Anycast-Aware Transport for Content Delivery Networks

Zakaria Al-Qudah
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
zma@case.edu

Seungjoon Lee
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
slee@research.att.com

Michael Rabinovich
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106
misha@eecs.case.edu

Oliver Spatscheck
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
spatsch@research.att.com

Jacobus Van der Merwe
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
kobus@research.att.com

ABSTRACT

Anycast-based content delivery networks (CDNs) have many properties that make them ideal for the large scale distribution of content on the Internet. However, because routing changes can result in a change of the endpoint that terminates the TCP session, TCP session disruption remains a concern for anycast CDNs, especially for large file downloads. In this paper we demonstrate that this problem does not require any complex solutions. In particular, we present the design of a simple, yet efficient, mechanism to handle session disruptions due to endpoint changes. With our mechanism, a client can *continue* the download of the content from the point at which it was before the endpoint change. Furthermore, CDN servers purge the TCP connection state quickly to handle frequent switching with low system overhead.

We demonstrate experimentally the effectiveness of our proposed mechanism and show that more complex mechanisms are not required. Specifically, we find that our mechanism maintains high download throughput even with a reasonably high rate of endpoint switching, which is attractive for load balancing scenarios. Moreover, our results show that edge servers can purge TCP connection state after a single timeout-triggered retransmission without any tangible impact on ongoing connections. Besides improving server performance, this behavior improves the resiliency of the CDN to certain denial of service attacks.

Categories and Subject Descriptors

C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network Communications*

General Terms

Performance, Design

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

Keywords

content delivery networks, anycast, connection disruption

1. INTRODUCTION

A significant portion of electronic content in today's Internet is delivered through content delivery networks (CDNs), such as Akamai and Limelight. For example, Akamai alone claims to be delivering 20% of the world's Web traffic [1]. While the exact numbers are debatable, it is clear that CDNs play a crucial role in the modern Web infrastructure. Most current CDNs rely on DNS to redirect client requests to an appropriate edge server. That is, an edge server that is close to the requesting client and not overloaded. While efficient, this redirection mechanism has a few major drawbacks. First, it can only select the edge server based on its proximity to the client DNS server, not the client that will be performing the download, and clients (especially residential clients) are sometimes far removed from their DNS servers [16]. Second, this method affords a CDN only limited control over load balancing decisions: caching of DNS responses by the clients makes server selection decisions persist for a long time, regardless of the current load conditions. This problem is exacerbated by the widely spread practice of clients disobeying time-to-live directives returned with the DNS responses, which in principle control how long the clients can reuse a response, and using cached responses much longer [17]. Furthermore, malicious users can intentionally bypass DNS responses and stick to a particular server, potentially causing significant degradation of the system performance [21]. Finally, because local DNS servers cache results, not all DNS requests represent the same client workload. For example, we expect a significant difference in workload between a DNS server serving a small community college and a DNS server serving a large broadband access provider.

IP anycast is a promising alternative request distribution mechanism; it was initially considered for CDNs and dismissed, but recently re-emerged as a practical approach for single-network CDNs [6]. An anycast CDN assigns the same IP address to multiple edge servers and relies on IP routing to deliver requests to the servers that are close in the network to the clients originating the requests. In particular, the server selection decision in anycast CDNs is based on the

proximity to the client itself—not to its local DNS. On the other hand, anycast CDNs face two major problems (which were the reasons IP anycast was originally dismissed as a viable alternative): (1) lack of load balancing and (2) connection disruption in connection-oriented downloads. The first problem arises from the fact that IP routers do not consider server load when routing a request. As a result, servers that happen to be a little closer to many clients may become overloaded.

The second problem stems from the possibility of a route change in the middle of a transfer. When that happens, connection-oriented flows might break because they might be routed to a server that does not maintain the connection state needed to continue the transfer.

In our previous work, we proposed a new design to address the first problem of load balancing [6]. The focus of this paper is to address the second problem of connection disruption due to routing changes in anycast CDNs.

If a connection disruption interrupts a small object download, a client can simply request the object again without significant impact on user experience, as long as the disruptions are infrequent. In the case of large file downloads (e.g., software download or multimedia streaming over HTTP), for which disruptions can be costly, we proposed in [6] to make edge servers redirect the requests to the servers' own unicast addresses at the beginning of the download. This additional redirection is done via application-level redirection mechanisms, e.g., an HTTP redirection response. This approach in effect achieves *static binding* of the edge server to the client, taking the download out of the anycast scope.

While static binding is simple to deploy, its performance may suffer in many scenarios. First, large file downloads could last several hours, during which the network and load conditions may change significantly. Static binding would not allow re-assigning the download to a different server to respond to these changes. Second, in a flash crowd situation, a server may have admitted more requests than it could serve. With static binding, the server will potentially remain overloaded for a long period of time, even if the CDN makes more resources available. Third, when the demand increases from a given geographical area, the CDN may want to bring up a new server in that area to satisfy the demand locally even if current servers are not overloaded. We would wish that the CDN be able to redirect some ongoing downloads to the new server. This will enhance user experience as well as reduce the routing cost of these large files for the ISP. Fourth, by the time a flash crowd ends, the CDN might have allocated a large amount of resources to serve the flash crowd. With static binding, the CDN has to wait for all connections to a server to finish (which might take hours for large files) before decommissioning that server. The same problem occurs when a server needs to be taken down for maintenance. Lastly, with static binding, malicious users can degrade system performance in a similar way to DNS-based systems [21].

