Riptide: Jump-Starting Back-Office Connections in Cloud Systems

Marcel Flores* Amir R. Khakpour† Harkeerat Bedi†
*Electrical Engineering and Computer Science Department, Northwestern University, Evanston, IL, USA
†Verizon Digital Media Services, Los Angeles, CA, USA
marcel-flores@u.northwestern.edu, {amir.khakpour, harkeerat.bedi}@verizon.com

***Abstract*—Large-scale cloud networks are constantly driven by the need for improved performance in communication between datacenters. Indeed, such back-office communication makes up a large fraction of traffic in many cloud environments. This communication often occurs frequently, carrying control messages, coordination and load balancing information, and customer data. However, ensuring such inter-datacenter traffic is delivered efficiently requires optimizing connections over large physical distances, which is non-trivial. Worse still, many large cloud networks are subject to complex configuration and administrative restrictions, limiting the types of solutions that can be implemented. In this paper, we propose improving the efficiency of datacenter to datacenter communication by learning the congestion level of links in between. We then use this knowledge to inform new connections made between the relevant datacenters, allowing us to eliminate the overhead associated with traditional slow-start processes in new connections. We further present Riptide, a tool which implements this approach. We present the design and implementation details of Riptide, showing that it can be easily executed on modern Linux servers deployed in the real world. We further demonstrate that it successfully reduces total transfer times in a production global-scale content delivery network (CDN), providing up to a 30% decrease in tail latency. We further show that Riptide is simple to deploy and easy to maintain within a complex existing network.**

I. Introduction

When operating large-scale cloud networks, increasing performance and reducing transfer times is one of the primary goals. Indeed, in many systems, the primary performance cost is incurred when performing regular communication between datacenters. Despite this prevailing goal of minimizing communication time, many modern networks are hampered by complex deployments, application restrictions, and operational limitations. Such complexities limit the ability to implement many complex solutions and approaches, which would require thorough overhauls [1].

Indeed many modern cloud systems are built on points of presence (PoPs) that span the globe [2], [3], which make solutions that depend on central control and coordination untenable. Worse still, these deployments are subject to the realities of real life operations. For example, different locations may run different operating system versions and software, making the deployment of new protocols and systems difficult. These locations may also be under different branches of administrative control, significantly complicating upgrades, maintenance, and other operational tasks.

Cloud systems further present a set of technical challenges to decreasing the communication cost. Much of this PoP-to-PoP traffic consists of short messages, as we explore further in this study. Therefore the cost of opening new connections, and in particular performing traditional TCP slow-start, is significant for such flows. In modern operating system, the default window of 10 segments [4] results in any flows larger than 15KB requiring more than a single RTT to transfer. Ideally, regular communications between the same datacenters could avoid this overhead by reusing existing connections. Reuse could significantly reduce the overhead associated with opening new connections. However, maintaining open connections for long periods in a global scale system is a challenge in both resource management and fault tolerance.

We argue that a solution based on learning past performance is able to reduce such costs without falling prey to the above complexities. Since many of the links between datacenters are used regularly, they are a significant source of information about the capacity of their corresponding paths. Indeed the performance of each historical connection reflects the likely capacity of future links. Newly arriving flows in the network can take advantage of this information, scaling new connections to reflect this learned performance capacity. By doing so, they are able to avoid much of the start-up cost traditionally associated with establishing new connections.

In this paper we present Riptide, a simple, yet effective tool, which observes the current congestion window sizes of existing connections and sets the initial congestion window (initcwnd) size accordingly for recurring connections. Riptide uses an evidence based approach that allows flows opening to locations with likely known paths to begin at a learned value, rather than reverting to a default state. In this way, Riptide is able to strike a balance between aggression and caution, entering the network at a level the path is known to support. Riptide is further applicable to any TCP protocol that employs slow start, granting it significant generality. To the best of our knowledge, Riptide is the first system to exploit open connection information for improving the performance of short flows in a cloud environment.

While its simplicity and ease of implementation in user-space mean that Riptide can deployed anywhere, it is particularly well suited for cloud network environments. Often cloud systems comprise of multiple datacenters or points-of-presence, which frequently communicate with each other and generate non user-facing *back-office* traffic [5]. In such environments, the usefulness of Riptide is maximized, as any pair of data centers often have existing information on the status of the network in the form of open connections.

We have implemented and deployed Riptide on a global-scale Content Delivery Network (CDN). CDNs represent a significant source of internet traffic, with data suggesting that up to 70% of HTTP traffic passing through a Tier 1 ISP originated at CDN [6]. Riptide has further been operating in production for over a year. Using data collected from its deployment, we demonstrate that Riptide is able to increase the median window size between datacenters by 200%. Next, we explore the Riptide parameter space, demonstrating that the imposing limits on the maximum size of the initial congestion window results in the best performance. We further show that this increase results in improved flow completion times, as additional flows are able to complete in fewer round trip times. Finally, we show that Riptide successfully improves the performance of slower flows, improving the 50th and 75th percentiles by nearly 30%.

In this work, we make the following contributions: (i) We demonstrate there is room to improve network performance with a straightforward modification to the way new connections are created, which does not rely on complex coordination. (ii) We show that such a system can easily be implemented on a production network. (iii) We demonstrate the effectiveness of Riptide in a cloud environment.

The remainder of the paper is organized as follows: In Section II we present our motivation in focusing on initial congestion windows, we examine the configurations that lead to challenges in a cloud environment, and highlight some of the technical challenges presented by Linux and modern TCP stacks. Next, in Section III we present the design of Riptide itself, and in Section IV we present an evaluation in a global cloud network. We provide a discussion in Section V and conclude in Section VII.

II. Background And Motivation

In this section, we motivate the development of Riptide. In particular, we consider the nature of modern cloud systems and the components of these modern systems which stand to be improved upon. We consider the particular challenges presented by complex cloud systems with datacenters spanning multiple, potentially distant, physical locations. We further explore the potential gains, which can be found by increasing the initial congestion window. Finally, we examine some of the challenges presented by the current implementation of the networking stack in Linux.

