Multipath TCP under Massive Packet Reordering

Nathan Farrington

UC San Diego
`farrington@cs.ucsd.edu`

Abstract

Conventional wisdom states that TCP performs poorly when packets arrive out of order. Reordered packets interact poorly with TCP's fast retransmit mechanism. For this reason, multipath routing is rarely used in the Internet, although it is supported by routers and routing protocols. Data center networks are very different from the Internet, having regular topologies, high-capacity-low-latency links, and switches with very small packet buffers. Per-packed load balancing would be extremely efficient in data center networks, eliminating the need to schedule the paths of individual TCP flows and the need to maintain per-flow state in each switch. In this paper, we survey techniques for making TCP more robust to packet reordering, in the hope of making per-packet load balancing a reality.

1 Introduction

Data center networks are LANs that interconnect hundreds to thousands of compute and storage nodes. They are typically Ethernet networks, using gigabit links between nodes and switches, and 10 gigabit links between pairs of switches. On top of Ethernet, the TCP/IP protocols are commonly used, perhaps because data centers often host web applications and web services. Recently, a new class of data center applications has emerged; MapReduce [13], and the associated open-source project Hadoop, are programming systems that enable large-scale computations across thousands of nodes. MapReduce works best for embarrassingly parallel problems that do not require nodes to exchange intermediate computations on small timescales, but do require massive network capacity to exchange terabytes to petabytes of data over the course of hours.

Current data center network fabrics cannot provide the capacity to satisfy the demand of these large MapReduce installations. Typically, data center networks are structured as multi-rooted trees for capacity scaling, load balancing, and fault tolerance [1]. The roots of these trees are large, proprietary, modular packet switches, often with a cost of over \$10,000 per port for 10 gigabit Ethernet. Besides the restrictive price, these packet switches are also difficult to scale beyond their current size. They use single-stage crossbar architectures that scale with complexity $O(n^2)$, where n is the number of ports.

This demand for network capacity has generated interest in redesigning data center network fabrics. The HPC community has successfully used the fat-tree [26] network topology for massive bandwidth scaling, and it has been proposed to do the same for data center Ethernet networks [1]. Figure 1 shows an example fat tree network; the networks proposed for data centers are much larger than this. Redesigning data center networks as fat trees raises many research questions, such as how to cope with the cabling complexity, how to treat the network as a single Layer 2 domain while still maintaining path diversity, and how to schedule TCP flows through the network to avoid congestion. It is this last question that is related to the topic of this paper[1].

With a fat-tree network topology, there are multiple equal-length paths between any pair of nodes, and packets could take any of these paths. Although IP is a best-effort protocol and does not guarantee in-order packet delivery, every switch and router architecture goes to great pains to enforce in-order delivery internally. This could be seen as the sacred cow of switch design: "never ever, ever reorder packets." Why is this? The primary reason is that TCP performs poorly when packets are received out of order. Even the reordering of a single packet once per second is enough to significantly degrade the throughput of a TCP flow in the wide area. In some

[1] In a recent research exam, Al-Fares surveyed on the area oflocal—network load balancing while maintaining packet ordering. In contrast, we argue that load balancing is trivial if packet ordering is not maintained.

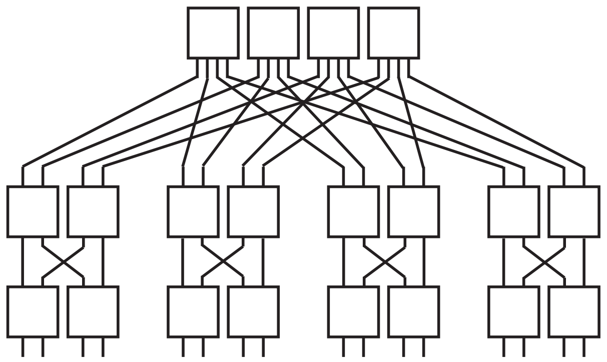

Figure 1: An example fat-tree network.

sense TCP violates the end-to-end principle [40] by assuming in-order delivery, when IP makes no such claim. The full story is given in section 2.

What if TCP was robust against reordered packets? We argue that this would greatly simplify network design. The alternative is to schedule and constrain each TCP flow to a single path through the network so that packets do not arrive out of order. TCP flow scheduling has many challenges. First, switches are required to maintain state proportional to the number of TCP flows, which is $O(n^2)$ where n is the number of hosts. Second, a logically centralized TCP flow scheduler is required that will periodically poll the switches for flow information. The delay in this control loop will lead to inefficiencies and overcompensation for some traffic patterns. There are stateless hash-based techniques, but they do not scale to networks as large as the ones we propose [18, 43].

Although TCP flow scheduling is the subject of ongoing research, we propose to use packet scheduling instead. With packet scheduling, each switch independently decides on the upward path of each packet, regardless of the final destination. Switches can choose any upward port, since just as all roads lead to Rome, all upward ports lead a root switch, which will then lead to the final destination. We expect simple policies such as least loaded port, least-used port, and uniform random, to provide good performance without the need of a central flow scheduler or maintaining any per-flow state in the switches.

We propose a radical shift in thinking in which we gladly accept massive amounts of packet reordering in exchange for a much simpler, less costly, and higher-performing network architecture. To our knowledge, no one has attempted to use TCP under such extreme conditions as we propose, in which packet reordering will be the common case. Requiring that routers and packet processors maintain packet ordering means that routers must be non-work-conserving, which means that they are less efficient than work-conserving routers and packet processors that are allowed to reorder packets. The ultimate goal of this work is to simplify network architecture to reduce the cost and power consumption of data center networks, while providing a very scalable technique for meeting future data center network capacity demands.

This paper surveys the area of TCP performance under packet reordering. Section 2 provides more background on why packet reordering leads to poor TCP performance. Section 3 presents the results of several Internet measurement studies that quantify how prevalent packet reordering is in real networks. Sections 4 and 5 describe the various solutions proposed in the literature, and the systems which implement those solutions, respectively. Finally, section 7 proposes experiments that could be conducted to shed light on packet reordering inside data center networks.

1.1 Glossary of Terms

ACK A TCP acknowledgement. A TCP receiver transmits an ACK to the TCP sender after the receipt of a data segment. In TCP, ACKs are positive acknowledgements, and the absence of an ACK is used by the sender as an indication of packet loss (either the data segment or the ACK) [39].

AIMD Additive Increase Multiplicative Decrease. A strategy used in the congestion avoidance state where a sender increases the number of packets in flight by one after every RTT, but reduces the number of packets in flight by a factor of two after packet loss [23].

Congestion Avoidance The steady state in the TCP state machine. During congestion avoidance, the sender uses AIMD to probe for spare network capacity, while using packet loss as an indication of congestion to quickly reduce the offered load [2].

Congestion Collapse A state where all router buffers in the network have filled and most transmitted packets are eventually dropped before reaching their destination [33].

Congestion Window (cwnd) A TCP state variable. The TCP sender may not have more segments in flight than size of the congestion window. The slow start, congestion avoidance, and fast recovery states all update the congestion window when events occur, such as the receipt of an ACK, a DUPACK, or a timeout [2].

Cumulative ACK An ACK will acknowledge all sent data up to the sequence number in the ACK. If a previously transmitted ACK is lost, then the next transmitted ACK will acknowledge the data that was also acknowledged in the lost ACK. This increases robustness [39].

Delayed ACK A TCP receiver may choose not to transmit an ACK after each received data segment, but may choose to wait up to 200 ms or after the receipt of two data segments, whichever comes first. This reduces network congestion but makes TCP less responsive [10].

DSACK Duplicate selective acknowledgement. An extension of SACK; DSACK allows a receiver to explicitly notify a sender when a duplicate segment was received. Duplicate segments are most often received due to spurious retransmissions. DSACK allows a sender to unambiguously detect spurious retransmissions, except for the very rare case of packet duplication in the network [30].