In this paper, we propose not to fight but *embrace* session disruption, by observing that redirecting long-running downloads to a new server can be an important tool in achieving agile server and network load management. We propose a very simple mechanism to handle session disruption. If a TCP session is disrupted, the browser or download manager reacts to the broken connection by issuing a new HTTP range request for the remaining bytes. The browser then reconstructs the entire file from pieces once the last portion of the file is obtained. The browser has all the information to construct this request: it obviously knows how many bytes it has obtained prior to the disruption and it can request the range from the subsequent byte to the end of the file. For multimedia downloads, the browser adds pieces of content to the playback buffer as they arrive. Note that multimedia streaming implemented as progressive HTTP download (e.g., a youtube video) is similar to other types of large file downloads as long as the download speed is higher than the playback speed. Therefore, we believe that our scheme works equally well for this type of traffic as well.

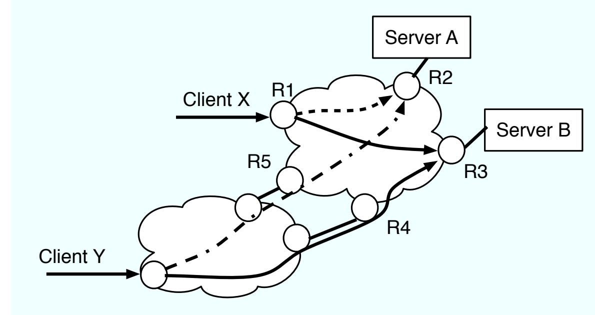


Figure 1: TCP session disruption due to routing changes in an anycast CDN

While the proposed scheme may appear simplistic, implementing it requires addressing several important questions. Specifically, in the rest of this paper, we address the following questions:

- How can we ensure the client's TCP stack will always learn of the connection disruption and issue the next range request? (Section 3.1.3)
- What is the implication of the dormant TCP state at the old servers for the connections that were moved to the new server? (Section 3.1.2)
- What is the implication of having multiple range requests on the overall download throughput? Does the overhead due to multiple TCP slow-starts require some mitigating mechanisms? (Section 3.2)
- What are the implications of our approach on server performance, especially when a routing change causes many new range requests to reach a server simultaneously? (Section 3.2)
- What are the security implications for a CDN that is employing our proposed mechanism? (Section 3.3)

The remainder of this paper is organized as follows. The next section provides necessary background on anycast CDNs and reviews related work. Section 3 describes our proposed mechanism and the motivation behind it in more details. In Section 4, we then present experimental results that evaluate our design. We conclude in Section 5.

2. BACKGROUND

This section presents necessary background information regarding anycast CDNs and their relationship with TCP. It then surveys the alternative mechanisms to support long-running downloads in an anycast CDN.

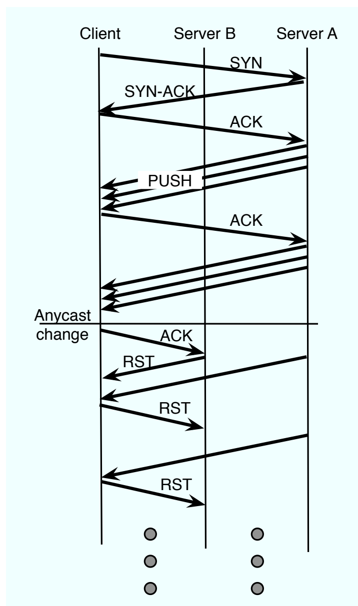


Figure 2: Default TCP behavior during anycast endpoint change

2.1 Anycast and TCP

IP anycast enables an IP routing and forwarding architecture to assign the same IP address to multiple endpoints. From a routing perspective this behavior is the same as having multiple paths to the *same* endpoint. This means that routers naturally handle multiple endpoint with the same IP address by selecting the shortest path from their perspective. As such IP anycast provides "optimal" (i.e., shortest path) routing for the delivery of packets to the anycast IP address.

Typically, such anycast endpoints are configured to provide the same service. For example, the multiple DNS root servers use a single IP anycast address for redundancy [8]. Among those endpoints, an IP anycast packet is routed to the "optimal" endpoint from an IP forwarding perspective (e.g., proximity). As a result, by configuring an IP anycast address for multiple CDN servers that can serve the same set of contents, we can get efficiency and redundancy at the same time, which was the reason why IP anycast was initially considered for CDNs. However, anycast CDNs face two major problems. First, since IP routing typically does not use application-level data, the routing decision can lead to server load imbalance. Our previous work [6] addresses this issue by using a fine-grained route controller [23] that monitors server load and informs route selection in the anycast enabled network to realize load aware anycast. Specifically, in this load aware anycast CDN, we override the default route selection process within the CDN network to affect load balancing between different CDN nodes.

Another problem with anycast CDNs is potential connection disruptions if a route change occurs in the middle of a connection-oriented transfer as in TCP. Figure 1 illustrates two types of route changes that could result in connection disruption. First, because of a routing change *external* to the anycast enabled network, anycast packets from client Y that first entered the network through router $R5$ now enters the network through $R4$. Assume that based on shortest path routing $R5$ will select the anycast route associated with $R2$ and server A, however, router $R4$ chooses the route associated with $R3$ and server B. Figure 2 illustrates the default TCP behavior if such a switch were to happen in the middle of a TCP transfer. TCP ACKs from the client will be redirected to a different server (B in this example), which has no TCP state for this connection. As a result, the server will send a TCP reset (RST) message to the client, who will in turn close the connection, thus terminating the transfer. Such an external route change happens infrequently in practice [7] and is therefore not a significant concern.

