Benefits and Practicality of Super-Packets

Yunlong Li

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-83
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-83.html

May 16, 2013

Benefits and Practicality of Super-Packets

Yunlong Li
yunlongli@berkeley.edu
UC Berkeley

Abstract

TCP's slow-start was designed to prevent hosts from sending more data into the network than it is capable of transmitting. A key component of slow-start is the initial congestion window or *init_cwnd*. Today, it is set to at most four segments or roughly 4KB of data. As a result, every short-lived TCP connection (henceforth "short flow") suffers a performance hit when it comes to throughput (and by extension, latency), for it does not last long enough to achieve its fair share of the network bandwidth. A commonly proposed technique to lower latency for these short connections is to increase the size of TCP's initial congestion window. However, simply increasing the *init_cwnd* does not protect short flows from packet loss, which affects them more adversely than it affects long-lived TCP connections. In this paper, we introduce the concept of a super-packet to augment the proposal of increasing the initial congestion window. A super-packet is a network abstraction consisting of a finite number of consecutive packets. These packets form a logical unit on which a router makes a forwarding decision. We will present an efficient algorithm for super-packet admission in routers, conduct a security analysis of the algorithm, discuss deployment opportunities, and analyze the benefits of using super-packets.

1. Introduction

TCP's slow-start was designed to prevent a computer host from sending more data into the network than the network is capable of transmitting. A key component of slow-start is the initial congestion window. Today, it is set to at most four segments[*] or roughly 4KB of data. As a result, every short-lived TCP connection (henceforth "short flow") suffers a performance hit when it comes to throughput (and by extension, latency), for it does not last long enough to achieve its fair share of the network bandwidth. In this paper, I propose and analyze a new technique for addressing this problem.

1.1. Motivation

Although most of the bytes transferred in the Internet are in bulk data transfers (e.g., video) [1], the majority of connections are short-lived. Unfortunately, TCP's slow-start forces many of these short flows to take several round-trip times (or RTTs) to complete, because the initial congestion window (or *init_cwnd*) is at most four segments or approximately 4KB of data [1], which is smaller than at least 30% of HTTP Web objects. As the average size of HTTP Web objects increases, the current default for the initial congestion window becomes increasingly less sufficient. A recent proposal by Google calls for TCP's initial congestion window to be increased to at least ten segments (about 15KB) to *(i)* reduce latency for short flows and *(ii)* allow short flows to compete more fairly with bulk data transfers [3]. Their argument goes as follows. Because 80% of HTTP Web objects have sizes within 8KB, an increased *init_cwnd* will allow most short flows to complete within one RTT when there is no packet loss [4]. With respect to bulk data transfers, when a short flow is initiated, it has to compete in the network with these long-running flows that statistically already have a bigger congestion window than TCP's *init_cwnd*. Because the short flow can complete when its congestion window is still quite small compared to that of a

[*] A segment refers to a unit of data in the transport layer, while a packet refers to a unit of data in the network layer. For the purpose of this paper, we will use the term packet to refer to both a packet and a segment.

long-running flow, it may never get a fair share of the network bandwidth. Thus, by increasing the *init_cwnd*, short flows can use more of the network's bandwidth right from the get-go.

Unfortunately, increasing the *init_cwnd* alone is not enough because it does not protect short flows against packet loss. TCP's mechanism for handling packet loss works as follows. Once a packet is lost, for each subsequent packet that the receiver receives, the receiver sends a duplicated ACK message (or duplicate ACK) with a 32-bit acknowledge number that acknowledges the receipt of all bytes prior to the packet loss. Upon receiving three of these duplicate ACKs, the sender will deem the packet that has not been acknowledged lost and retransmit that packet. This mechanism is called fast retransmit. If the sender does not receive three duplicate ACKs in time, a timeout occurs, bringing the congestion window (or *cwnd*), which is one factor that determines how many bytes can be outstanding, all the way down to 1 maximum segment size or MSS. Therefore, losing packets toward the end of a short flow can be detrimental. For example, consider a short flow consisting of eight packets (*init_cwnd* = 8). If three packets out of the last five, two packets out of the last four, or one packet out of the last three were dropped, then there would not be enough duplicate ACKs to trigger fast retransmit. A timeout would occur, bringing us back to the same problem as before: it would take several RTTs for the short flow to complete.