DUPACK Duplicate acknowledgement. When a receiver receives a data segment that is later than the next expected segment, then the receiver retransmits the last sent acknowledgement, hence a duplicate acknowledgement. The sender can use the receipt of duplicate acknowledgements as an early indication of packet loss [39].

Duplicate Threshold (dupthresh) TCP Reno hardcodes the number of DUPACKs required to trigger fast retransmit as 3. Some TCPs change this constant number to a variable, named dupthresh, and use this variable as an estimator of packet reordering in the network.

Fast Recovery A state in the TCP state machine. TCP enters the fast recovery state after a fast retransmit event, and then enters the congestion avoidance state after the receipt of a regular acknowledgement, or the slow start state after a timeout [2].

Fast Retransmit When a TCP sender receives a sequence of three identical duplicate acknowledgements, it assumes that a segment was lost and immediately retransmits the lost segment. The sender also reduces the slow start threshold and congestion window [2].

Frame An Ethernet frame. We use the terms frame, packet, and segment interchangeably.

IP Internet Protocol [37].

Limited Transmit An extension to fast retransmit that allows a sender to transmit data segments after the receipt of the first two (of possibly three) identical duplicate acknowledgements. These extra packets are useful for TCPs that use dupthresh, since more packets increase the chance of exceeding dupthresh when a packet loss has occurred. Without limited transmit, a TCP that uses dupthresh would be more likely to timeout and enter slow start [16].

Packet An IP packet. We use the terms frame, packet, and segment interchangeably.

RTO Retransmission Timeout. A TCP sender's RTO timer is based on an estimate of the RTT of connection [39].

RTT The round-trip time of a TCP connection; the amount of time it takes for a sender to transmit a data segment to the receiver and for the receiver to transmit an ACK to the sender [39].

Segment TCP provides the application with the abstraction of an in-order byte stream. TCP *segments* this byte stream into contiguous arrays of varying lengths, subject to a maximum segment size (MSS) [39]. We use the terms frame, packet, and segment interchangeably.

Sequence Number Each TCP segment carries a sequence number, with segments sent later having larger sequence numbers. This allows a receiver to acknowledge individual segments by reporting back the sequence numbers [39].

Slow Start The initial state in the TCP state machine when a new connection is established. During slow start, the sender's offered load starts at one packet, and increases exponentially in proportion to the connection's RTT. This allows an efficient way to estimate the capacity of the path. Once a packet is lost, the TCP state machine transitions into the congestion avoidance state [2].

Slow Start Threshold (ssthresh) A TCP sender state variable used to determine whether to enter the slow start state or the congestion avoidance state [2].

SACK Selective acknowledgement. Allows a TCP receiver to notify the sender as to which data segments have been received. A sender can use this information to retransmit only data segments that have not yet been received [29].

Spurious Retransmission When a TCP sender retransmits a segment that was assumed to be lost, but was actually just delayed by the network [27].

TCP Transmission Control Protocol [39].

2 A Brief History of the TCP Packet Reordering Problem

The Transmission Control Protocol was first described in a 1974 paper by Cerf and Kahn [12]. It is surprising how much of the original design still exists in modern TCP/IP, such as IP packet fragmentation, TCP segmentation and reassembly, multiplexing and demultiplexing among multiple processes, segment retransmission, duplicate segment detection, sliding window flow control, and the three-way handshake for connection establishment. In the original design, both TCP and IP were merged into a single layer, and did not become separate layers until 1978. Although layering was understood during this time [14], it wasn't until the introduction of the OSI model [46] in 1980 that layering became a fundamental principle of network protocol design. TCP was finalized in 1981 [39]; we refer to this version as TCP Postel.

2.1 Congestion Control

Interestingly, TCP Postel says nothing about congestion control, only flow control. Flow control prevents a sender from transmitting more packets than a receiver can properly receive, without leading to buffer overflow and dropped packets in the receiver. In contrast, congestion control prevents all senders in aggregate from transmitting more packets than the Internet routers can properly forward, without leading to buffer overflow and dropped packets in the routers. In an ideal network, packets would never be dropped, since a dropped packet indicates a loss of time and energy for the sender and all routers along the path, as well as lowered throughput and increased latency between the sender and receiver.

It turns out that congestion control is a much more difficult problem to solve than flow control for several reasons. First, congestion control deals with all senders and all routers in the network, rather than just one sender and one receiver. Second, congestion control is decentralized, with no authority dictating the maximum transmission rate of any particular sender. Third, in game-theoretic terms, congestion control has an inefficient Nash equilibrium; all senders would like to transmit at the maximum rate that their receivers can accept, but if all senders did this, then the network would experience *congestion collapse* [33], where all oflocal— the routers' buffers fill to the point where most transmitted packets get dropped. When congestion collapse occurs, some packets will get through without being dropped, but throughput will plummet and latency will skyrocket. The only way to repair a congestion collapse is for the senders to lower their transmission rate to the point where the router queues will drain, but this is not the best strategy for any individual sender.

As the Internet experienced exponential growth, congestion quickly became the most pressing problem of TCP. Nagle's 1983 paper [33] coined the term "congestion collapse" and also presented a few techniques to help control it. Nagle's algorithm improved the efficiency of telnet-like applications by buffering several characters of data and transmitting them in a single packet, rather than transmitting a separate packet for each character. Nagle also experimented with the ICMP Source Quench Message [38] to allow routers to throttle end-host transmissions when the router's queues started to fill, and proposed several fair dropping policies for routers. Unfortunately, none of these modifications were sufficient to prevent congestion collapse, perhaps because the original behavior of TCP was to send *more* packets when loss occurs, not less. We refer to this version as TCP Nagle, which is TCP Postel with the addition of Nagle's algorithm.

In 1986, congestion collapse occurred in the Internet on several occasions. For example, in the path between Lawrence Berkeley Laboratory and UC Berkeley, throughput dropped from 32 Kb/s to 40 b/s. Jacobson, in his seminal paper [23], added robust congestion control to TCP which practically eliminated congestion collapse. Essentially, a new state machine was added with three states: slow start, congestion avoidance, and fast recovery. The existing TCP mechanisms were left unchanged. The congestion control state machine interacts with the rest of TCP via a single variable called the congestion window (cwnd). The new constraint is that a TCP sender may not have more packets in flight than can fit into the congestion window.

How does the TCP congestion control state machine work? A new TCP connection begins in the slow start state. The sender increases its transmission rate exponentially until a packet is dropped. This allows a sender to quickly determine the capacity of the network between itself and the receiver. The alternative would be to begin a transmission at the line rate of the link interface, which is often much larger than the network interface and would lead to a large degree of packet loss. The congestion avoidance state uses additive increase multiplicative decrease (AIMD) to slowly search for more bandwidth, while reducing the transmission rate exponentially when congestion occurs. Fast recovery is described in the next section.

Jacobson's modifications were so important and profound that they instantly became the de facto standard of TCP. The version of BSD UNIX in use at the time was 4.3 Tahoe and Jacobson's modifications, minus fast recovery, were added to this version. BSD 4.3 Reno contained all of these modifications plus fast recovery, and the name *TCP Reno* became synonymous with congestion control. Modern TCP implementations used in production operating systems have not strayed too far from TCP Reno.

2.2 Fast Retransmit and Fast Recovery

Fast retransmit and fast recovery were not described in the initial paper [23], but were implemented simultaneously and were described later [10, 41]. One serious problem with TCP Postel and TCP Nagle is that the loss of a single segment will cause the sender to wait for an RTO before retransmitting the lost segment. Timeouts were typically 500 ms or 1000 ms, much longer than the RTT of the TCP connection. While the sender was waiting for a timeout, the sliding window algorithm prevented it from transmitting more data.