Figure 1 shows another type of route change which could result in TCP session reset. In this case anycast traffic from client X enters the network at router $R1$. Initially this router might select the anycast route associated with router $R2$ and server A. However, due to a routing change *internal* to the anycast enabled network, $R1$ might select the anycast route associated with router $R3$ and server B, resulting in similar behavior as discussed above. As described above, a load aware anycast CDN informs route selection based on the server load in the CDN. As such, this internal form of route change might happen much more frequently, resulting in TCP session resets. Dealing with this problem is the primary goal of the work presented in this paper.

A route change in the middle of a TCP transfer also creates a dormant state at the old server, which can waste significant system resources for a prolonged period of time. Indeed, if TCP does not receive an ACK within RTO (Retransmission Timeout), it retransmits the unacknowledged segment multiple times before it gives up. For example, Linux by default performs 15 retries. Because the time between each two retries is increased exponentially, the entire retry interval can be from 13 to 30 minutes [4]. In anycast CDNs, after the route change, all ACKs (or subsequent RSTs) from a client now go to a new server (B in Figure 1); without the ACKs, the old server (A in our example) will continue retransmitting the segments to the client for the full maximum retry interval.

2.2 Related Work

Our previous work [6] utilized two recent developments to propose a practical anycast CDN design that can balance the server load in a single-network CDN. First, the design exploits an observation that Internet routing is remarkably stable if all anycast endpoints belong to the same autonomous system (AS), so anycast traffic from the same external destination will overwhelmingly arrive at the same AS entry point [7]. Second, it employs recently developed mechanisms for fine-grain route control within an AS to periodically change internal IP routing within the AS based on the load of edge servers [23]. Our present work complements this anycast CDN design by providing a mechanism to deal with long-running download session disruptions due to routing changes, most of which will be the result of route changes to affect load balancing in the CDN.

As mentioned in Section 1, one way to handle routing changes in anycast CDNs is to use static binding based on HTTP redirection [6]. We can potentially apply other existing schemes to address the issue of connection disruption. One option is to use socket migration [10,20,22] and transfer the connection state between the old and new servers when a server redirection occurs. However, socket migration requires complex and reliable coordination between the new

and old server (and sometimes the client). Furthermore, it may incur the socket migration delays.

In addition to socket migration, another key element in the approach by Szymaniak et al. [22] utilizes mobility support in IPv6 to intentionally redirect the client to a new anycast end-point in the middle of an ongoing TCP connection. When IPv6 becomes available, this aspect can be combined with our approach to avoid socket migration.

Another way to handle connection disruption is to use a stateless transport protocol such as Trickles [19]. To enable the stateless operation for servers, Trickles encodes the connection state in every packet in both directions of the connection (data and acknowledgment packets). This entails a significant overhead in anycast CDNs scenarios, because Trickles needs to carry at least 87 bytes to every data and ACK segment even if a server switching does not occur.

Yet another possibility is to use a receiver-centric transport protocol [14], where the server is completely stateless, and the receiver keeps all the relevant connection state. In such a scenario, the server could simply be presented with a file name and an offset in that file to be able to send the next packet of the connection. This approach requires complex security protective measures as it puts the untrusted clients into the driving seat in terms of controlling the sending rate.

An alternative approach to improve the performance of large file downloads is to divide the file into smaller pieces and request each of these pieces in a separate connection in parallel [12, 18]. This approach is only effective when a single connection cannot fully utilize the link capacity of the client, that is, when the connection bottleneck is in the core network or at the CDN servers. However, in a CDN serving mostly residential clients that we target, the bottleneck is typically at the last mile to the client.

While CDNs are traditionally considered to show good resistance to DoS attacks, Su et.al. [21] have recently highlighted feasible and simple DoS attacks that can be launched against DNS-based CDN services such as Akamai's streaming service. Briefly, multimedia streams require large amount of network bandwidth and server resources to serve them. On the other hand, DNS redirections are extremely sluggish with redirections and decisions remain in effect for tens of seconds. During this period, an attacker can request a large number of streams from an edge server with the effect of overloading the network and/or the edge server. Since Akamai's CDN performs essentially a *static binding* between the attacker's machines and the edge servers, the CDN will not be able to offload these servers for the duration of these streams (potentially several hours). We note that this problem exists in anycast CDNs that perform static binding, but is avoided in CDNs that employ our approach.

3. ANYCAST-AWARE TRANSPORT

This section presents our mechanism for handling connection disruptions and examines, on a descriptive level, its performance and security implications. We quantify the performance implications in the next section.

3.1 Mechanisms

3.1.1 Client-side

We use a very simple mechanism to handle connection disruption during an HTTP session. When a client detects a TCP connection failure during an ongoing download, the client issues an HTTP range request for the remaining portion, assuming that the failure is due to a redirection to a different server. However, to distinguish a redirection from a true network outage, the client treats a connection failure during TCP handshake differently – the client aborts the download in this case. The client has all the information to construct the HTTP range request: it knows how many bytes it has obtained prior to the disruption and it can request the range from the subsequent byte to the end of the file.

Note that the client can potentially go through multiple rounds of connection disruptions and subsequent range requests before completing a download. In the end, the client reconstructs the entire object of interest from the multiple pieces obtained from potentially different servers or it can progressively add pieces to the playback buffer for multimedia streaming applications implemented as progressive HTTP downloads.

In practice, we can implement all the necessary changes on the client side at the *application layer* as exception handling in response to TCP socket errors. Failure to open the socket corresponds to handshake failure; the client handles this exception by aborting of the download. When getting an exception on read from the socket, the client stores the downloaded portion and issues a range request to the same IP address for the remaining bytes. Architecturally, these changes can be implemented as a browser extension or in a stand-alone "download manager" for a particular content provider (such as Apple's iTunes), which many content providers already require their users to install before they can access the provider's content.