Furthermore, increasing the *init_cwnd* alone does not solve incast—a communication pattern wherein a host issues read request to multiple servers who in turn concurrently respond with an aggregate large amount of data that travels along a congested link on the way back to the host. To put it more concretely, imagine that ten short flows, each consisting of eight packets and destined for the same destination, arrive concurrently at a router with a bottleneck link that does not have enough buffer space to buffer all the packets in time. If the router employs a droptail policy, each of these ten flows may see a few packet drops, and these drops tend to be at the end. Therefore, the majority of flows are likely to experience a timeout.

1.2. Introducing Super-Packets

To address these problems, we introduce the notion of a "super-packet." A super-packet is a finite number of consecutive packets in a flow that is treated by the network as a single logical packet (see Figure 1). When a super-packet arrives at a router, the router makes a forwarding decision for the entire super-packet. The router accepts all packets of the super-packet if it decides to accept the super-packet. If the router decides to reject the super-packet, then it intentionally drops all packets of the super-packet. We call this the "all-or-nothing property."

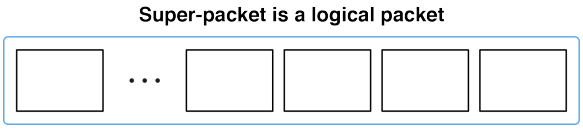

Figure 1

If the sender treats his short flow as a super-packet, he can protect it against packet loss and incast. The amount of protection is roughly proportional to the number of routers that support super-packets in the path of the short flow. In the case where the super-packet is accepted by a router, the sender can be fairly confident that the router will forward all packets of the super-packet. On the other hand, if the super-packet is denied, the sender can try again after a short amount of time, (e.g., after RTO). Because packets travel across the network via multiple routers, some of which with no super-packet support, it is possible that a router with no super-packet support drops a packet that belongs to a super-packet. We will discuss the recovery mechanism in Section 2.3 that handles this case.

There are various challenges associated with developing an abstraction of super-packets. First of all, one must consider the interface that super-packets expose to the layers below and above. Secondly, one must craft an admission algorithm in routers that is easy to reason about and efficient since routers are performance-critical hardware devices. Lastly, one must provide reasons why super-packets are

preferable to normal packets as a means of transport for short flows. In This paper, we will address each of these issues.

1.3. Outline

The rest of the paper is organized as follows. In Section 2, we describe our implementation of super-packets, including the admission algorithm used in routers. Section 3 contains a security analysis of the admission algorithm. Section 4 lays out the experimental results in ns2 that demonstrate the benefits of using super-packets. We look at related work in Section 5 and discuss future research in Section 6. Finally, we conclude in Section 7.

2. Implementation

2.1. Changes to IP Header

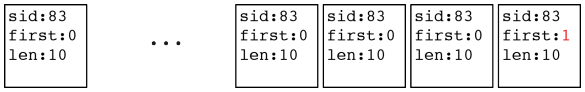

Figure 2

Implementing super-packets requires modification to the network layer (layer 3). Every packet of a super-packet contains basic information about the super-packet (see Figure 2), such as the super-packet id `sid`, which is unique per `(src, dst)` pair, where `src` and `dst` are the source's IP address and the destination's IP address respectively, the length of the super-packet `len`, and the boolean `first` which signifies whether it is the first packet or not. When a super-packet's first packet arrives at a router[*], the router makes a decision about the entire super-packet using the information supplied by the first packet. If it decides to accept the super-packet (i.e., accept the first packet of the super-packet), the router needs to forward every packet of the super-packet. If it decides to drop the super-packet (i.e., drop the first packet of the super-packet), the router drops every single packet of the super-packet.