A. Challenges in Cloud Systems

We now explore some of the challenges introduced directly by cloud systems and architectures. In particular, we consider why obvious solutions, such as simply using persistent connections, are not tenable in most cloud environments. For our settings, we consider a situation in which there is a potentially large number of PoPs, distributed over a large geographic area, with a large number of nodes in each PoP.

First, as each node opens a connection to a different node in each other PoP, the total amount of resources devoted to such connections increases. If each node maintains each of these connections indefinitely, this would approach an n^2 mesh between all nodes, consuming significant resources for many potentially unused, or only lightly used, links.

Second, application behavior may restrict the applicability of maintaining a meaningful number of persistent connections. Indeed, applications may require parallel communications between two nodes, requiring additional links. Application errors may further necessitate the periodic closing of connections, for example in unmanageable error cases. The need to close these connections would cause a node to entirely forget all its learned information about the state of a link, and the node would have to open a new connection, starting with the default congestion window parameters. Simple procedures that close all connections to a node (*e.g.,* rebooting to apply updates) lose not only local connection information, but eliminate all information about the node on remote machines.

Third, cloud platforms often need to perform load balancing between PoPs [2]. Such load balancing may require termination of existing persistent connections and the creation of additional connections to accommodate increased capacity. Such shifts would again nullify many of the obvious gains of persistent connections.

Finally, large scale cloud systems are often delicately configured with little room for additional complex systems to be layered on top of existing infrastructure. In these situations, solutions that require vast re-architecting are untenable and can not be reasonably deployed. Furthermore, approaches with high resource requirements may push nodes and datacenters over-capacity and are therefore unusable. This restricts the solution space to possibilities that use only low to moderate resources and do not require large scale changes to the design of networks or datacenters. Therefore, any solution must not require costly instrumentation, complex maintenance, such as regular kernel patches, or incur high maintenance costs.

Beyond the above challenges, application workloads that require regular communication between datacenters or other PoPs, form a significant portion of the modern cloud system traffic [5]. Unlike traditional user-facing services that are designed to handle a variety of networks and capacities, these workloads are characterized by regular interactions over well provisioned links. However, modern day network stacks, in particular the Linux implementation of TCP and its corresponding congestion control protocols, have been designed for general case applications, attempting to enforce fairness and general network principles globally [7].

B. The Cost of Slow Start

Here, we explore some of the costs associated with the above cloud workload patterns. In particular, challenges arise when new connections must be opened between these PoPs and a relatively small amount of data must be transferred. In such cases, the default TCP Cubic in Linux begins connections with a congestion window of 10 segments. With common 1500 byte packets, this means approximately only 15KB of data may be injected into the network in the first round trip time (RTT). As cloud PoPs may be geographically distant from each other, this small congestion window means that many transfers of relatively small files are unable to complete in a single RTT, adding potentially significant delay to their transfer. Figure 1 illustrates the described scenario. These increased flow times will only be exacerbated as files become larger to meet the needs of streaming video and new application demands, as

a greater number of files will require more than a single RTT. Indeed, as we see from Figure 2, which shows the file size distribution of a production CDN platform, a significant fraction of files, 54%, are too large to fit in the default window of 10 segments. The majority of such files can significantly benefit from larger initial congestion window sizes.

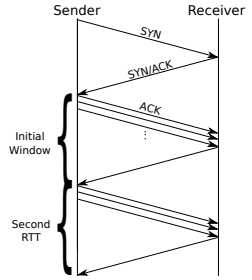


Fig. 1: Example of a file larger than the initial congestion window requiring an additional RTT to complete

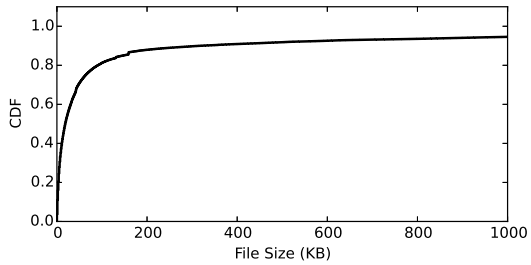


Fig. 2: Distribution of file size in a production CDN network

To understand the impact on a system with the above file distribution, we consider a model to show how using different initial congestion window sizes affects their file transfer performance. In particular, we consider the total number of RTTs consumed, as it directly reflects the performance cost to the system. For our model, we assume that the delay to put packets on the wire is negligible, and packets can be sent right away. We also assume that the receiver does not delay sending ACKs, and the connections experience no loss. We further ignore all potential flow control bottlenecks. All such events would increase the resulting delay.

Figure 3 shows the CDF for the number of RTTs needed to complete transfer of files with four different initial congestion window sizes using the size distribution shown in Figure 2. We see that an increase to an initial congestion window of 50 would allow significant gains, with over 31% more files able to complete in the first RTT. Further increasing the window to 100 would allow all but 15% of files to complete in the first RTT. In addition, there is a size interval in which such increases are able to effectively improve performance.

Figure 4 demonstrates the potential gain, percentage reduction in RTTs, by using higher initcwnds when compared to the default initcwnd of 10 for increasing file sizes. In this figure, we see that the primary improvements are seen between 15KB and 1000KB, after which the benefits of reducing a single RTT diminish. Very large files will necessarily require many round trips, and therefore cannot hope to reasonably fit entirely in

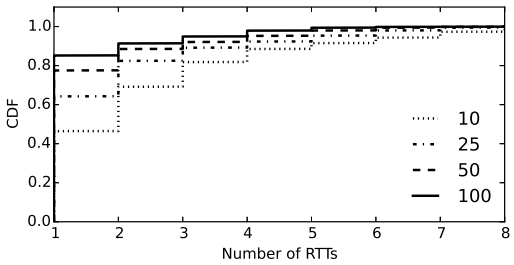


Fig. 3: Number of RTTs needed for transferring files of various sizes in a production CDN, assuming no loss or delay.

the initial window. However, as we see from Figure 2, these large files do not dominate the distribution.