3.1.2 Server-side

Functionally, no changes on the server side are required. However, our approach leads to a potentially significant performance penalty on the server side. After a server redirection, the old server maintains the open TCP connections and keeps retransmitting unacknowledged segments, while the acknowledgments all go to the new server. As mentioned in Section 2.1, this can go on for as long a time as 13 to 30 minutes and cause significant waste of server resources such as CPU, memory, and network bandwidth.

Furthermore, because servers have a limit on the number of concurrently open connections, these dormant connections reduce the number of useful TCP connections a server can process. Because server switching in an anycast CDN is performed through routing change, it is typically coarse-grained and affects hundreds of connections simultaneously [6]. Thus, a server may have a large number of dormant connections and need to wait for a significant amount of time before it can accept new connections. Yet we cannot just configure the server to accept more connections to support such dormant connections, because that would cause server overload and affect server performance in the common case where most of connections are active.

We address this problem by operating more aggressively by simply using a small value (e.g., 1) for the maximum retry count, thus closing dormant connections quickly. With typical round-trip times in a CDN are much less than the minimum RTO of 200ms stipulated by TCP, one retry translates into 600ms of retaining a dormant connection: 200ms before the retry and 400ms of waiting for the acknowledgment for the resent segment.

Such a drastic (from at least 13 minutes to 600 ms) reduction in connection retaining time certainly addresses the issue of dormant connections but immediately raises the issue of potential disruption of ongoing connections that may happen due to normal variation of network delays and not due to anycast redirection. However, our live Internet experiments show that this change rarely affects the ongoing connections in practice (Section 4.2). Even if connection disruptions occasionally occur, the client in our system will handle them gracefully by continuing the download session using HTTP range requests. In fact, our results may have implications beyond the immediate topic of this paper as discussed in Section 3.3.

We implement the above change by extending the socket interface on the server host to allow the application to specify the number of retries when opening a socket, on a per-socket basis.

3.1.3 Detecting Connection Disruption

As discussed earlier, the client relies on its TCP stack to detect connection disruption. Specifically, as shown in Figure 3(a), when a redirection occurs due to anycast routing change, (1) TCP ACKs from the client will go to a new server. Then, (2) the new server will respond with an RST message, which will cause an exception on a read from the socket. Based on this signal, (3) the client issues a new HTTP range request.

Note that the TCP ACK from the client or RST from the new server can get lost inside the network. In this case, data transmissions from the old server (e.g., due to other packets in transit or re-transmissions) will normally trigger another set of ACKs and RSTs between the client and the new server, as illustrated by the continued transmissions by server A in Figure 3(a). However, in rare cases when all (re)transmitted packets from the old server are lost in the network the connection may go silent from the client's perspective.

We implement another extension to the server TCP stack to make our approach more robust to the above boundary conditions. As shown in Figure 3(b), we make a server host send an RST segment to the client and thus explicitly indicate a connection disruption when the server gives up on the connection after retries (which happens quickly in our approach as described earlier in Section 3.1.2). Again, we implement this modification by extending the socket API so that this behavior can be enacted on a per-socket basis.

If all above fails to deliver an RST message to the client because they all were lost in the network, the client times out on the connection, and just issues a new HTTP range request assuming a connection disruption.

Note that packets from the two connections (before and after the anycast change) belong to totally different sockets and thus RSTs or ACKs from the old connection will not interfere with the new connection.

3.2 Performance Implications

Our seemingly simple approach may potentially have significant performance implications. Examining these implications is important to establishing the viability of our approach. In particular, we consider the following issues.

First, what is the implication of splitting a download into a sequence of range requests on the overall download throughput? When a client detects a new redirection, it needs to establish a new HTTP/TCP connection to a new server, which requires a three-way handshake and TCP slow start. This can potentially degrade the download performance in modern high-bandwidth environments, especially if the redirection occurs frequently.

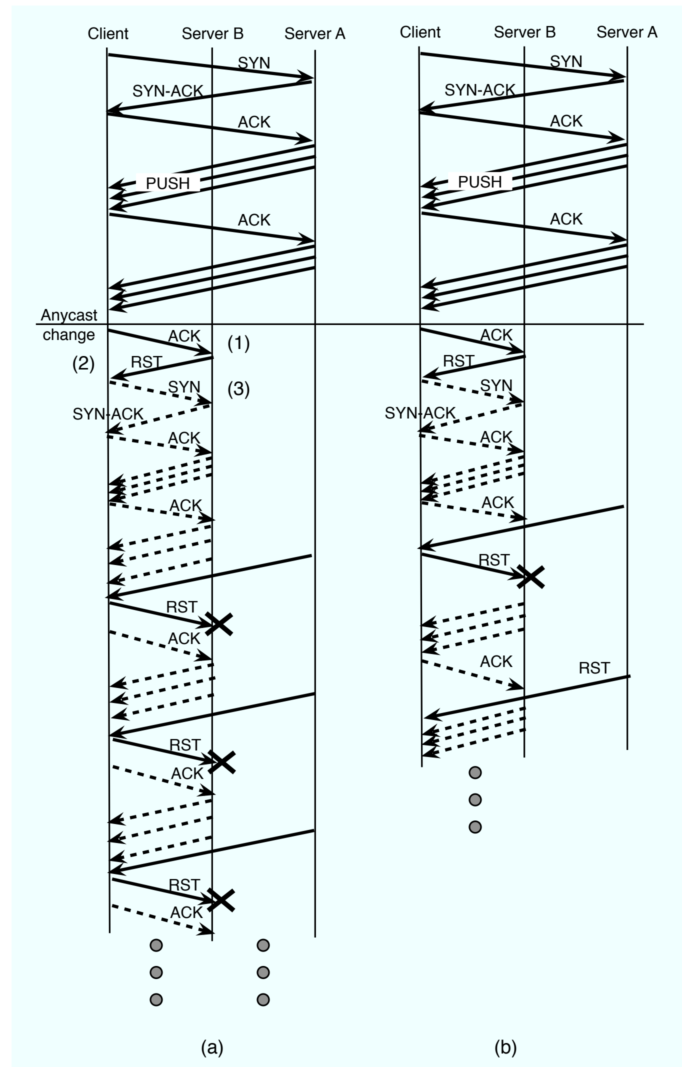


Figure 3: TCP behavior during anycast endpoint change: (a) with modified client (b) with modified client and server