Instead of reducing the RTO timer, Jacobson invented a simple heuristic called fast retransmit to detect packet loss (with high probability) as early as possible, and much sooner than the RTO timer would normally expire. When a segment is lost in the network, the receiver will receive the following segment, which will not be the segment that the receiver was expecting. The receiver will buffer this out-of-order segment and will transmit an acknowledgement of the last in-order received segment, which will be a duplicate acknowledgement. If the receiver continues to receive future segments other than the one it expects next, then the receiver will continue to transmit the same duplicate acknowledgements. Fast retransmit allows the receiver to detect a series of three duplicate acknowledgements and to retransmit the missing segment immediately.

Why three? A receiver can transmit a duplicate acknowledgement for three reasons. The first reason is packet loss, as previously described. The second reason could be that one segment was delayed in the network and arrived later than segments that were transmitted later. We call this phenomenon packet reordering. This case is indistinguishable from packet loss until the receiver finally receives the delayed packet. The third case is that the network could duplicate packets explicitly, which is typically a sign of a fault. Jacobson found that three duplicate acknowledgements was a good balance between tolerating packet reordering, and being responsive to packet loss. The explicit packet duplication case was ignored due to its rarity.

Furthermore, Jacobson concluded that packet losses on the Internet were most likely caused by router congestion rather than transmission errors, meaning that following a fast retransmission, the conservative action would be to reduce the sender's offered load. In TCP Tahoe, fast retransmit transitions back into slow start, followed soon after by congestion avoidance. TCP Reno introduces a third state: fast recovery. During fast recovery, a sender will continue to transmit new data every time an additional duplicate acknowledgement is received from the network. This follows the conservation of packets principle. Once an ACK is received for the retransmitted segment, the sender transitions from fast recovery into congestion avoidance, skipping slow start altogether.

Jacobson designed fast retransmit and fast recovery with a specific network model in mind: a low bandwidth, wired, unipath network, just like the one he used to connect from LBL to UC Berkeley. As network technology has evolved over the past 20 years, the assumptions underlying fast retransmit and fast recovery have caused major problems. For example, TCP is known to perform poorly on large bandwidth-delay product networks [21], wireless networks [4], and networks with multiple paths. Modifying TCP to perform well on these types of networks has been the subject of hundreds of papers. We will limit our scope to studying only multipath networks in this paper.

2.3 Packet Reordering in Multipath Networks

In an ideal network, packets would never be reordered, for two reasons. First, presumably, the original transmission order is the order in which the sender wishes the receiver to receive the packets. Second, receiving the packets in order simplifies the design of the receiver. TCP is robust to packet reordering in the sense that each TCP segment (packet) caries a sequence number, and the TCP receiver uses this sequence number to ensure that packets are placed in their original transmission order before delivering the data to the application.

The largest example of a multipath network is the Internet itself. The Internet contains potentially thousands of paths between two end hosts. In principle, each packet of a TCP flow could be sent over a different path and the IP protocol would ensure that all packets reach their destination. When packets from the same flow take two or more paths through a network, it becomes very likely that packets will arrive out of order. The TCP receiver will reorder the packets before presenting the data to the application, so this should not be a problem in theory. But in practice, the TCP sender's fast retransmit mechanism will often mistake packet reordering for packet loss and will continue to reduce its offered load to unacceptably low levels. In fact, multipath routing remains mostly unused principally because of this reason.

Data center networks are very different from the Internet. The Internet topology is irregular, and there exist multiple unequal cost paths between pairs of hosts. Latencies are measured in the tens to hundreds of milliseconds. The routers used in the Internet typically have very large DRAM-based packet buffers. In contrast, data center networks usually have a regular topology with multiple equal cost paths between pairs of hosts. Latencies are typically 100 μs. And data center switches often have a minimum amount of SRAM-based packet buffers. The focus of this paper is on improving multipath TCP performance in data centers, but we focus mainly on the work in wide-area multipath TCP because of its richer history.

3 Measurement Studies

As mentioned in the introduction, routers and switches try very hard to prevent packet reordering. But despite these efforts, several studies have found a significant degree of packet reordering in the Internet. The degree of reordering is likely to be much worse in data center networks that use per-packet load balancing. This section synthesizes the findings of several major studies of packet reordering to answer two important research questions.

3.1 How Common is Packet Reordering?

From about 2,880 TCP connections to a single server, most of which transferred less than 100 KB of data, Mogul found that about 125, or 4.3%, of those connections had at least one reordered packet [32]. Furthermore, the distribution followed a power law. For example, approximately 100 connections had 1 reordered packet, while approximately 1 connection had 100 reordered packets. Unfortunately, Mogul does not provide insight into the degree of reordering, i.e., how late the reordered packets were when they finally arrived. Without this data, it is not possible to determine the number of spurious retransmissions.

Paxson analyzed 20,000 TCP connections between 35 Internet nodes and found that packet reordering was actually common in the Internet with 36% of all connections having at least one reordered packet, and 2% (0.6%) of all data (ACK) packets arriving late [36]. In fact, one connection had 36% of its packets reordered.

Paxson found that reordering was not uniformly observed and some sites exhibited a much higher degree of packet reordering than others. Paxson writes,

> "Reordering is also highly asymmetric. For example, only 1.5% of the data packets sent to `ucol` ... arrived out of order [, whereas 15% of the data packets sent by the `ucol` site ... arrived out of order]. This means a sender cannot soundly infer whether the packets it sends are likely to be reordered, based on observations of the acks it receives, which is unfortunate, as otherwise the reordering information would aid in determining the optimal duplicate ack threshold to use for TCP fast retransmission."

Paxson found that in some cases, connections with large degrees of packet reordering had no packet loss and no spurious retransmissions due to fast retransmit. He found that for every 22 "good" retransmissions due to packet loss, there was one "bad" retransmission due to mistaking packet reordering for packet loss. He stops short of computing the lost throughput due to these spurious retransmissions.

Bennett, Partridge, and Shectman analyzed bursts of ICMP ping packets between hosts that all transit the same router and were able to correlate the degree of reordering with the configuration and instantaneous load of this router. They found that over 90% of "connections" experienced packet reordering. In some sense, this experiment is incongruent with the previous TCP studies since routers may treat ICMP packets differently than TCP packets, and the small size of ping packets only increases the chances of reordering due to parallelism, since more pings will fit in the pipe than the larger TCP segments.

Iannaccone, Jaiswal, and Diot conducted a much larger and more complete study of several million TCP flows passing through a single core router [19]. They found that less than 5% of TCP flows experience packet reordering and less than 2% of the individual packets within these flows. This disagrees with earlier studies. They also found that only 40% of reordered packets will trigger a fast retransmit. Only 10% of reordered packets were delayed by 8 packets or more.

Laor and Gendel emulated reordering on a backbone link and showed that flows with a very small RTT of less than 1 ms are more robust to packet reordering and have a higher throughput over time [24]. This is most likely due to the fact that with a small RTT, the sender's congestion window never gets very large, so it takes a much shorter amount of time to recover from a decrease in the congestion window. This is very good news for data center networks that have a small bandwidth-delay product.

3.2 Why do Packets get Reordered?

Mogul suggested that packet reordering may be caused by "multiple paths through the Internet" [32], but does not elaborate.

Paxson claims that rapidly-oscillating routes can lead to packet reordering [36], and provides evidence that the two are strongly correlated. Paxson also provides evidence that routers can sometimes "freeze" while processing a routing update, which can lead to packet reordering. For these reasons, Paxson categorizes packet reordering as pathological, i.e. abnormal, behavior.

Bennett, Partridge, and Shectman [5] claim that packet reordering is actually completely normal behavior and can be explained simply as the effects of parallelism inside routers as well as striping across parallel external links, both of which are commonly used for bandwidth scaling. Furthermore, reordering increases as load increases.

