisher thread must be directed to fill the new loading half-space once the switch occurs.

The replenisher thread may be trivially synchronized with switching using semaphores. However, imposing blocking semantics on non-blocking network-layer processing that initiates half-space switching is non-trivial. Our solution involves the fastest connection raising a "need-to-switch" flag in the data pump once it transfers all data in the active half-space, all other connections deferring their processing by means of a "backlog" queue in the data pump if the "need-to-switch" flag is raised, and the replenisher thread actually performing the switch and executing the deferred processing on behalf of the connections once it awakens.

E. Data Taps and Network-Layer Processing

Collections of state allowing a connection to interface with a data pump are referred to as *taps* (they allow clients to tap into the data pump). In our implementation, tap data structures are stored in the transmission protocol control blocks and distinguish connections that require special-case processing.

Data taps contain at least four essential items:

- A pointer to the data pump for accessing the data in half-spaces.
- A version of the data pump that this tap has last accessed, to detect if the half-space switch has occurred since the last time this tap was active. The global version of the data pump is incremented whenever one of the taps attached to it commences the half-space switch.
- The index of the next block in the half-space to be transmitted.
- A lightweight queue storing timestamps of packet transmissions to enable TCP to estimate the round-trip time. Additionally, the lightweight queue stores SACK flags.

Our implementation of the Cyclone subsystem involves a modification of the Linux TCP stack to introduce special-case processing for Cyclone-enabled connections. Such connections mimic the behavior of regular TCP connections, with the exception of always behaving as though more data is available for transmission to the clients, drawing the blocks for transmission from half-spaces, and using the lightweight queue for RTT estimation, selective acknowledgment and retransmission.

Beyond the transmission of the first piece of the encoding, initiated within Cyclone system call by an application desiring to transfer a file using the Cyclone subsystem, further transmission of encoding data is initiated by TCP's transmission mechanisms, i.e. by client ACKs or timeouts.

Delivery of content through a particular Cyclone-enabled socket ceases when any of the following cases:

- The client closes the connection.
- A connection error occurs.
- The application explicitly shuts down the transfer using the Cyclone system call.

F. API

The following functions comprise the Cyclone subsystem server API:

○ `void initCyclone(void)`
Initializes the Cyclone subsystem's internal state.

○ `InternalSocketHandle closeCycloneSocket (int fileDescriptor)`
Allows the application to close the application-side portion of the open socket to preserve per-process resources. However, unlike a call to **close()**, the kernel-side portion of the socket remains open until the Cyclone subsystem closes it.

○ `void deliverCycloneFile (PumpDescriptor *desc, InternalSocketHandle ish)`
Transfers control of file delivery to the Cyclone subsystem. **PumpDescriptor** contains parameters such as the name of the file, the size of the block, and the number of blocks per half-space. This function either finds the data pump corresponding to the pump descriptor, or creates a new pump with the given parameters. Then it creates a data tap, integrates it into the socket state, and requests the transmission of the initial Tornado block, initiating the slow start of TCP.

○ `void shutDownPump (PumpDescriptor *desc)`
Marks the given data pump as "shutting down." Existing connections are allowed to finish the file delivery, then the pump is deallocated.

○ `void destroyPump(PumpDescriptor *desc)`
Immediately tears down connections drawing from the data pump, and deallocates the pump.

VI. 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 [18] and SURGE, a representative web workload generator [7].

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. Additional results, combined with the description of the architecture presented in this technical report, will comprise the basis for a full version of this paper.

VII. 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] M. Arlitt and C. Williamson. Trace-driven simulation of document caching strategies for Internet web servers. Technical report, Dept. of Computer Science, University of Saskatchewan, Saskatchewan, Canada, 1996.

[4] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions in Networking*, 5(5):631–646, October 1997.

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

[6] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in web client access patterns: Characteristics and caching implications. Technical Report BU CS Technical Report 98-023, Boston University, December 1998.

[7] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1998.

[8] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *SIGCOMM Symposium on Communications, Architectures and Protocols*, Stockholm, Sweden, January 2000.

[9] A. Bestavros. "aida-based real-time fault-tolerant broadcast disks". In *Proceedings of the 16th Real Time System Symposium*, June 1996.

[10] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad. Application level document caching in the internet. In *IEEE SDNE'96: The Second International Workshop on Services in Distributed and Networked Environments*, June 1996.

[11] 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.

[12] 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.

[13] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM Symposium on Communications, Architectures and Protocols*, Vancouver, Canada, 1998.

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

[15] Open Source Community. Linux documentation project. URL = **http://www.redhat.com/mirrors/LDP**.

[16] 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.

[17] Li Fan, Quinn Jacobson, and Pei Cao. Potential and limits of web prefetching between low-bandwidth clients and proxies. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1999.

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

[19] Robert Frangioso, 1999. Private communications.

[20] Network Working Group. RFCs 2045 - 2049: Multipurpose internet mail extensions. Work in Progress.

[21] 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.

[22] 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.

[23] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *Proceedings of ACM Sigmetrics '2000*, pages 1–12, Santa Clara, CA, June 2000.

[24] Akamai Technologies Inc. URL = **http://www.akamai.com**.

[25] Information Sciences Institute. RFC 793: Transmission control protocol. Work in Progress.

[26] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole. Reliable multicasting with an overlay network. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, October 2000.

[27] Shudong Jin and Azer 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.

[28] M. Luby. Practical erasure codes based on random irregular graphs. In *RANDOM*, 1998.

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

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

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

[32] J. C. Mogul. Operating system support for busy Internet servers. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[33] E. N. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. Technical report, IBM, February 1999.

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

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

[36] 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.

[37] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI)*, 1999.

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

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

[40] ArrowPoint Web Network Services (WebNS). URL = **http://www.arrowpoint.com/solutions/white_papers/WebNS.html**.

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