One could attempt to mitigate this issue by maintaining estimates of path characteristics between edge servers and client subnets. Depending on the estimated parameters, a server may choose to skip slow start for a new flow and immediately enter the congestion avoidance phase. We in fact followed this path in our design, until our performance results indicated that the impact of repeated slow starts on the overall download throughput is marginal with the rate of redirections expected in a CDN (see Section 4.1). Thus, one of our findings is that no mitigation in this case is required.

Second, what is the implication of closing dormant TCP connections quickly? Doing so is essential for the well-being of CDN servers in our approach, yet it could potentially disrupt ongoing downloads in the absence of routing change.

However, our extensive experiments over the global Internet (Section 4.2) indicate to the contrary: these spurious disruptions are extremely rare. Coupled with the fact that our approach recovers gracefully from such a disruption, we conclude our quick purging of dormant connections to be appropriate, and in fact beneficial beyond the CDN setting (see Section 3.3).

Third, what are the implications of our approach on server performance? When we change anycast routes, all traffic to a given anycast address entering the network at a given entry router will move from one server to another. This means the new server will face a large number of simultaneous range requests from clients trying to recover their disrupted connections. *Will this cause the new TCP connections to adjust their sending rate in a synchronized manner?* Intuitively, if a large number of new connections all go through a slow start at the same time, they can potentially lead to network-wide oscillation and significant performance degradation of the network and server [9].

In our scenario, however, different flows have different RTTs, which should provide randomization to counter the synchronization effect. Furthermore, our experiments in Section 4.3 show that even in the ideal scenario for synchronization to occur, there is enough randomization due to host processing delays so that we could not observe any evidence of synchronization.

3.3 Security Implications

What are the security implications on a CDN that is employing our mechanism? As discussed in Section 2.2, CDNs have been recently shown to be vulnerable to DoS attacks targeting streaming services [21] such as Akamai's streaming service. The principle vulnerability stems from two issues: (1) sluggishness of DNS, and (2) the incapability of the CDN to offload overloaded servers. While the authors in [21] proposed some architectural modifications to Akamai's streaming service to mitigate this problem, our mechanism can effectively solve it. Specifically, our mechanism gives the CDN a tool with which it can offload the overloaded servers and reassign some of the ongoing downloads to new edge servers (and thus absorb the attack).

Furthermore, by shortening the connection timeout of servers, our mechanism provides another security enhancement. Long timeouts (e.g., up to 30 minutes in Linux) impose great DoS threats. An attacker can request a large number of downloads from a server using relatively small number of botnet nodes and then simply disappear. Whether the server is a CDN server or a traditional stand-alone web server, it then waits for a long time and tries to retransmit unacknowledged packets before dropping the connection state. During this time, significant resources (e.g., resources associated with the open TCP connection, a server process or thread with the associated memory responsible for the pending HTTP request, etc.) are allocated for each of these actually dead connections. Since we reduce these timeouts to a fraction of a second, the server can get rid of these connections and make the resources that were allocated for them available to other requests quickly.

Finally, anycast CDNs in general keep control over request routing with the CDN. In particular, in DNS-based CDNs, a client can simply override the DNS redirections altogether and flood a particular server with requests. In anycast CDNs, the client can submit a request, but it has no control over which server will serve this request. Equipped with our proposed mechanisms, anycast CDNs can enjoy a significantly better DoS attack resistance.

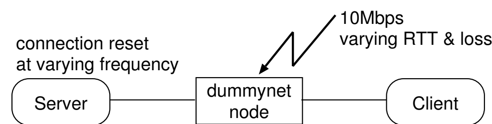


Figure 4: Experiment setup for connection resets

4. PERFORMANCE EVALUATION

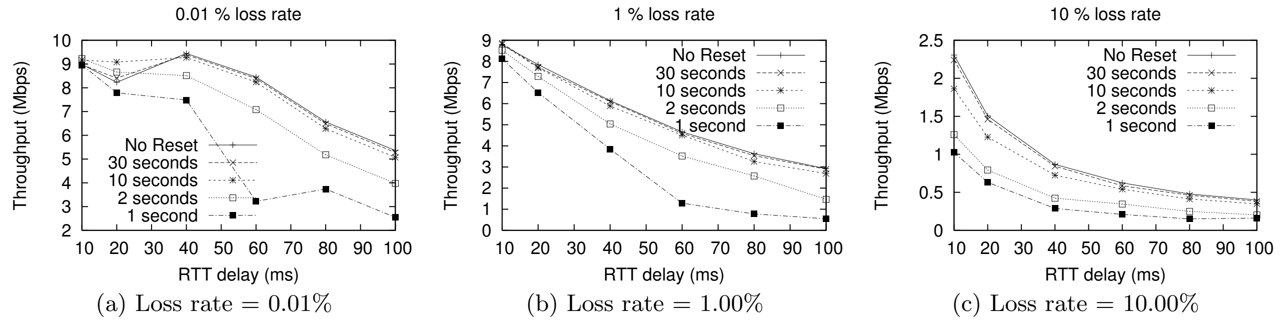
This section provides quantitative answers to the questions we raised earlier in the paper: what are the implications of TCP connection disruptions on the overall application-level download throughput; what is the effect of quick purging of dormant TCP connections on active connections; and what is the performance effect of moving HTTP downloads en-mass to a new edge server. We consider these issues in turn in the following subsections.