2.2. Changes in Router

In the router, we implement the admission algorithm shown in Appendix A1. On a basic level, the router needs to maintain a white-list (or a black-list) to remember which super-packet it has accepted (or denied). The router uses this information to make an admission decision for the subsequent packets of a super-packet. In our algorithm, we use a moving hash table `hash` that maps the tuple `(src, dst)` to the tuple `(#_of_total_packets_reserved, #_of_outstanding_packets)`.

A moving hash table is a hash table where each new entry is guaranteed to be present in the hash table for no less than a specified amount of time (in our case `EPOCH`). Internally it is made up of two hash tables `h1` and `h2` that after an `EPOCH` of time undergo the following transformation: `h1 <- h2` and `h2 = new HashTable()` (See Appendix A2). At any given moment, when a new entry is saved in `hash`, it is stored in `h2`. At the end of each `EPOCH`, all entries from `h1` are deleted and replaced by entries from `h2`. We use a hash table to store information per `(src, dst)` to save memory and yet still have a way to perform basic accounting on super-packets. If a source `src` sends a super-packet to a destination `dst`, and the router accepts, we increment its `#_of_total_packets_reserved` by the length of the super-packet `len` and increment the `#_of_outstanding_packets` by `len - 1`. Overall, as long as the

[*] We will use the term "super-packet" to refer to either the first packet of a super-packet or an entire super-packet as a whole. The meaning of the term should be clear from the context. Also, an IP packet without any super-packet information will be referred to as "normal." Thus, all the packets in the Internet today are normal. We will use the phrase "a packet marked with super-packet information" to refer to any packet inside a super-packet. We will use the phrase "a subsequent packet of a super-packet" to refer to a non-first packet of a super-packet. Finally, when we will often address the congestion window size as unit numbers rather than actual bytes (so *cwnd* of 1 is the same as 1 MSS).

`#_of_outstanding_packets` remains above 0, the source `src` has at least one legitimate packet of a super-packet that he has yet to send to the destination `dst`. All subsequent packets of an admitted super-packet will be admitted as well, and the value `#_of_outstanding_packets` is decremented by 1 for each of these packets that arrives at the router.

To test whether a specific packet is part of a super-packet that has been accepted and to determine which internal hash (`h1` or `h2`) to use to do accounting, we use a moving bloom filter to test its membership (shown in Appendix A3). The key to the bloom filter is `(src, dst, sid)`.

2.3. Changes in TCP

Changes must be made to TCP in order to accommodate super-packets. One major change directly addresses what happens if a super-packet is denied. In this case, the sending host can simply resend the super-packet after a timeout. If the super-packet is continuously being denied, then after a few attempts (e.g., 3), the sender can try sending the short flow using normal packets (fall back to regular TCP).

Another major change centers around the question, "What happens when a non-first packet of a super-packet is dropped?" Because packets travel across the network via multiple routers, it is possible that a non-first packet of a super-packet gets lost (via packet corruption or being dropped by a router with a congested link and no super-packet support). In this case, the sending host can simply send the dropped packet as a normal packet (i.e., no super-packet information in the IP header). In other words, we simply strip the packet of its super-packet information and rely on the underlying TCP algorithm to ensure the reliable delivery of that packet.

2.4. Admission Policy

The admission policy we have implemented inside routers is fairly simple. We basically divide the buffer space via a threshold value `t`. For simplicity, assume that all packets (normal packets and packets marked with super-packet information) are of the same size (e.g., 1500 bytes). If not, `t` will simply be an actual size limit (i.e., x bytes). The picture shown in Figure 3 demonstrates a lightly queued buffer. Each grey block represents either a normal packet or a packet marked with super-packet information. Thus, if an entire super-packet of length `len` is queued (i.e., all packets of the super-packet are present in the queue, not necessarily back-to-back), then it will contribute as `len` in the total number of packets in the queue. As long as the number of packets in the queue is less than `t`, the router will continue to accept super-packets as well as normal packets. If the number of packets in the queue is `t` or more, then the router will stop accepting normal packets and any new super-packet. In other words, it will only accept the subsequent packets of the super-packets that it has accepted.