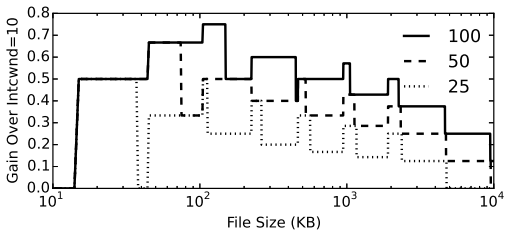


Fig. 4: Theoretical gain (reduction in RTTs) by using initcwnds of 25, 50, and 100 for various file sizes

In Figure 5, we present an example of the distribution of RTTs seen in such a globally deployed network. In particular, we see that in the median case we observe RTTs of over 125ms. We therefore expect that a large number of connections will traverse such routes.

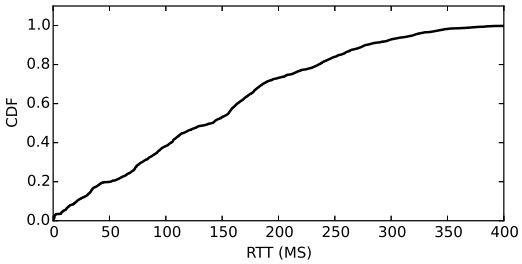


Fig. 5: RTT variation between globally deployed datacenters. 50% of links have an RTT $>$ 125 ms

Figure 6 shows the resulting change in file transfer times when we apply our transfer-time model to the observed RTTs for four different initcwnds of 10, 25, 50, and 100. We see that the small initial congestion windows incurs a significant penalty. In the median case, the transfer time is over 280ms longer than the initial congestion window of 100 case, while at the 90th percentile, we see the total transfer time increase by 290ms, about 100%.

While cloud system datacenters are likely connected through reliable and well provisioned links, granting additional flexibility, they are still subject to the usual challenges of

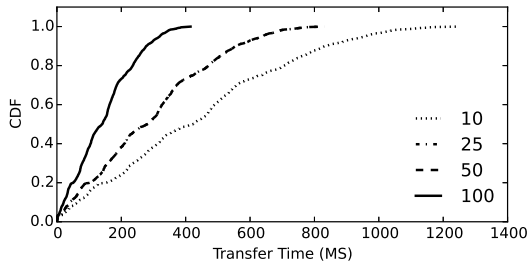


Fig. 6: Total transfer time for a 100 KB file over different initcwnds using our model and RTT distribution

Internet communication. These links must therefore be prepared to deal with the eventualities of congestion and loss. As such, direct aggressive schemes, such as simply increasing the value of the congestion window to a higher value [4], [8], increase the risk of inducing congestion when many new connections are opened. Therefore, any scheme seeking to improve performance must take into consideration that, at certain times, a more conservative approach to selecting an initial window is required.

III. Riptide Design

In this section, we present an overview of our design goals of our tool, Riptide, and a detailed consideration of the algorithm that Riptide employs. Next, we explore variations on the approach Riptide uses to store historical data. Finally, we present an analysis of the result of varying the granularity at which Riptide functions.

A. Design Objectives

Riptide is framed around the following principles: *(i)* Simplicity: The implementation of Riptide must be as a small agent which is easy to deploy, control, and modify in the context of a larger system. *(ii)* Distributability: The system must not require a central controller and must be able to make decisions promptly and independently. *(iii)* Adaptability: The system must adjust to changing network conditions and avoid adding additional strain to the network during busy periods.

In line with these objectives, the current implementation of Riptide runs in user space and takes advantage of a small number of standard Linux kernel interfacing utilities. As such, it is quite simple to operate and maintain, and does not require explicit kernel maintenance or devoted development resources.

B. Riptide Algorithm

Algorithm 1 presents how Riptide calculates the initial congestion window sizes (Init_CWND) using the a set of tunable parameters listed in Table I.

When Riptide is launched, it begins by polling the congestion window of all open connections via the `ss` utility in Linux every i_u seconds. It stores these values in its *observed table*. Next, for each destination that Riptide observed a connection to, it computes the *average* congestion window over the observed values.

Algorithm 1: Riptide's Init_CWND Calculation

```
while Running do
    observed table ← Current CWND for all connections;
    grouped windows ← Connections from observed table grouped by destination;
    for group in grouped windows do
        average ← average of all current windows;
        final ← moving average with history;
        Init_CWND to destination ← final;
    wait for i_u seconds
```

Parameter	Use
α	Weight to apply to newest data.
i_u	Update interval to poll current windows.
t	Time to live value of a stored window.
c_{max}	Maximum allowed window.
c_{min}	Minimum allowed window.

TABLE I: Riptide input parmeters

Figure 7 shows a simple example where a congestion window of size 80 is set based on the average of the existing observed windows.

Fig. 7: The core principle of Riptide is to set new initcwnds according to observed behavior.

Next, Riptide computes an exponentially weighted moving average using the previous average value, assigning α weight to the historical value, and $1 - \alpha$ to the newly seen value. Performing such historic weighting prevents the congestion window from enacting dangerous increases, and likewise prevents the window from plummeting in the case that all connections to a destination close or reset. The result is then bound by to be greater than the minimum c_{min} and less than the maximum c_{max}. This final value is stored as a list of final window values for each destination. Riptide then sets a route (using the Linux `ip` tool, as shown in Figure 8) to each host with the corresponding initial congestion window. Therefore, new connections to that destination will begin with the corresponding window, which was chosen by Riptide to reflect current network conditions.

```
ip route add 10.0.0.127 dev eth0 proto \\
    static initcwnd 80 via 10.0.0.1
```

Fig. 8: An example of the command used by Riptide

Final values are further stored with a time-to-live value t that each time a new value is computed. If the time-to-live expires, the entry is removed from the table, and the

corresponding route is removed, restoring the default initial congestion window of 10. This can occur if there are no active connections to a particular destination. In such a situation, Riptide has no information on the state of the connection to that destination, and therefore reverts to a more conservative choice of the default window. In our implementation, we take this value to be 90 seconds.

Since the entirety of Riptide is contained in a single Python script, it can be tuned and maintained with minimal cost. By further designing it as a stand-alone process, rather than a kernel module or modification, we avoid the need to alter it significantly in the face of updates. Given Riptide's direct and not-overly complex approach, such a straightforward deployment is necessary.