4.1 Impact of Connection Disruptions

In this section we evaluate the performance implications of splitting a large file download into a series of downloads via range requests. In particular, we assess the impact of this mechanism on the overall download throughput as experienced by individual clients. Every range request requires that the client reestablishes a connection to the new server. This requires the client to perform a new three-way connection establishment and the connection will need to go through the slow-start phase before reaching its full speed. Another perspective of the experiments in this subsection is to establish how often the anycast CDN can reconsider server load balancing and associated routing decisions without impacting client downloads.

To answer these questions, we conduct the following experiment. As shown in Figure 4, we have a server and a client machines connected via a FreeBSD dummynet [2] with 100 Mbps links and network interface cards. The bandwidth, however, is shaped by the dummynet to 10 Mbps. The client downloads a 50MB file during which we emulate the server switching by having the server reset the connection at a given frequency. We vary the connection reset (or disruption) frequency. We also experiment with diverse path characteristics by varying the loss and delay parameters at the dummynet node and observe the overall download throughput. The result is shown in Figure 5.

Each data point in Figure 5 is the average of 10 trials. The solid line (No Reset) represents a baseline case of the throughput achieved by a normal TCP connection that is never reset. When the connection is reset every 30 seconds, the throughput degradation of disrupted connections is marginal. The throughput of the disrupted connections is at most within 6% of the baseline throughput, and this extreme is only reached for unrealistically high 10% loss rate. Most of the data points show only 1% to 2% drop in throughput. A switching frequency of once every 30 seconds seems more than sufficient for anycast CDNs. For example, remapping connections to servers in [6] is done once very 120 seconds. When the server resets the connection very frequently (e.g., once every one or two seconds), the overall



(a) Loss rate = 0.01% (b) Loss rate = 1.00% (c) Loss rate = 10.00%

Figure 5: Overall throughput of a 50 MB file download with various path characteristics and connection reset frequency.

download throughput experiences a significant degradation especially for high loss or long RTT paths. The degradation becomes less pronounced for high loss *and* long RTT paths for which the normal TCP shows low throughput anyway. When the connection is reset once every 10 seconds, throughput degradation becomes less tangible especially for low loss paths. Specifically, at 0.01% and 1% path loss rates, the throughput of the disrupted connections is within 10% of the baseline, while, in some experiments, the disrupted connections achieve higher throughput than the baseline. At 10% loss rate, the throughput of the disrupted connections is within 19% of the baseline.

The marginal penalty under realistic rates of server switching makes any mitigation unnecessary in the current environments. Indeed, mitigation might be counter-productive. For example, one mitigation approach might involve a new server starting to serve a range request from a new client in the congestion avoidance (instead of slow start) phase using estimated path characteristics from the new server to the client subnet. The penalty due to inaccurate path estimation could outweigh the potential benefits. Specifically, if the server underestimates the actual path bandwidth, it will ramp up the sending rate slower in the congestion avoidance phase than using slow start. If it overestimates the bandwidth, it will incur unnecessary loss. Should we find situations in the future that require high switching frequency, we might have to revisit these design choices.

4.2 Quick State Purging

In addition to an endpoint change, normal variations in network delay and loss could potentially cause a server to miss client's ACKs for a period of time enough to cause server to retransmit packets. The purpose of this section is to answer the question of how small the maximum retry count can be set before the percentage of dropped connections due to normal network variation increases to an "unacceptable" value. Note that we have a large degree of freedom in defining what is an unacceptable rate because the consequence of a spurious connection drop is only that the client needs to reconnect with a range request. Therefore, from the perspective of the client, a spurious connection drop is equivalent to a server switching to the same server.

We address this question by performing live Internet experiments which are prepared as follows. We setup two servers next to each other: one server uses regular TCP (number of retries equals 15) while the other uses a short-

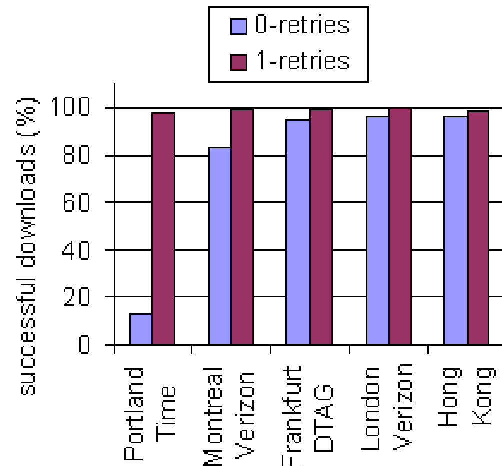


Figure 6: The five Keynote clients that achieved the worst successful download rate for 0 maximum retry count: For 1 retry, the the successful download rate increases dramatically.

timeout TCP (the number of retries is an experimental parameter with 0 retries corresponds to 1 RTO timeout, 1 retry corresponds to 3 RTOs timeout, and so on). We sign up for 59 Keynote [13] clients distributed across the Internet. (Most agents were in the United States, and some in Europe and Asia). Each client issues one request to download a 2 MB file from each of the two servers every 15 minutes in a randomized fashion. Logs of these downloads are maintained both at the sever (by us) and at the clients (by Keynote). The 2 MB file size restriction is imposed by Keynote. However, we verified the results of this experiment by downloading a 50 MB file from 3 residential machines that we own for various setting of maximum retry count. The results were consistent with the those obtained from Keynote with 2 MB file size.