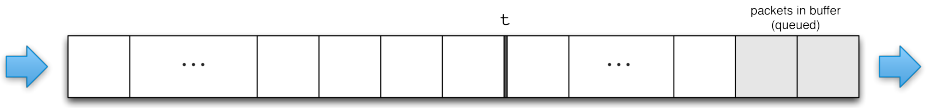

Figure 3

The threshold `t` cannot have one single optimal value for all workloads, and we have used different values of `t` in different experiments. The general rule is that the more super-packets there are in the network, the smaller the threshold `t` should to be. The value of `t` should be dynamically adjusted to accommodate a varying workload.

3. Security Analysis

Security is an important concern when it comes to supporting super-packets in a network. Super-packets may allow malicious users to exploit new vulnerabilities in the network. They may make existing attacks such as IP address spoofing and distributed denial-of-service (or DDoS) more severe. Therefore, our super-packet system (the changes we make to the network layer, the changes we make to TCP, and the actual admission algorithm running in routers) has to be evaluated to determine the feasibility of deployment in an un-safe environment.

3.1. Security Analysis of the Admission Algorithm

The admission algorithm tries to strike a balance between memory consumption and performance. If an adversary were to attack a state table, then she would look at two places. First is the hash table `h2`. With IP address spoofing, an adversary could perform a denial-of-service attack on a router using this table. Since the hash is keyed on `(src, dst)`, she could send an inordinate amount of spoofed first packets of super-packets with random faked IP addresses in the `src` field. The router would create a new entry for each unique `(src, dst)` pair it sees. Depending on how the hash table is implemented, if there are a large number of entries in it, lookup and/or resizing might become very computationally expensive. For example, if the entries in the same bucket of the hash table are chained together in a linked list, then to lookup the `#_of_outstanding_packets` count associated with a particular `(src, dst)` pair, we have to traverse an entire list, which is $O(n)$ in complexity where n is the size of the list. If the router has to perform $O(n)$ computations for each packet with super-packet information to determine whether to accept it or not, it would make the use of super-packets very infeasible.

Another target is the bloom filter `b2`. The bloom filter turns on a few bits for each super-packet that the router accepts. The adversary could increase the likelihood of a false positive by sending an inordinate amount of super-packets. Furthermore, using IP address spoofing, she could try to undermine another sender's `#_of_outstanding_packets` count by sending in spoofed packets that trigger a false positive and cause the router to decrement it.

There are also indirect attacks. Suppose that super-packet support is ubiquitous in the Internet. The adversary could buy rich media ads that include JavaScript that requests the user's browser to send a super-packet to a target server. If many people see these ads around the same time, then that server will be inundated with super-packets.

A more subtle security concern comes from the implementation of the moving hash table and the moving bloom filter. The interface we expose to hosts is that all packets of a super-packet need be sent to the router within the time period `EPOCH`. In reality, a super-packet's state is maintained for a time period between one `EPOCH` and two `EPOCH`s, depending on when the super-packet arrives at the router. An adversary could try to synchronize with the moving hash table/bloom filter to maximize the amount of time her super-packet state is maintained. If every host takes this greedy approach, then we will encounter a large aggregate burst of traffic at the beginning of each epoch.

Because of the security concerns mentioned above, super-packet deployment seems to work only in a trusted environment. One example of a trusted environment is a datacenter.

3.2. Deployment Opportunities

Super-packet deployment seems to be only feasible in a trusted environment (e.g., a datacenter). In datacenters, a typical network topology consists of either two- or three-level trees of switches or routers [6]. In a three-tiered topology, the edge tier is connected to the leaves or hosts of the network. The set of routers that connects to the edge routers forms the aggregation tier. Finally, the routers in the aggregation layer are connected to each other via a set of routers in the core tier.