We emphasize that by setting only the *initial* congestion window, Riptide only alters the starting-point of connections. Once a connection begins transmitting data normally, the behavior of the congestion window is handled by the congestion control algorithm (for example, via TCP Cubic, which is the default in most Linux distributions). Riptide's chief alteration is therefore in starting connections at a suitable value. This approach allows the congestion control algorithm to safely react to changes in network conditions as necessary, without a need for Riptide itself to adjust. Importantly, however, if the set of a connections to a destination do demonstrate smaller windows, Riptide will respond accordingly, shrinking the initial windows.

Combination Algorithm: In the currently implemented scheme, Riptide employs two averages in order to combine the current observations with the existing data about the state of a link. The choice of using averages was made to best suit our particular deployment, but are fundamentally generic. For example, a more aggressive system might use the maximum congestion window observed on a path, rather than just the average, as the maximum represents the most the link is capable of handling. On the other hand a more conservative system might instead weight the value of an observed window by the amount of traffic that has passed through the link, information which is also available via `ss`.

The use of history is also flexible. First, this can be adjusted by altering the value of the weight α. Applying less consideration to historical statistics may particularly be worthwhile in environments with rapidly changing workloads. Furthermore, an exponentially weighted moving average need not be used at all. The system could perform longer-view historical analysis, in order to most effectively utilize consistent links, or ignore history entirely, to more rapidly respond to changes in network conditions.

Destinations as Routes: The granularity on which Riptide operates is also flexible. Specifically, the destination described above may be as specific as a particular host, as Linux will allow the setting of a specific route (*i.e.* a /32 route), and therefore a specific initial congestion window to only a specific host. On the other hand, the grouping of destinations can be a much broader selection: initial congestion windows can apply to entire prefixes.

Consider the following example, where two PoPs, A and B, are communicating, each draws their addresses from a particular /30 prefix owned by the operator. Further assume that the interconnect within the data center is evenly distributed. Therefore, connections between machines in each datacenter are subject to similar constraints. That is, the network conditions for a host from A communicating with any host in B are approximately the same. Therefore, a host from A could reduce the overhead incurred by Riptide by considering all hosts in B as a single destination, and only require computing a single initial congestion window value.

C. Linux/TCP System Components

The current state of the art in Linux system presents an additional set of capabilities and challenges. In particular, the current TCP implementations are geared towards general purpose deployments, and attempts to maintain accepted fairness standards on the Internet. While important, many of these heuristics of good behavior do not apply in internal cloud system communications, and present barriers to effective network usage.

The first restriction lies with the TCP initial receive windows (initrwnd) on receiving hosts, which determine the number of packets a host can receive at once. While the receive window usually grows rapidly, outpacing the senders congestion window, it is necessary that the initial window be large enough to accept any leading burst from the sender. If a sender opens with large initial congestion window, the default receive window may not be able to handle the first incoming burst. To avoid this limitation, the initrwnd must be increased to accommodate the maximum initial congestion window, c_{max}. While such an increase necessarily increases memory consumption, past increases in the TCP congestion window in Linux have been accompanied by corresponding increases in receive windows [7].

The next challenge arises from how initial congestion windows are set in Linux. While there has been significant discussion on allowing the initial window to be set as a per-socket parameter [9], the decision has generally been made to refrain from allowing such an option to maintain standard behavior on the Internet. Therefore, it is impossible to set the initial congestion window programmatically with standard socket APIs. However, initial congestion windows may be set on a per-route basis. While this is technically possible by using the Linux Netlink interface, it is intended to be done through the `ip` command-line utility[1]. However, even if such a kernel mechanism could be implemented, deploying it widely could prove challenging, as global kernel updates are not always tenable for compatibility and software management reasons.

In general, setting routes comes with greater overhead then setting only a congestion window. In the case of Riptide, we do not aim to alter the configuration of the route beyond the change to the initial congestion window, and therefore must set a route, which otherwise reflects identical settings to the default route. This however introduces potential for errors, as misconfiguration may cause machines to become inaccessible or misroute traffic.

On one hand, operating on a route basis presents Riptide with some restriction, namely it is no longer able to control specific connections. But at best is only able to determine the

[1]Since Linux kernel version 3.2.0-55.85

size of the initial window to an entire host. Therefore, *all* new connections to the specified host will begin with the specified value. On the other hand, this allows Riptide to adjust its granularity, since the `ip` tool can set routes with any given mask size, routes can be set for entire subnets, instead of a single host, as discussed above.

IV. Evaluation In A Cloud System

Next, we consider the benefits of a deployment of Riptide in a production CDN. In particular, we explore the ability of Riptide to increase the average congestion window size and decrease total flow completion time for appropriately sized files, improving performance. In the following section, we provide an overview of a global scale cloud system that deploys the Riptide system for communication between datacenters. We demonstrate the increase in congestion window sizes over the course of connection lifetimes as a result of Riptide. We show that this increase further results in a decrease in the total time consumed by network transfers. However, it is notable that these increases do not come at the cost of greater loss or instability in the network.

A. Infrastructure Overview

Riptide is deployed on a production CDN consisting of 34 well provisioned and globally distributed PoPs. Figure 9 shows the approximate location of the PoPs running Riptide, and Table II shows a continent level summary of these locations. As PoPs are located across the globe, Riptide is faced with links that encounter both low to high latency due to their physical distance and network conditions. Moreover, the application of this platform requires that any PoP may need to transfer data to any other depending on clients' traffic profile. Since Riptide only changes the behavior on the sender side, we are therefore able to consider the performance from each PoP to many others of varying distances. In other words, no single PoP only encounters others of a certain distance.

In order to measure the improvement in performance, we consider measurements from regular diagnostic probes sent across the system. In particular, every hour, each machine in each PoP requests a small probe object from every other PoP. If there is an existing and idle connection between the particular node and the remote PoP, the connection is reused, otherwise a new connection is made. We use three versions of probes of sizes 10, 50 and 100KB, simultaneously.

This probing infrastructure therefore allows us to observe 1) the current state of existing connections and 2) the performance for a freshly opened connection that has been adjusted by Riptide. Notably, the 50 and 100KB probes are too large to fit in the Linux default initial congestion window of 10. Finally, we consider a sampling interval, i_u of 1 second.