We set the number of retries to 0 for the short-timeout TCP in the first experiment and to 1 in the second experiment. We ran each experiment for over a week and we collected over 33 thousand connections to each of the two servers. Table 1 summarizes the results of these experiments. The field "Log" in the table represents which log the results are extracted from (server logs or client logs).

Retries Count	Log	Percentage of dropped connections (%)		
		Regular TCP	Short-Timeout TCP	Difference
0	Server	0.145	2.57	2.42
0	Client	1.18	3.66	2.48
1	Server	0.006	0.167	0.161
1	Client	0.02	0.171	0.151

Table 1: Summary of live Internet results

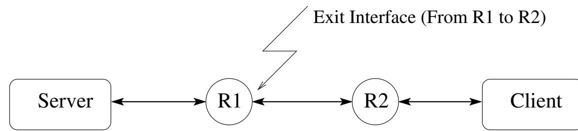


Figure 7: Experimental topology for the synchronization experiment

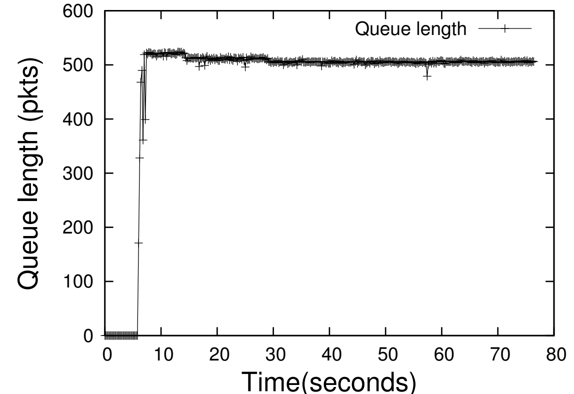


Figure 8: Queue length versus time. There is no evidence on synchronization.

While the relative difference between the two TCP configurations is similar in both logs, the actual average drop rates are different. This is attributed to the fact that some Keynote clients can fail due to for example client overload and network problems. While Keynote try to minimize failures caused by its clients/network [3] and try to compensate for these failures when reporting statistics from client logs, this creates a small discrepancy between client and server logs.

In the first experiment, the majority of the difference in connection drop rate is due to one client which was able to complete the download successfully only 13.11% of the time according to the client logs and 13.23% according to the server logs. The server timed out on the rest of the downloads. Looking at the the average file download time for this client, we notice that it is 19.2 seconds compared to 2.91 seconds the average download time of all other clients, which suggests that the path quality to this client seems to be extremely poor.

Interestingly, only a small increase in the timeout period results in a dramatic decrease in the average spurious connection drop rate. For example, Figure 6 plots the five clients that achieved the lowest rate of successful downloads (highest connection drop rate) with 0 retries. The figure shows that the setting of 1 maximum retry count dramatically increases the percentage of successful downloads for these clients. For example, even the earlier problem client is now able to successfully download the file 97.61% of the time despite its apparent poor network path.

With this extremely low drop ratio achieved by a maximum retry count of one, we conclude that setting the timeout to as short as three RTOs does not impact the performance due to spurious connection dropping. However, this quick timeout (typically a fraction of a second) helps the server clean up the dormant connections' state quickly and resist some DoS attacks.

4.3 Synchronization Concerns

An existing mechanism for an anycast CDN with load-balancing capability maps connections to edge servers at the granularity of an ingress router [6]. In other words, the load balancing controller (e.g., [6]) re-maps all connections coming into a particular ingress router to a new server.[1] The implication of such mapping granularity is that the new server might be faced with a burst of new TCP connections trying to perform range requests for the remaining part of their respective files. In such a situation, one concern is that the server might experience synchronization problems whereby many TCP connections ramp up and back off around the same time. On the other hand, random influences such as RTT variations would dampen any oscillations.

This oscillating behavior would result in low network and server utilization. Thus, one would need to implement mitigating traffic shaping at the server to prevent oscillations. It is therefore important to assess the likelihood of synchronization.

We design the following experiment for this purpose. We set up a client and a server with two routers in between as shown in Figure 7. R1 is a Linux box that is connected to Case Western Reserve University's campus network. Traceroute from R1 to the client (which is also at Case) shows that there is a one hop between R1 and the client and ping from R1 to the client shows roughly 0.2 ms round trip time. R1's network interface to the campus network is 100 Mbps. The traffic exiting the router interface (marked with "Exit interface" in the figure) is shaped using Linux TC [15] to 50 Mbps to create a bottleneck. We emulate a large number of connections arriving at the server at around the same time by starting 1000 connections from the client machine to download a 5.2 MB file from the server. The server machine runs

[1]One could extend the mechanism of [6] to allow remapping at the granularity of particular interfaces of ingress routers. However, this does not change the argument in this section because even this finer granularity still involves hundreds of connections.

Apache [5] web server that is configured to accept this large number of connections and serve them simultaneously.

The rationale behind this experiment is as follows. Our setup imitates the worst-case scenario where synchronization would be most likely to happen. Indeed, in our experiment, the network path characteristics are homogeneous among all the connections, thus any randomization due to RTT variation would be minimal. The fact that connections are originated from the same client machine creates some randomization that stems from resource sharing. However, this randomization should be small as compared to that available in the Internet with many flows competing for bandwidth. We plot the bottleneck queue length sampled every 200 ms. In situations where synchronization problem exists, we expect the bottleneck queue to oscillate for a long time.

Figure 8 shows the result of this experiment. Despite minimal RTT variations, what little randomization is contributed by host processing is sufficient to prevent synchronization as evidenced by the lack of oscillations in queue length. Again, we expect much higher randomization effect in a real deployment with variable RTTs. With this result, we believe that the new server does not need to handle switched connections in any special way.