4 Solutions Described in the Literature

The measurement studies presented in section 3 revealed that packet reordering is prevalent in the Internet, and that packet reordering interacts poorly with fast retransmit, leading to poor TCP performance. Many solutions have been proposed to improve TCP performance under high degrees of packet reordering. This section describes these proposed solutions independent of their original implementation and analysis. Section 5 describes and compares the original systems that implemented these solutions. Our rationale for separating the solutions from their original implementations is that in most cases, multiple solutions can be combined into a single system.

There are roughly four different types of solutions described in the literature. The first solution is to solve the packet reordering problem at a lower layer in the protocol stack, such that TCP won't even receive DUPACKs. The second solution is to dynamically adjust dupthresh, the number of DUPACKs required to trigger fast retransmit. The third solution is to enter fast retransmit, by delaying the start of fast recovery. The fourth solution is to detect when a retransmission was spurious and to restore the original cwnd and ssthresh. This involves being able to differentiate between good retransmissions and spurious ones. These solutions are depicted on a timeline in Figure 2.

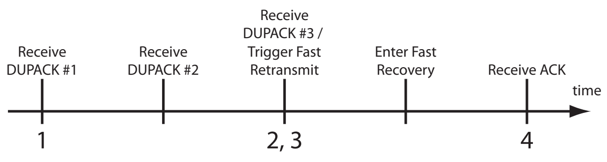

Figure 2: When each of the four solutions can be used.

4.1 Solution 1: Solving the Problem at a Lower Layer

For deployment reasons, it may not be possible to modify TCP, and thus lower-layer solutions become attractive.

4.1.1 Reorder Buffer

Most switches and routers are designed to maintain packet order, even though it adds cost and complexity. Despite this, packets still arrive reordered at end hosts. One way to improve the performance of TCP, without modifying TCP, is to reorder the packets before TCP receives them using a separate reorder buffer. This would prevent reordered packets from reaching the receiver, thus preventing DUPACKs from reaching the sender and causing a spurious retransmission. It would also prevent reordered ACKs from reaching the sender.

A reorder buffer could be implemented in software as a Layer 3.5 shim layer between IP and TCP. Or it could be implemented in hardware in the NIC as a new Layer 2.5, similar in spirit to a TCP offload engine. It is not immediately clear which implementation would be more cost effective and more research should be performed.

4.1.2 Network Coding

Another way to shield TCP from reordering is to use network coding. Network coding can be defined as "coding above the physical layer in a packet network" [17]. In its full generality, network coding allows each node or switch in a network to extract the information from one packet, mix it with information from a local cache of packets, and then generate another newly encoded packet to be transmitted to the next hop of the network. The encoded packet then contains information from all of the mixed packets. The receiver of these encoded packets can forward them as is, or decode them after enough encoded packets have arrived.

Network coding effectively removes the notion of ordering from a sequence of TCP segments; if there is no ordering, then there can be no reordering. The question then becomes how best to integrate network coding into TCP.

Although network coding has shown promising results for wireless networks, it may not be the best solution for a wired network; there are important tradeoffs which must be considered. Communication latency increases because the decoding node must collect enough packets before the original packets can be recovered. Larger buffers are also required on the sender to hold the original packets to be mixed, and on the receiver to hold the mixed packets to be decoded. However, network coding provides many benefits. In certain topologies and communication patterns, network coding can actually increase throughput by reducing the total number of required hop-by-hop transmissions. Network coding, specifically erasure coding, can mask packet loss by adding redundant information to the packets; the original set of packets can be decoded even if only a subset of the encoded packets are received. Network coding also slightly increases security since an eavesdropper must collect enough encoded packets before any original packet can be decoded.

4.2 Solution 2: Dynamically Adjusting dupthresh

In 1988, on a 56 Kb/s wired Internet path with three routers, Jacobson found that three identical DUPACKs was a good estimator for packet loss. The magic number "3" was thus hardcoded into every implementation of TCP for the next decade, with little question as to why this number was optimal. Let us consider replacing this constant with a variable, named dupthresh. Just as TCP's RTO (retransmission timeout) is dynamically adapted depending on the estimated RTT, one could imagine adapting dupthresh depending on the measured degree of reordering in the network [35,36]. This could potentially reduce the number of spurious retransmissions because it could take more than three DUPACKs to trigger fast retransmit. Unfortunately, this will also delay a fast retransmit when packet loss occurs, so care must be taken not to be too tolerant of packet reordering.

4.3 Solution 3: Delaying the Onset of Congestion Control

Another possibility is for the sender to avoid entering fast recovery immediately after fast retransmit [35, 36]. This allows the sender more time to wait for an ACK for a delayed segment and like solution 2, has the possibility of not invoking congestion control. The downside is that it makes TCP less responsive to packet loss. This solution is very similar to solution 2 with similar pros and cons.

4.4 Solution 4: Detecting and Recovering from Spurious Retransmissions

The most devastating effect of a spurious retransmission is that the sender needlessly reduces its congestion window by a factor of 2. One simple improvement to TCP is to detect these spurious retransmissions and to rollback the congestion window to its state exactly before fast retransmit was triggered [5]. The argument is as follows: since packet reordering is not packet loss, the path between the sender and the receiver is not congested and therefore the congestion window should not be reduced. Although rollback alone will not prevent the lost throughput and added latency from a spurious retransmission, the biggest gains come from restoring the congestion window. Otherwise it would take many ACKs to restore the window to its original value.

This of course requires that the sender can determine when a retransmission was spurious. Standard TCP Reno lacks such a technique, which has been referred to as *retransmission ambiguity* [27]. The literature has proposed at least three techniques for eliminating retransmission ambiguity, and thus determining when a packet retransmission was spurious.

4.4.1 ACK Timing

If an ACK "comes back far too soon" after a retransmission [5], then with extremely high probability the ACK belongs to the original transmission and not the retransmission. Hence, packet reordering occurred and not packet loss so the retransmission was spurious.

But how should the phrase "far too soon" be defined? One suggestion is to use 75% of the estimated RTT [3,8]. It is not clear what this value should be. In fact, the other options are more robust techniques that are not subject to variations in the RTT.

4.4.2 Timestamping

TCP supports an option to place a timestamp in a data segment or an ACK segment [21, 22]. If a sender places a timestamp in a retransmitted data segment, then one of two things may happen. First, the sender could receive an ACK with the same timestamp. This would indicate that the original data segment was lost but the retransmitted segment was received. Second, the sender could receive an ACK with an older timestamp. This would indicate that the original data segment was delayed by the network and arrive out of order. Hence, the retransmission was spurious and the congestion window can safely be rolled back [3].

One problem with using timestamps in every segment and every ACK is that each timestamp consumes 12 additional bytes [20]. In any case, the sender need only remember the timestamps associated with retransmitted segments rather than all segments, although some TCP variants such as TCP Vegas [11] actually do keep track of timestamps for all unacknowledged segments.

4.4.3 DSACK

DSACK [30] allows a receiver to notify a sender whenever it receives a duplicate segment, as well as the sequence number of that segment. If the sender keeps track of which segments have been retransmitted, then the sender can determine when packet reordering has occurred. DSACK is an extension of SACK, and due to SACKs popularity as well as its communication efficiency, DSACK is commonly accepted in the literature as the best technique for detecting spurious retransmissions.

Name	Year	Extends	S1	S2	S3	S4
TCP Eifel	2000	TCP Reno				✓
TCP-LPC	2002	TCP Reno		✓	✓	
TCP Westwood	2002	TCP Reno				✓
TCP-BA	2002	TCP Eifel, SACK		✓	✓	✓
RR-TCP	2003	TCP-BA		✓		✓
TCP-PR	2003	TCP Reno				✓
TCP-DCR	2004	TCP SACK			✓	
TCP-NCR	2006	TCP-BA, RR-TCP, TCP-DCR		✓		
TCP/NC	2009	TCP Vegas	✓			

Table 1: TCP variants.