B. Performance

1) Congestion Windows: First, we seek to demonstrate that employing Riptide results in an overall increase in the observed congestion window sizes in the network, and that there is a reasonable maximum window size that Riptide should set. While the congestion window size is not a metric for actual performance, it represents a prerequisite for improved performance, and ensures that Riptide is operating as expected.

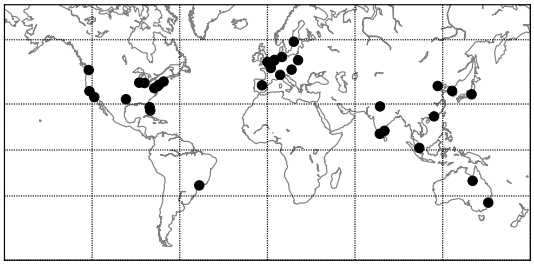


Fig. 9: Markers indicate PoPs with current Riptide deployment

Continent	PoP Count
Europe	10
North America	11
South America	1
Asia	9
Oceania	3

TABLE II: CDN PoPs with Riptide deployed

In order to measure the congestion window size, we sample the sizes of outgoing connections each minute using the `ss` tool. We further consider only connections that were created after Riptide was started, ensuring that all observed connections began with Riptide determining their initial window. To understand the effect of the maximum initial congestion window, c_{max}, we further considered Riptide under a wider range of conditions. In particular, we consider 50, 100, 150, 200, and 250 as values of c_{max}. In addition, we performed our measurements on a control group, where Riptide was not running, allowing us to understand the default congestion window distribution and to appropriately quantify the improvements caused by Riptide.

Figure 10 presents a CDF of the windows observed during a 12 hour period when Riptide was operating with various c_{max} values. We recall that Riptide only sets the *initial* congestion window. The later values are determined by traditional TCP dynamics. Despite this, we see a notable increase when running Riptide. In the median case, we see that the difference between the default and Riptide using the lowest c_{max} setting of 50 results in an increase of 100%. Furthermore, for the smallest 10% of connections, all c_{max} values achieved similar congestion window sizes. Such matching performance makes intuitive sense, as these connections likely encountered loss or other congestion events, and were therefore unable to take advantage of the larger window set via Riptide.

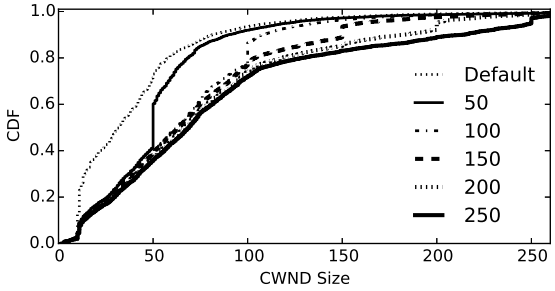


Fig. 10: Varying congestion windows across all datacenters for each c_{max} value. A knee appears at 100 suggesting diminishing returns beyond this value.

When examining the difference in c_{max} behaviors, each line features a mode at its corresponding value. These modes are the result of connections that were opened but not used again (or used for very small transfers), leaving them with their initial windows. This behavior is not a fundamental characteristic of Riptide, but is instead a result of the traffic to which Riptide is exposed. Since Riptide chooses values based on empirical information, it only begins setting windows at the c_{max} when it observes them in the wild. Traffic patterns that feature higher volumes of traffic may result in different behavior. This need to achieve natural growth (*i.e.,* Riptide never "hops" ahead of observations) results in the strong linear trend seen in lower 80% of connections for c_{max} value of 100 and above. We therefore use a c_{max} value of 100, as it offers the majority of potential gains, while limiting the risk of overloading the network in the event of many new connections.

Figure 11 demonstrates the effect of traffic patterns on the congestion windows set by Riptide. Here, we consider two individual PoPs. One subject only to probe traffic, while another is among the busiest in the network. We see that the PoP with organic traffic sees much higher windows, achieving a congestion window of 100 for over 44% of connections. On the other hand, the probe-only traffic is below a window of 100 in 99% of cases, and has a median window of 75 segments. This clearly indicates the impact of the PoPs organic traffic profile for determining the initial congestion windows.

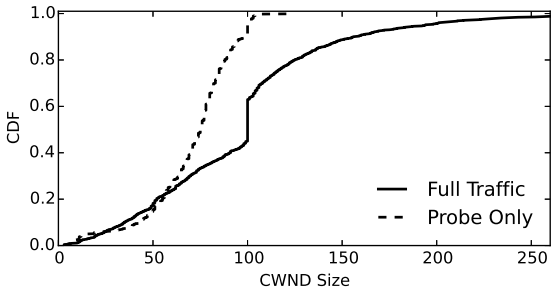


Fig. 11: Comparison between the observed congestion windows for Riptide at two datacenters: one running only probe traffic, and one operating on full traffic

2) Transfer Times: Next, we demonstrate that implementing Riptide resulted in an improvement in the transfer times of our measurement probes. In particular, we expect when running Riptide, the greater congestion window sizes will result in an increase in the fraction of probes that are successfully able to fit in the initial congestion window. Such an increase should result in an improvement in their flow completion time.

Here, we consider the performance of our probing infrastructure. In particular, we consider the performance from 2 PoPs, one located in North America and the other in Europe when sending data to a subset of other locations. We consider 4 groups, (1) close destinations (less than 50ms RTT), (2) medium destinations (<100ms), (3) far destinations (<150ms) and (4) very far destinations (>150ms), all relative to the sending PoP. We consider the performance of our 10, 50, and 100KB probes over a 20 hour period. As with the previous experiment, we only consider connections that were opened after Riptide began.

Figures 12, 13, and 14 present sets of CDFs of the probe transfer times from a single PoP, grouped by the RTT to the destination for the 10, 50, and 100KB probes, respectively. The first feature to note in Figure 12 is that Riptide had no discernible effect on the 10KB probes. This behavior is expected, since files of such small size already fit inside the default initial congestion window, and therefore such flows do not benefit from running riptide, regardless of RTT. We do however observe that riptide caused no negative side-effects for such flows. Their performance matched the default case.