5. CONCLUSIONS

In the paper, we address the issue of supporting long-running client sessions, such as large file downloads and multimedia streams implemented as progressive HTTP downloads, in anycast CDNs. Existing CDNs, including anycast CDNs, "pin" these sessions to a given server at the beginning of the download, regardless of the changing network and server load conditions. This reduces the adaptability of the CDN and, as shown recently [21], its resiliency to denial of service attacks. This paper presents our approach to overcome these limitations. The contributions of this paper are summarized as follows.

- We propose a simple mechanism to handle TCP connection disruptions that occur in an anycast CDN, whereby clients detect the disruptions when they occur, and re-request the remaining portion of the file. While simple, we demonstrate that this mechanism works efficiently and thus dismiss the need for more complicated mechanisms. In particular, our mechanism results in no tangible download throughput reduction for all reasonable server switching frequency and network path characteristics.

- We recognize the ability of switching servers in the middle of a large file download as an important tool for server offloading. Particularly, the characteristics of these downloads (long duration and large resource requirements) impose many performance and security problems if these downloads are statically bound to servers as in existing CDNs.

- We discuss the security enhancements provided by our mechanism. For example, in addition to quickly making server resources available to active connections, rapid purging of dormant TCP state in our mechanism increases server resistance to DoS attacks with no tangible penalty. We have conducted live Internet experiments to demonstrate this point. Furthermore, server offloading capability offers an effective solution to a recently identified DoS threat in a commercial CDN streaming service [21].

- We verify that en-mass redirection of ongoing downloads that occurs in anycast CDNs employing our mechanism does not cause any global synchronization effects. We have demonstrated the lack of synchronization in a controlled lab experiment, which maximizes the potential for synchronization. Thus, we certainly do not expect this problem to occur in practice, where variable network delays introduce significant randomness into the behavior of different TCP sessions.

Anycast CDNs are based on the assumption that anycast-addressed packets are usually delivered to the nearest endpoint within the anycast group. Policy-based routing may distort this assumption. While policy routing applies equally to both anycast and unicast routing paths, a careful study of its effect on the benefits of the anycast CDNs remains a topic for future work.

With large file and progressive streaming downloads taking a growing part of Internet traffic [11], we believe our mechanism fills an important emerging gap in the CDN capabilities. Furthermore, the simplicity of our mechanism increases its chances of being employed in a real network.

6. REFERENCES

[1] Akamai Inc. http://www.akamai.com/html/perspectives/index.html.

[2] IP Dummynet. `http://info.iet.unipi.it/~luigi/ip_dummynet/`.

[3] Keynote Data Accuracy and Statistical Analysis for Performance Trending and Service Level Management. `http://www.keynote.com/docs/whitepapers/keynote_data_accuracy_for_WebPerformance.pdf`.

[4] TCP Manual. `http://linux.die.net/man/7/tcp`.

[5] The Apache Web Server. `http://httpd.apache.org/`.

[6] H. A. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. V. der Merwe. Anycast cdns revisited. In *Proceeding of WWW '08*, pages 277–286, New York, NY, USA, 2008. ACM.

[7] H. Ballani, P. Francis, and S. Ratnasamy. A Measurement-based Deployment Proposal for IP Anycast. In *Proc. ACM IMC*, Oct 2006.

[8] T. Hardie. Distributing Authoritative Name Servers via Shared Unicast Addresses. IETF RFC 3258, 2002.

[9] G. Huston. Faster. The ISP Column, June 2005.

[10] H. Jun, M. Sanders, M. H. Ammar, and E. W. Zegura. Binding clients to replicated servers: Initial and continuous binding. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, page 168, Washington, DC, USA, 2003. IEEE Computer Society.

[11] C. R. Kalmanek. Evolving nature of content delivery. `http://www.dcia.info/activities/p2pmsla2007/ATT.pdf`.

[12] R. P. Karrer and E. W. Knightly. Tcp-paris: a parallel download protocol for replicas. In *WCW '05: Proceedings of the 10th International Workshop on Web Content Caching and Distribution*, pages 15–25, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Keynote. `http://www.keynote.com/`.

[14] K.-H. Kim, Y. Zhu, R. Sivakumar, and H.-Y. Hsieh. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. *Wirel. Netw.*, 11(4):363–382, 2005.

[15] Linux TC. `http://lartc.org/`.

[16] Z. Mao, C. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers. In *USENIX Annual Technical Conference*, 2002.

[17] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan. On the Responsiveness of DNS-based Network Control. In *Proceedings of Internet Measurement Conference (IMC)*, October 2004.

[18] P. Rodriguez and E. W. Biersack. Dynamic parallel access to replicated content in the internet. *IEEE/ACM Trans. Netw.*, 10(4):455–465, 2002.

[19] A. Shieh, A. C. Myers, and E. G. Sirer. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 175–188, Berkeley, CA, USA, 2005. USENIX Association.

[20] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 19–19, Berkeley, CA, USA, 2001. USENIX Association.

[21] A.-J. Su and A. Kuzmanovic. Thinning akamai. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 29–42, New York, NY, USA, 2008. ACM.

[22] M. Szymaniak, G. Pierre, M. Simons-Nikolova, and M. van Steen. Enabling service adaptability with versatile anycast. *Concurrency and Computation: Practice and Experience*, 19(13):1837–1863, September 2007.

[23] P. Verkaik, D. Pei, T. Scholl, A. Shaikh, A. Snoeren, and J. Van der Merwe. Wresting Control from BGP: Scalable Fine-grained Route Control. In *2007 USENIX Annual Technical Conference*, June 2007.