5 TCP Variants

This section describes the many TCP variants that have been proposed over the past decade to solve the TCP packet reordering problem. Each of these variants implements one or more of the solutions from section 4. Table 1 summarizes these TCP variants.

Some TCP variants were designed specifically to solve the packet reordering problem in multipath networks. Others were designed to solve the packet corruption problem in wireless networks, although they often draw from the same solution space. The fact that two different communities are working in the same area, as well as the fact that the computer science (ACM) and electrical engineering (IEEE) communities often don't cross pollinate as much as they should, has led to duplication of work and rediscovery of these basic solutions. Therefore, one should not be surprised if a TCP variant proposed later is actually less sophisticated than one proposed earlier.

We compare only the high-level features of these TCP variants. A performance comparison o determine the relative benefits and drawbacks of the different solutions would require experimentation. See section 7.

5.1 TCP Eifel

TCP Eifel [27] was designed to improve the performance of TCP across wireless links, where packet loss due to corruption is mistaken for congestion, causing the sender to keep reducing its congestion window. TCP Eifel uses solution 4: detecting and recovering from spurious retransmissions and spurious timeouts. For detection, TCP Eifel uses timestamps embedded in each data segment and each ACK segment. When a retransmission or timeout is found to be spurious, TCP Eifel restores cwnd and ssthresh to their former values.

The authors note that this solution can cause a sender to transmit a burst of packets after restoring the congestion window, which is undesirable. The authors recommend using a separate "burst pacer" to compensate. It is known that ACK reordering can also lead to bursts [5], so this suggestion should be given consideration.

Unfortunately, the paper does not provide a good analysis of the performance improvements of TCP Eifel over TCP Reno. One of the arguments of why such an analysis is difficult is that TCP Eifel's enhancements are only used when packet reordering occurs in the network to a large enough degree to trigger spurious retransmissions. One experiment that would have been helpful is to vary the frequency and degrees of reordering in a TCP flow while plotting the average throughput.

TCP Eifel was implemented as a modification of an existing TCP stack. In order to perform experiments, a new Layer 2.5 protocol was implemented between the network layer (IP) and the link layer (PPP). This layer was called "hiccup" and was used to artificially delay certain packets, thus causing reordering. Experiments were conducted by connecting two machines directly with a 9.6 Kb/s serial line.

5.2 TCP-LPC

Lee, Park, and Choi [25] did not cite any of the common literature on packet reordering and seem to have been working in this area independently. TCP-LPC was submitted 13 months after TCP Eifel's publication, and many of the original solutions were proposed in [36] and [5].

TCP-LPC uses solution 2 for the sender and solution 3 for the receiver. For the sender, they statically set dupthresh based on the number of possible paths between sender and receiver. They claim their heuristic is supported by simulation. The receiver waits a short time period before transmitting a DUPACK. Note that this is mostly equivalent to a sender waiting before triggering fast retransmit, except that limited transmit will not be invoked.

5.3 TCP Westwood

TCP Westwood [28] was designed for wireless networks. TCP Westwood is unique among the TCP variants studied here. Instead of avoiding spurious retransmissions, or recovering from them, it simply ignores them. Instead, a TCP Westwood sender estimates the actual end-to-end bandwidth by sampling the ACKs. If this instantaneous ACK bandwidth is large, then TCP Westwood assumes that fast retransmit does not indicate congestion, and so only reduces the congestion window by a small amount. On the other hand, a timeout would indicate severe levels of congestion, enough to stop the flow of ACKs. TCP Westwood then would drastically reduce the congestion window.

Simulations of TCP Westwood show that it performs well on wired, multipath networks [15]. However, these simulations also show that TCP Reno and TCP SACK perform just as well when the RTT is less than 20 ms and when only 2 or 3 paths are used. For 4 paths, TCP Westwood performs an order of magnitude better.

5.4 TCP-BA

TCP-BA [8] directly improves upon TCP Eifel by retaining solution 4, but also adding solution 2: the ability to dynamically increase dupthresh according to the degree of packet reordering in the network. They evaluate three techniques for increasing dupthresh after a spurious retransmission: constant increment, increment relative to length of reordering event, and an exponentially-weighted moving average of the length of the reordering events. The first two have good performance but EWMA does not.

They also evaluate solution 3: waiting a short time after receiving the third DUPACK before triggering a fast retransmit. Their experiments show that their solution 3 performs the best in terms of throughput and unnecessary retransmissions. Unfortunately, solution 3 requires an extra timer.

One minor difference between TCP-BA and TCP Eifel is that TCP-BA uses DSACK rather than TCP timestamps to detect spurious retransmissions.

To solve the burstiness problem, they use limited transmit [16], which allows the sender to transmit new TCP segments whenever DUPACKs arrive in fast recovery mode. Normally, limited transmit only allows sending new data for the first two DUPACKs; their extension allows TCP-BA to send one new TCP segment whenever two additional DUPACKs arrive.

Unlike TCP Eifel, these experiments were conducted entirely in simulation using the ns-2 simulator. They extend the simulator to introduce a router that reorders packets internally. However, their model of reordering is very simplistic and does not try to model an actual router. Instead, they wait until a queue of packets forms and randomly swap two packets in the queue. With simulation, they show that one packet reordering event per second is enough to drop performance by 7.4%.

They determined that their solutions 2 and 3 only improved performance by approximately 1%, and conclude, “ *. . . that the impact any particular compensation scheme has on performance is minimal and that the performance increase comes from [solution 4] when a retransmission is determined to be spurious.*”

5.5 RR-TCP

RR-TCP [45] extends TCP-BA by adding the ability to *reduce* dupthresh as well as increase it. The authors claim that this is important since TCP-BA will keep increasing dupthresh until eventually not enough DUPACKs arrive and a timeout occurs, at which point TCP-BA will reset dupthresh to 3. RR-TCP did not adopt solution 3.

RR-TCP also extends limited transmit to send up to an entire congestion window of data when in the fast recovery state. This is more aggressive than TCP-BA, which only transmits a new TCP segment for every two DUPACKs received.

5.6 TCP-PR

TCP-PR (Persistent Reordering) [9] is similar to TCP Westwood in the fact that it ignores DUPACKs that would trigger a fast retransmit. Instead, TCP-PR timestamps every segment, and ACKs that do not arrive before the timeout get retransmitted. This is a very straightforward design but could suffer from implementation challenges, especially on high-speed links.

5.7 TCP-DCR

TCP-DCR (Delayed Congestion Response) [7] was designed for wireless networks. TCP-DCR extends TCP SACK and implements solution 3: delaying the onset of congestion control after receiving three DUPACKs. The delay is set to the RTT, which is just below the RTO value.

5.8 TCP-NCR

TCP-NCR (Non-Congestion Robustness) [6] is a synthesis of TCP-BA, RR-TCP, and TCP-DCR. Fundamentally, TCP-NCR only implements solution 2: dynamically varying dupthresh. However, dupthresh is set to be equal to a congestion window’s worth of data, i.e. one RTT. Notably missing is a solution 4; the authors suggest that TCP-NCR can be combined with any other TCP that features a solution 4.

TCP-NCR offers two limited transmit modes. Careful limited transmit is essentially the same behavior as TCP-BA, where one new data segment is sent for every two DUPACKs. Aggressive limited transmit is the same as RR-TCP, where one new data segment is send for every DUPACK.

RFC4653 does not present simulation results, only a description and a specification.

5.9 TCP/NC

TCP/NC (Network Coding) is a radical idea [42]. Although TCP/NC was designed to overcome lossy links in wireless networks, it might also be suitable for solving the packet reordering problem in wired multipath networks. TCP/NC is implemented as a Layer 3.5 protocol, meaning that it can be used with any TCP implementation. The paper uses TCP Vegas [11] as the base implementation due to its ability to use increases in the RTT as congestion signals. However, the TCP/NC may need to be heavily modified to work well on wired networks, since TCP/NC transmits extra packets to mask packet loss, which could lead to congestion.