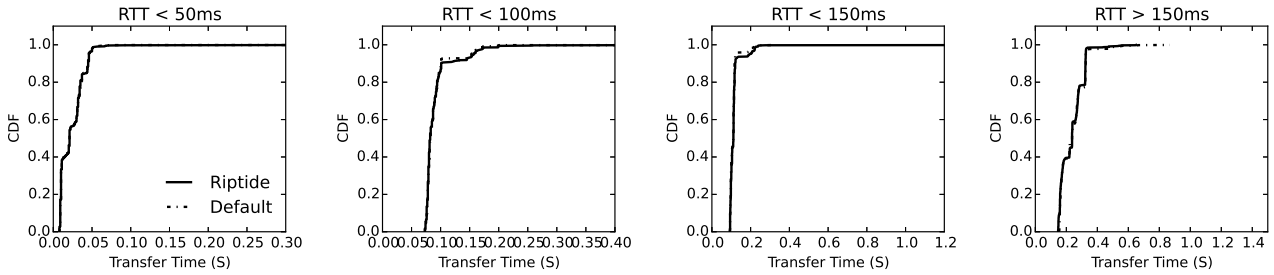
In Figure 13, we begin to see the completion times from the Riptide enabled flows begin to differentiate themselves, with transfer times decreased for 30% of connections. And finally, in Figure 14, we see that the 100KB probes were able to achieve gains across 78% of the observed connections. Indeed, this is because the larger window enabled the largest probes to complete faster. In particular, we see a stair-stepping behavior (especially in Figure 13 (b) and Figure 14 (b) and (c)). Such patterns are the result of the flows completing an entire RTT faster than the default case. As destinations move further away, the gap between Riptide and non-Riptide probes increases, with Riptide flows regularly completing an RTT sooner.

The performance with Riptide appears to be no-worse than the default case. Conventional wisdom dictates that the increase of window size results in an increase of loss due to the higher load on the network. However, if this were the usual case, we would see an increase in completion times for a large fraction of flows. In the figure, we see that the tail performance is very similar, with and without Riptide, suggesting that Riptide does not lead to dangerous congestion in the network. This is likely a result of Riptide's adaptive design: since it reacts to observed conditions, for example losses, it is unlikely to exceed the overall capacity of the link.

C. Performance by Percentile

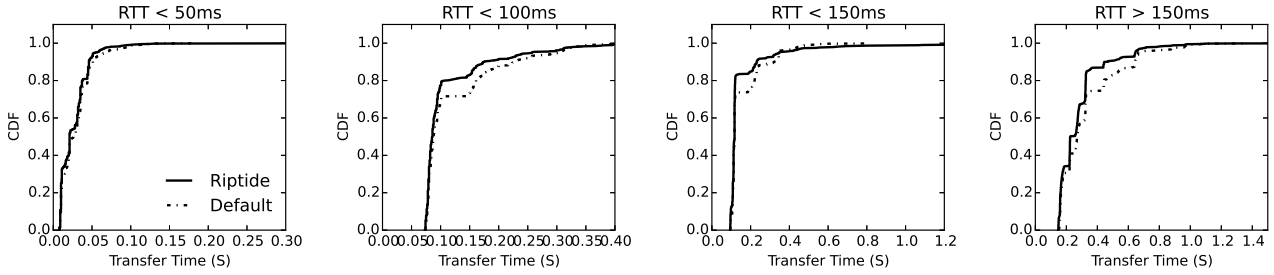
Next, we seek a deeper understanding of the impact Riptide has on the network performance by considering the change in completion time broken down by percentile and averaged across destinations. Such analysis provide us with insight into which flows Riptide impacts the most. We expect that for the slowest probes their performance was due to adverse network conditions, and therefore we don't expect any particular change when running Riptide. We again consider outgoing probes from the two PoPs described in the previous section, as they provide a sufficient volume and variety of traffic.

Figure 15 shows the changes in performance by percentile from the (a) European and (b) North American PoPs in 5% steps for the 50KB probes. In this case, we see that majority of gains were achieved by the worse performing flows. The 5th through 60th percentiles (50th for the North American PoP) saw almost no change in performance, but higher percentiles saw notable improvements, 30% and 21%, respectively. Such differences are in line with expectations, as the best performing 50K probes likely completed in a minimum number of RTTs previously, while worse performing percentiles took a greater number of RTTs. Riptide was successfully able to reduce the number of round trips, improving performance. We do observe that the 95th percentile observed lesser gains in the North American case, and suspect this is the result of flat performance in the longest case.



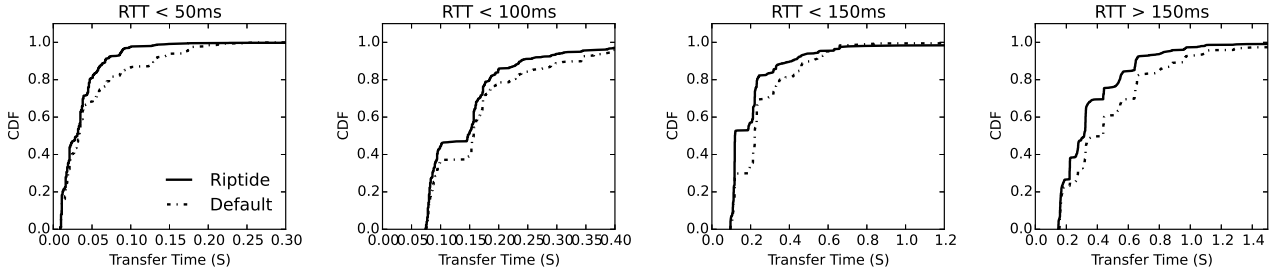
(a) (b) (c) (d)

Fig. 12: CDF of probe completion time for 10K probes, grouped by RTT: (a) is < 50 ms, (b) between 51 and 100 ms, (c) between 101 and 150 ms, and (d) > 150 ms for the 10K probes



(a) (b) (c) (d)

Fig. 13: CDF of probe completion time for 50K probes



(a) (b) (c) (d)

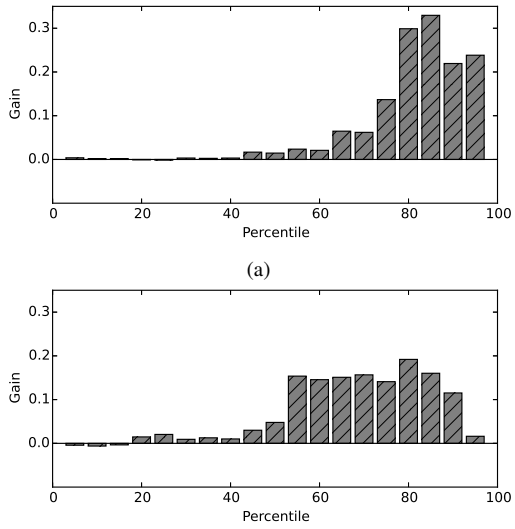
Fig. 14: CDF of probe completion time for 100K probes

Figure 16 shows analogous data for the 100K probes. Here, we see greater improvements across the percentiles. The European PoP sees improvement at the 30th percentile and above, while the North American PoP sees gains in all cases. This broader increase is the result of more flows being able to take advantage of the increased window size, improving not only the performance in the tail, but also for the better performers. We further note that the total gains were higher in the North American location, with gains up to 25%.

D. Edge Cases

Finally, we explored both the minimum and maximum performance for the 100KB probes. In the minimum case, we see essentially no change, as expected. In the European PoP case, 75% of destination PoPs showed no change, and the remaining 25% saw changes within $\pm 5\%$. Results were similar for the North American case, with 50% of destinations showing no change and the remainder being within $\pm 5\%$. In the best case, probes were already completing within the minimum number of RTTs possible and increasing the congestion window provides no gains.

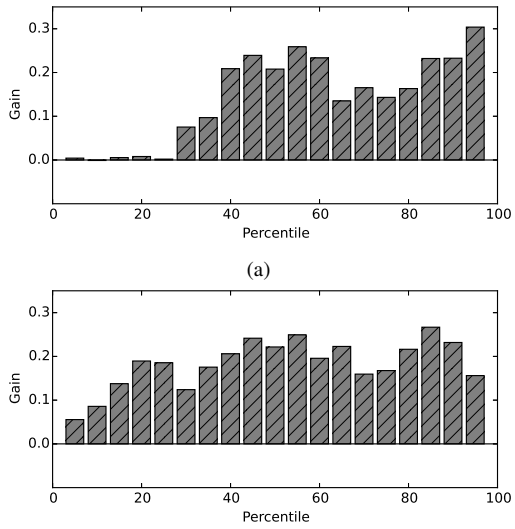
Next, we consider the performance in the maximum case, *i.e.,* the worst-case performance. For the European PoP, 50% of the destinations saw changes within 6%. The remaining destinations were not consistent, with some seeing significant positive improvements, and others seeing comparable decreases. The North American PoP was subject to higher variance, but featured no discernible trend. We suspect this



(a)

(b)

Fig. 15: Fraction of gain by percentile for the (a) European and (b) North American datacenter for 50KB probes



(a)

(b)

Fig. 16: Fraction of gain by percentile for the (a) European and (b) North American datacenter for 100KB probes

behavior is due to the relative unpredictability of worst case behavior in networks. Losses and other failures can cause signficant decreases in performance, but did not seem to occur more frequently with Riptide. As expected, Riptide is most effective in improving performance in approximately 50th to 95th percentiles, and seems to have little effect on edge cases.

V. Discussion

Effectiveness: As noted before, Riptide sets the initial congestion window values based on what it learns from similar connections. Hence, its gain and effectiveness is tightly correlated with not only the traffic pattern and size of the objects sent through the connections, but also the quality of the links and their congestion level along the path to the destination. If a connection tranfers large objects and links are not congested, it is more likely that Riptide will set higher window sizes for similar connections, and hence prove more effective. Conversly, as we learned from Figure 11, if a server is idle or the links that connects it to a given destination is congested, Riptide effectiveness would be minimal.

Overhead: Since Riptide runs in an entirely distributed fashion, it is able to avoid the significant complexity of coordination traffic, and all its overhead is restricted to computation on the single node. While such coordination may offer further benefits, we believe the current simplicity offers greater opportunity for real-world deployment. Furthermore, these calculations are scheduled, *i.e.* every i_u seconds, avoiding the need for processing data as it arrives.

Additional Algorithms: Riptide is a mechanism by which initial congestion windows can easily be controlled based on observations of current network performance. While our implementation and current design are straightforward, setting congestion windows in-line with those observed, leaving actual window computations to TCP, it could easily be extended to incorporate additional information into its computations. For instance, if a cloud system were able to provide it with higher level information (*e.g.,* the need to perform immediate load balancing), it could be used to set more conservative congestion windows to avoid sudden crowding. Alternatively, more complex algorithms could be used in attempt to foresee network level variations. For example, a significant decrease in congestion window over a short time may indicate the need to aggressively decrease the initial windows, beyond what is happening to existing connections. While Riptide is designed to generically incorporate additional algorithms, our study in Section IV found that simple per-destination averages resulted in faster completion times for many of the observed flows and offered a balance between complexity and performance gains.

Kernel Implementation: Riptide could further be implemented directly in the Linux kernel. Such an implementation would likely reduce load, as an external program no longer has to monitor all open connections, and potentially enable higher granularity computations. It could further allow setting of initial congestion windows on a per connection basis, rather than per route, as can be done from user space. While such an implementation may prove useful eventually, Riptide's current user space design allows significant flexibility, while avoiding many of the costs associated with maintaining a potentially significant kernel patch.

As discussed in Section III-C, portions of Riptide's design could be improved with additional interfaces for controlling the congestion window. Such changes would represent a middle ground, offering Riptide tighter control without necessarily introducing the above maintenance overhead. We emphasize, however, that such adjustments are not necessary for Riptide to be effective.