6 Critical Analysis

Without realistic measurements of actual or simulated data center networks, it is difficult to perform a quantitiative analysis. Nevertheless, we will attempt such an analysis.

6.1 Which Solution is Best?

First, given the lengths of Ethernet cables in data centers, the delay through individual packet switches, and the number of switch hops between end hosts, a typical data center RTT is 100 μs. Second, Ethernet specifies a maximum payload size of 1500 bytes[2]. This payload size corresponds to 1538B "on the wire," which takes 12.304 μs to transmit. Therefore, the *pipe* size (maximum window size) is 8 packets.

Consider the following scenario. Alice is transmitting to Bob and has reached a steady state of 8 packets per window. Over the duration of 7 RTTs, Alice would send 8*7=56 packets to Bob, in the absence of frame corruption, congestion, and packet reordering.

Now consider what would happen if during the first RTT, one of Alice's packets was delayed by exactly three frames. This would cause Bob to generate three DUPACKs, which would cause Alice to enter Fast Recovery. During the second RTT, Alice's Fast Retransmit mechanism would (spuriously) retransmit the delayed frame. This would reduce the number of good packets from an ideal of 16 (8*2) to 15 (8+7). During the third RTT, Alice has already exited Fast Recovery and has re-entered Congestion Avoidance, but with cwnd reduced by a factor of 2. So during the third RTT, Alice will only transmit 4 frames instead of 8. This number will increase by 1 each RTT, until the 7th RTT, when Alice has grown cwnd back up to the original and maximum 8 frames. The total number of good frames transmitted during this period would be 8+7+4+5+6+7+8=45, which is 80% of the ideal number of 56 frames.

From the above analysis, the spurious retransmission accounts for only a single lost frame, but the associated Fast Recovery, and the corresponding halving of the congestion window, accounts for 10 lost frames. For WANs with much larger RTTs, this difference is even more pronounced. From this analysis, we predict that solution 4, detecting and recovering from spurious retransmissions, is likely to be the most important solution employed in any high-performance data center TCP implementation. Solutions 2 and 3, which modify dupthresh to prevent spurious retransmissions, will most likely have very little benefit given the presence of solution 4.

Solution 1, such as either a reorder buffer or network coding, is unlikely to yield a satisfactory solution to the original problem. The primary reason is the lack of economy of mechanism; any solution 1 will be duplicating functionality that already exists in TCP. In the case of the reorder buffer, it would be duplicating TCP's reorder buffer, and would be hiding important details from TCP. It would also delay packet delivery to TCP, thus inflating the RTT and increasing the variance of the RTT and the corresponding bustiness of the traffic. Network coding is nice in theory, but as we saw in previous sections, the current proposal for network coding with TCP completely ignores congestion control. The effort to produce a network coded TCP with congestion control is probably much more than simply implementing solution 4 using DSACK.

6.2 What about Timeouts?

For data center networks, large timeouts must be avoided at all costs. Recall that a timeout is necessary when Fast Retransmit is impossible, such as when an entire sender's window of packets has been lost, or the pathological case of when dupthresh has grown so large that it is impossible to trigger Fast Retransmit. The original minimum TCP timeout period, RTO_{min}, was approximately 1 second. Later implementations of TCP reduced RTO_{min} to 500 ms, and most recently to 200 ms. However, even 200 ms is much too large for data centers. Note that this is a minimum timeout period, and the actual timeout period is slightly larger than the estimated RTT, but never less than the RTO_{min}.

Assuming an RTT of 100 μs, 200 ms accounts for 2,000 RTTs, during which time a total of 16,000 packets could be sent. If a sender loses an entire window of 8 packets, then from the TCP connection's point of view, this is equivalent to losing 16,000 packets, although the network interface can use this idle time to send packets from other TCP connections.

To avoid timeouts, solution 2 must be used very carefully to avoid the pathological case mentioned previously. Solution 3 can be tailored to trigger Fast Retransmit right before a timeout, which is ideal. This hints that solution 3 might be easier to implement in practice and

[2]Most network equipment supports Jumbo Ethernet frames, i.e. frames with a payload strictly larger than 1500B. However, Jumbo frames are seldom used in practice due to their buffering requirements and long latencies in store-and-forward networks.

is more likely to yield good performance across a wide range of networks and use cases. Solution 1 also must be used with care not to delay packets longer than the timeout period, which complicates the design of solution 1 since it may not have access to the TCP's estimated RTT and corresponding timeout value. Solutions 2 and 3 will have direct access and will not suffer from this interface problem.

One must also question the usefulness of a timeout period that is 2,000 times larger than the RTT. Others have experimented with lowering RTO_{min} to 200 μs and have reported very good performance, and solving a different but important data center network problem called *incast*. See section 7.5. We recommend lowering RTO_{min} as well, and 200 μs seems like a good place to start.

7 Proposed Experiments

Our group has constructed a small fat-tree data center network that will allow us to conduct TCP and packet reordering experiments. Figure 3 shows the network topology of our testbed, which consists of 37 computers in a single rack. Each computer occupies one rack unit (1U).

One computer, called the *NAT box*, is used as a network address translator. The NAT box isolates our testbed from the extraneous traffice on the UCSD CSE network, while also allowing the rest of the computers to access the Internet to download software updates. The NAT box is used to compile software, as a repository of scripts, configuration files, compiled binaries, and test results, and to allow direct ssh access to the remaining computers in the testbed. Finally, the NAT box hosts an OpenFlow controller [31] during the actual experiments.

The remaining 36 computers are divided into 16 end hosts and 20 custom 4-port gigabit Ethernet (GbE) switches. The end hosts run standard data center applications such as MapReduce [13], as well as network performance benchmarks such as iperf. The custom switches each run an OpenFlow client named *secchan* and connect to the OpenFlow controller on the NAT box. The custom switches also contain our switch modifications to the secchan program as well as hardware modifications to the NetFPGA [34].

The testbed contains two separate networks, a control plane and a data plane. All 37 computers connect to the control plane via a single 48-port GbE switch[3]. The control plane is used for ssh logins, for providing Internet access for software updates, and for setting up the experiments. During experiments, the control plane is used to allow the computers to coordinate with the NAT box; the end hosts write their results to the NAT box and the custom switches coordinate with the OpenFlow controller on the NAT box.

The data plane is only used during experiments. Each end host has a direct connection to one of the custom switches. The rest of the ports on the custom switches are used to form the fat-tree interconnect. Since our fat-tree multi-stage switch is a Layer 2 switch, only the end hosts have IP addresses on the data plane. The IP addresses are part of the 10.0.0.0/8 private address range.

7.1 Per-Packet Load Balancing Algorithms

Exactly how should each NetFPGA perform per-packet load balancing? It is important to realize that not every packet should be load balanced. Only packets entering a switch element on a downward-facing port, and destined for an upward-facing port, should be load balanced. A consequence of this is that load balancing only happens on edge and aggregation switches, not on core switches.

One naive load balancing algorithm would be to maintain a single counter corresponding to an upward-facing output port to forward the next packet out of. This counter would wrap around once all upward-facing ports have been enumerated. While simple and straightforward, this algorithm may not perform well in practice. Consider the scenario where two different TCP flows are arriving at an edge switch from downward-facing ports, one on port 0 and one on port 1, both sustaining 1 Gb/s. Our naive load balancer may not distribute the packets from these two flows evenly across both upward-facing ports. Synchronization could occur where all packets arriving at port 0 are forwarded out port 2, and all packets arriving at port 1 are forwarded out port 3. This pathological condition is not the intended behavior of the load balancer. However, for reasons yet unknown, this pathological case may not occur in practice, making this simple algorithm a very good choice.