VI. Related Work

Reducing transfer time on Internet traffic has long been a goal in network research. These optimizations have come in the form of application specific adjustments, which hope to more effectively use the total network [10], [11], [12], [13], [14]. Generally, these methods attempt to reduce the total number

of round trips. In general, such improvements are orthogonal to Riptide, and many could be used alongside it.

Other systems have operated further down the stack, seeking to improve performance at the transport layer. For example, by reducing the penalty associated with connection handshakes, [15], [16], or the time spent handling time outs [17], [18]. Riptide is similar in goal to many of these works, as it aims to eliminate the cost associated with regular opening and closing of connections, but does so in a generic fashion that allows it to operate without changes to the protocols themselves, but instead utilizes available TCP features, *i.e.* setting the initial congestion window. Riptide could be used alongside many of these systems, providing greater performance.

Others have proposed increasing the aggression of TCP [4], [17], [19] Indeed, the default TCP congestion window has already been increased to 10 segments. Others have proposed eliminating portions of TCP, as the Internet's stability does not rely on them [20]. Riptide avoids the risks of increased aggression by observing the current state of the network and adjusting any changes it makes to reflect the network behavior. Riptide is further intended to operate on datacenter-to-datacenter connections and is therefore likely to be operating on well provisioned links.

Another set of works has explored the complexity of data center network behavior. Much of this work has focused on data within a datacenter, which often includes its interactions with the outside world [21], [22]. Other work has specifically explored the interactions between datacenters, for example the traffic patterns seen between datacenters [3], and the character of back-office traffic visible from the web [5]. The complexity of large cloud networks and its impact on latency has further been considered [23], [24]. Others have examined the relationship between back-office traffic and user-perceived latency [25], [26]. Riptide is motivated by many of these insights and attempts to improve datacenter-to-datacenter communication in order to improve total system performance.

VII. Conclusions

In this paper, we demonstrated the design and implementation of Riptide, a tool for determining optimal initial TCP congestion windows based on the observed behavior of existing flows. It is noteworthy that Riptide is designed to be simple, easy to maintain, user space application that can be deployed within complex existing infrastructure, with minor considerations. We further demonstrated the effectiveness of Riptide in a globally deployed CDN. We showed that by implementing Riptide, we were able to achieve a 200% increase in the size of the live congestion windows in production. We also demonstrated that this change results in a decrease of flow completion time for our probes. Although, we learned that the effectiveness of Riptide depends on the production traffic profile, in most cases Riptide considerably improved the latency of object delivery, which is critical to the performance of cloud systems.

Acknowledgment

Special thanks to Rob Peters, Derek Shiell, the HTTP-Dev team, and others at Verizon Digital Media Services for all their feedback and implementation assistance.

References

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *SIGCOMM CCR*, Dec. 2008.

[2] F. Chen, R. K. Sitaraman, and M. Torres, "End-user mapping: Next generation request routing for content delivery," in *Proc. of Sigcomm*, 2015.

[3] Y. Chen, S. Jain, V. Adhikari, Z.-L. Zhang, and K. Xu, "A first look at inter-data center traffic characteristics via yahoo! datasets," in *Proc. of IEEE INFOCOM*, 2011.

[4] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," in *SIGCOMM CCR*, 2010.

[5] E. Pujol, P. Richter, B. Chandrasekaran, G. Smaragdakis, A. Feldmann, B. M. Maggs, and K.-C. Ng, "Back-office web traffic on the internet," in *Proc. of IMC*, 2014.

[6] I. Poese, B. Frank, G. Smaragdakis, S. Uhlig, A. Feldmann, and B. Maggs, "Enabling content-aware traffic engineering," *SIGCOMM CCR.*, Sep. 2012.

[7] J. Corbet, "Increasing the TCP initial congestion window," Feb. 2011.

[8] S. Kayan, "Initcwnd settings of major cdn providers," Aug. 2014. [Online]. Available: http://www.cdnplanet.com/blog/initcwnd-settings-major-cdn-providers/

[9] "Socket option to set congestion window," http://www.spinics.net/lists/netdev/msg131097.html.

[10] M. Rabinovich and H. Wang, "Dhttp: an efficient and cache-friendly transfer protocol for web traffic," in *Proc. of INFOCOM 2001*, 2001.

[11] B. Krishnamurthy, R. Liston, and M. Rabinovich, "Dew: Dns-enhanced web for faster content delivery," in *Proc. of WWW*, 2003.

[12] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystify page load performance with wprof," in *Proc. of USENIX NSDI*, 2013.

[13] ——, "How speedy is spdy?" in *Proc. of USENIX NSDI*, 2014.

[14] "mod_spdy," https://code.google.com/p/mod-speedy.

[15] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan, "TCP fast open," in *CoNEXT*, 2011.

[16] W. Zhou, Q. Li, M. Caesar, and P. B. Godfrey, "Asap: A low-latency transport layer," in *CoNEXT*, 2011.

[17] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: The virtue of gentle aggression," in *SIGCOMM*, 2013.

[18] M. Rajiullah, P. Hurtig, A. Brunstrom, A. Petlund, and M. Welzl, "An evaluation of tail loss recovery mechanisms for TCP," in *SIGCOMM CCR*, 2015.

[19] T. Leighton, "Improving performance on the internet," *Commun. ACM*, vol. 52, no. 2, Feb. 2009.

[20] A. Mondal and A. Kuzmanovic, "Removing exponential backoff from TCP," *SIGCOMM CCR*, vol. 38, no. 5, Sep. 2008.

[21] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc of ACM IMC*, 2009.

[22] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. of ACM IMC*, 2010.

[23] C. Huang, A. Wang, J. Li, and K. W. Ross, "Measuring and evaluating large-scale cdns," in *Proc. of ACM IMC*, 2008.

[24] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan, "Mapping the expansion of google's serving infrastructure," in *Proc. of ACM IMC*, 2013.

[25] Y. Chen, S. Jain, V. K. Adhikari, and Z.-L. Zhang, "Characterizing roles of front-end servers in end-to-end performance of dynamic content distribution," in *Proc. of ACM IMC*, 2011.

[26] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang, "A provider-side view of web search response time," in *Proc. of SIGCOMM*, 2013.