One can also imagine more complicated scheduling policies, such as least loaded output port, least recently used output port (within some given time window), and uniform random. We propose a second literature search to complement this survey, focused on per-packet load balancing algorithms. The best algorithm may have already been discovered, and a literature search would both generate a menu of choices for further experimentation, as well as direct us to likely candidate algorithms that are expected to perform well. We propose implementing several of these per-packet load balancing algorithms in hardware using a NetFPGA, and running these algorithms on our testbed.

Since the fat tree is a regular network, we expect that

[3]Humorously, this single 48-port switch is both more powerful and an order of magnitude less expensive than our experimental switch built from 20 servers and NetFPGAs. However, our fat-tree network architecture can scale much larger than any single-stage switch, and any commercial design would reduce cost by omitting unnecessary components.

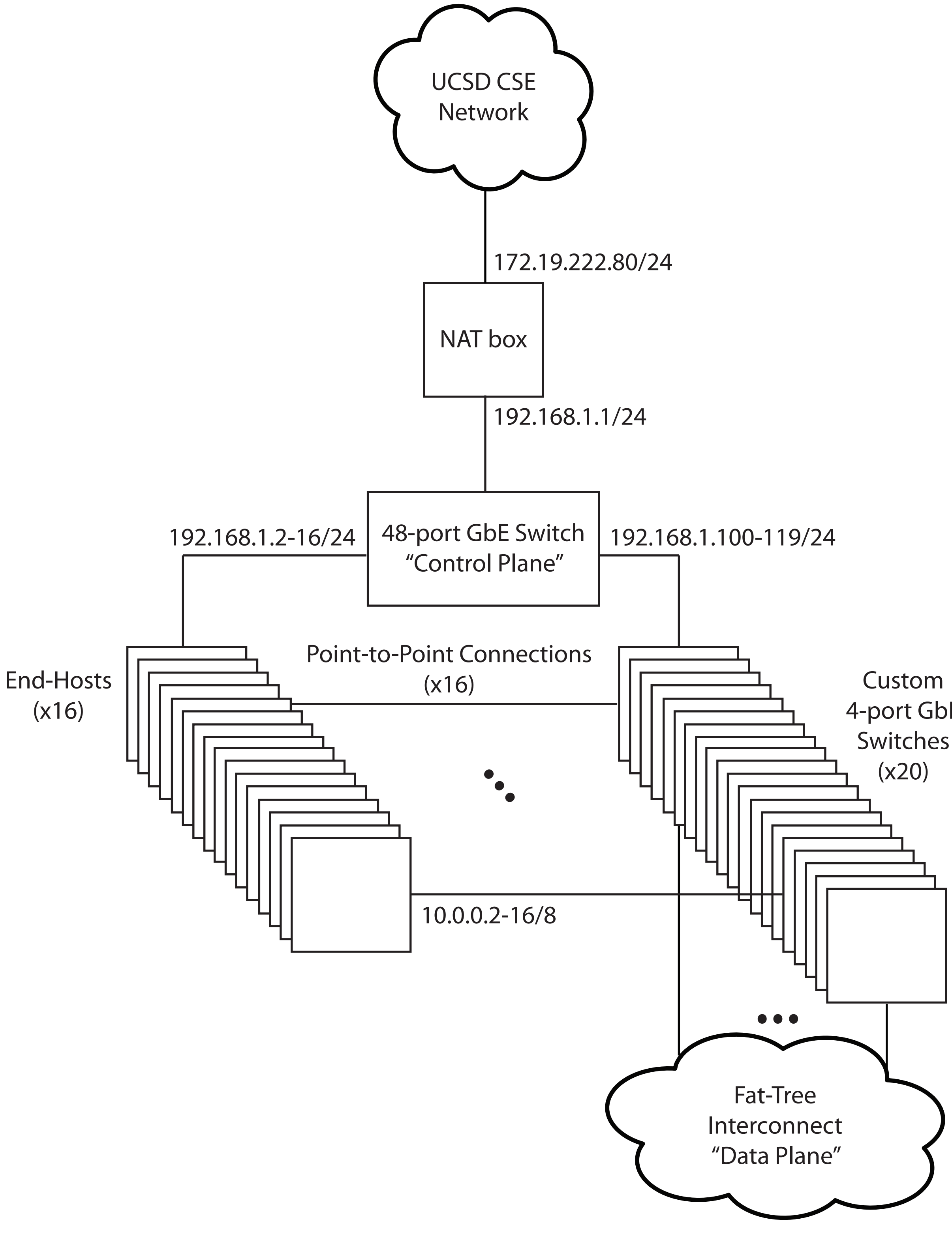
UCSD CSE Network
172.19.222.80/24
NAT box
192.168.1.1/24
192.168.1.2-16/24
48-port GbE Switch "Control Plane"
192.168.1.100-119/24
End-Hosts (x16)
Point-to-Point Connections (x16)
Custom 4-port GbE Switches (x20)
10.0.0.2-16/8
Fat-Tree Interconnect "Data Plane"

Figure 3: Our experimental testebed for conducting TCP and packet reordering experiments.

all algorithms will perform equally well on an unloaded network with a single TCP flow between two end hosts. In fact, all packets should arrive in order, making TCP reordering unnecessary. The real test will come with cross traffic and multiple TCP flows from multiple end-host pairs. In this case, we hypothesize that the more complicated algorithms, and uniform random, will perform better at avoiding congested links than the more naive algorithms. To generate cross traffic, we propose using multiple iperf instances on various end-host pairs to allow scaling the experiment from a single flow up to $O(16^2)$ flows.

7.2 Popular TCP Variants

The Linux kernel includes the following 10 TCP variants: TCP Reno, TCP BIC, TCP CUBIC, TCP Hybla, TCP Illinois, Scalable TCP, TCP Vegas, TCP Veno, TCP Westwood, and TCP Yeah. Each variant has a different goal and history, such as improving performance over large bandwidth-delay product networks or wireless networks. None of these variants was designed explicitly to improve multipath performance such as TCP Eifel [27] or RR-TCP [45]. However, we have already seen simulation results in section 5 that TCP Westwood performs well on multipath networks. And it may be the case that other "wireless TCPs" such as TCP Beno or TCP Hybla also perform well. Regardless, it would be important to know which TCPs perform particularly well or particularly poorly on multipath networks.

We propose to repeat the per-packet load balancing experiments of section 7.1 under these 10 different TCP variants, rather than just with TCP BIC, the default TCP variant for Linux. We hypothesize that TCP Westwood will perform the best out of all variants due to its ability to completely ignore DUPACKs, whereas implementations like TCP Reno will perform the worst. The most interesting results will be when the network is heavily loaded with lots of cross traffic, meaning that the amount of reordering in the network due to queueing delay will be maximized.

7.3 Implementing the 4 Solutions

The experiment outlined in section 7.2 is simple to conduct since the variants have already been implemented. It is less straightforward to compare the 4 solutions from section 4. Since there is no single TCP variant that implements all 4 solutions, with sufficient parameterization to allow "turning off" various solutions to determine the performance improvement each solution in isolation donates to the overall performance improvement.

We propose a slightly more laborious experiment in which we take the standard TCP Reno variant and add solutions 2, 3, and 4, to the implementation. We will parameterize the variant in accordance with the recommendations by Blanton and Allman [8], using their same variable names and strategies. We will then repeat the experiment described in section 7.1, varying the different parameters each time.

We hypothesize that the largest performance increase will come from solution 4, and that a simple strategy of restoring the original cwnd and ssthresh variables will be sufficient.

One possible outcome of these experiments is identifying a TCP variant that will perform well over the Internet, over wireless paths, and over multiple paths. We call this variant TCP Awesome.

7.4 Delayed Acknowledgements

Delayed acknowledgements [2] were introduced to reduce the computational requirements of a TCP receiver. Normally, a TCP receiver would acknowledge every received segment immediately. Delayed acknowledgements allow a TCP receiver to wait up to 200 ms before sending an acknowledgement, or the receipt of two segments, whichever happens first. Delayed acknowledgements increase the theoretical goodput of bidirectional TCP flows on Gigabit Ethernet from 897 Mb/s to 921 Mb/s. However, the reduction in the total number of acknowledgements transmitted may not work well in the data center [44].

We propose repeating all previously described experiments with delayed acknowledgements disabled. We hypothesize that this will provide an improvement in throughput of between 0% and 5% for all experiments due to the TCP senders ability to receive ACKs and DUPACKs immediately, thus being able to recover more quickly after a congestion event. The tradeoff is increased bandwidth utilization by the extra ACK segments which will lower throughput.

7.5 RTO_{min}

It has been suggested that RTO_{min} does more harm than good in data center networks and should be eliminated [44]. The reason is that RTO_{min} is on the order of 200 ms, whereas the RTT in a data center network is typically 100 μs. RTO_{min} effectively overrides the RTO timer's estimate of when to timeout and forces the RTO timer to wait much longer, in this case 2,000 times longer.

Similar to section 7.4, we propose repeating previous experiments with RTO_{min} removed in order to determine the performance enhancement that a smaller timeout value has after packet loss.

We hypothesize that eliminating RTO_{min} will make solutions 1, 2 and 3 unnecessary. For example, if the timeout is on the order of one RTT, then one could simply ignore DUPACKs, fast retransmit, and fast recovery, and allow a timeout whenever packets are delayed for too long of a time period. However, solution 4 can be used to detect spurious retransmissions and restore the original cwnd and ssthresh variables almost immediately afterward, meaning that very little throughput is lost due to being in slow start temporarily.

8 Conclusion

Many different TCPs have been proposed that claim to be able to solve the packet reordering problem. However, none of them have been evaluated in the context of multipath data center networks, nor have they found significant deployment on the Internet. One of the reasons for the lack of adoption is that that previous TCPs might be more aggressive than TCP Reno, meaning that TCP Reno flows will suffer, leading to a TCP arms race that could culminate with congestion collapse. Data center networks are not limited by these constraints and have more flexibility with trying different congestion control algorithms.

Today, the consensus seems to be that packet reordering in the wide area is not a pressing enough problem to warrant a change to TCP. However, data center networking is a new area that has not yet examined TCP performance. One of the major differences between TCP in the data center is that the network consists of high-capacity, low-latency links, which very few TCP implementations address. Further, the regularity of data center network topologies combined with multiple equal-cost shortest paths make it intriguing to solve the packet reordering problem so that packet switches can use per-packet load balancing rather than per-flow load balancing.

Our analysis indicates that solution 4, detecting and recovering from spurious retransmissions, is most likely to almost reach the ideal performance of no reordered packets compared to the other three solutions. Solution 4 is also simple to implement using already standardized TCP protocols such as DSACK. We also conclude that the current RTO_{min} of 200 ms is much too large for data center networks where typical RTTs measure around 100 μs. A better RTO_{min} would be something around 200 μs. These two changes, solution 4 and lowering RTO_{min}, are likely to yield excellent performance in per-packet load balanced data center networks with no other changes necessary.

References

[1] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A scalable, commodity, data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2008.

[2] M. Allman, Vern Paxson, and W. Stevens. RFC 2581: TCP Congestion Control, April 1999.

[3] Mark Allman and Vern Paxson. On Estimating End-to-End Network Path Properties. In *SIGCOMM '99: Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 263–274, New York, NY, USA, 1999. ACM.

[4] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, December 1997.

[5] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, December 1999.

[6] S. Bhandarkar, A.L.N. Reddy, M. Allman E., and Blanton. RFC 4653: Improving the Robustness of TCP to Non-Congestion Events, August 2006.

[7] Sumitha Bhandarkar, Nauzad Erach Sadry, A.L. Narasimha Reddy, and Nitin H. Vaidya. TCP-DCR: A Novel Protocol for Tolerating Wireless Channel Errors. *IEEE Transactions on Mobile Computing*, 4(5):517–529, Sept.-Oct. 2005.

[8] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 32(1):20–30, 2002.

[9] S. Bohacek, J.P. Hespanha, Junsoo Lee, Chansook Lim, and K. Obraczka. TCP-PR: TCP for Persistent Packet Reordering. In *Proceedings of the 23rd International IEEE Conference on Distributed Computing Systems*, pages 222–231, May 2003.

[10] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, 1989.

[11] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[12] Vinton G. Cerf and Robert E. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, May 1974.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, December 2004.

[14] Edsger W. Dijkstra. The Structure of the "THE"-Multiprogramming System. In *SOSP '67: Proceedings of the First ACM Symposium on Operating System Principles*, pages 10.1–10.6, New York, NY, USA, 1967. ACM.

[15] Mario Gerla, SS Lee, and G. Pau. TCP Westwood Simulation Studies in Multiple-Path Cases. *Proceedings of SPECTS*, 2002.

[16] M. Allman Hari, Balakrishnan, and Sally Floyd. RFC 3042: Enhancing TCPs Loss Recovery Using Limited Transmit, January 2000.

[17] Tracey Ho and Desmond S. Lun. *Network Coding: an Introduction*. Cambridge University Press, 2008.

[18] C. Hopps. RFC 2992: Analysis of an Equal-Cost Multi-Path Algorithm, November 2000.

[19] Gianluca Iannaccone, Sharad Jaiswal, and Christophe Diot. Packet Reordering Inside the Sprint Backbone. Technical report, Technical Report TR01-ATL-062917, Sprint ATL, 2001.

[20] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, 1992.

[21] V. Jacobson and R. T. Braden. RFC 1072: TCP extensions for long-delay paths, 1988.

[22] V. Jacobson, R. T. Braden, and L. Zhang. RFC 1185: TCP extension for high-speed paths, 1990.

[23] Van Jacobson. Congestion Avoidance and Control. In *SIGCOMM '88: Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 314–329, New York, NY, USA, August 1988. ACM.

[24] Michael Laor and Lior Gendel. The Effect of Packet Reordering in a Backbone Link on Application Throughput. *IEEE Network*, 16(5):28–36, Sep/Oct 2002.

[25] Youngseok Lee, Ilkyu Park, and Yanghee Choi. Improving TCP Performance in Multipath Packet Forwarding Networks. *Journal of Communications and Networks*, 4(2):148–157, 2002.

[26] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[27] Reiner Ludwig and Randy H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM SIGCOMM Computer Communication Review*, 2000.

[28] Saverio Mascolo, Claudio Casetti, Mario Gerla, M.Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 287–297, 2001.

[29] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgment options, 1996.

[30] M. Mathis, J. Mahdavi, Sally Floyd, and M. Podolsky. RFC 2883: An Extension to the Selective Acknowlegdment (SACK) Option for TCP, July 2000.

[31] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[32] Jeffrey C. Mogul. Observing TCP Dynamics in Real Networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 1992.

[33] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, 1984.

[34] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow Switch on the NetFPGA Platform. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, 2008.

[35] Vern Paxson. End-to-End Internet Packet Dynamics. In *SIGCOMM '97: Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 139–152, New York, NY, USA, 1997. ACM.

[36] Vern Paxson. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[37] J. Postel. RFC 791: Internet Protocol, 1981.

[38] J. Postel. RFC 792: Internet Control Message Protocol, 1981.

[39] J. Postel. RFC 793: Transmission control protocol, 1981.

[40] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[41] W. Stevens. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, 1997.

[42] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Michael Mitzenmacher, and João Barros. Network Coding Meets TCP. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2009.

[43] D. Thaler and C. Hopps. RFC 2991: Multipath Issues in Unicast and Multicast, November 2000.

[44] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, and Garth A. Gibson. A (In)Cast of Thousands: Scaling Datacenter TCP to Kiloservers and Gigabits. Technical Report CMU-PDL-09-101, Carnegie Mellon University, February 2009.

[45] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, pages 95–106, November 2003.

[46] Hubert Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 1980.