Oftentimes, many datacenters oversubscribe their resources as a means to lower cost. In a network topology, this translates to an oversubscribed output link of a router in the edge or aggregation tier. There is an opportunity to deploy super-packets whenever a router's output link is oversubscribed, because it can bring all the benefits mentioned in the next section (Section 4).

4. Experimental Results

We conducted experiments in ns2, a discrete event simulator used for networking research [7]. Our super-packet implementation differs from our ideal in that once a router denies a super-packet, we simply fall back to the original TCP implementation (for these experiments, we used TCP New Reno). That is, we do not construct another super-packet to send, but rather let the short flow, consisting of the super-packet, timeout and *cwnd* be reduced all the way to 1 MSS. We then retransmit the short flow using normal packets. Despite using this weaker version of the retransmission scheme, we were able to confirm that super-packets reduced the average latency of short flows across all our experiments.

4.1. Experimental Methodology

Every TCP connection begins with a SYN packet. In our experiments, we tried to remove the effect of the SYN packet on our latency measurements. Although the mechanisms (described in the following sections) we put in are not perfect, we do not believe that the presence of the SYN packet influences our latency measurements significantly.

4.2. Incast

The first experiment we report shows that super-packets solve the incast problem to the best degree possible. Incast is a communication pattern wherein a receiver issues read requests to multiple senders who in turn concurrently respond with an aggregate large amount of data back to the receiver. The response from all senders can travel along a bottleneck link, leading to a dramatically reduced throughput for each sender [5]. Our system tackles the incast problem by concentrating packet loss to as few flows as possible.

Using ns2, we set up a topology, shown in Figure 4, consisting of ten nodes all sending to the same destination via a router in the middle. The role of the destination is that of the receiver in the incast definition. The ten nodes serve as the senders. In this setup, each of the ten nodes initiates a TCP connection to the destination as opposed to the other way around, but the desired behavior is exactly the same as incast.

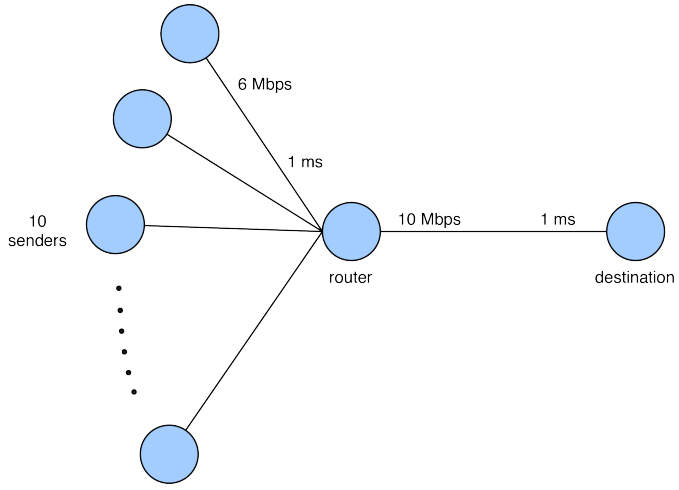

Figure 4

The ten nodes each sends a SYN packet to the destination. Once the destination sends the SYNACKs to these ten nodes, they each concurrently sends a short flow consisting of 10 packets to the destination. We set the *init_cwnd* to be 10 so that the entire flow can be transmitted in one round. We set up two scenarios to look at the benefit of using super-packets.

In the first scenario, the senders send normal packets through a regular droptail router in the middle that does not support super-packets. The router has a buffer size of 50. In the second scenario, the senders send super-packets of length 10 through a special router with super-packet support in the middle. The router has a buffer size of 50, a `t` of 5, and an `EPOCH` of 0.08 second.

The value of `EPOCH` in this experiment (and subsequent ones) is set to be big enough such that the state associated with a new super-packet does not expire pre-maturely inside the hash table and the bloom filter. In practice, `EPOCH` should be proportional to a small multiple of the round-trip time of the entire super-packet. This encourages the sender to send all packets of his super-packet as tightly as possible.

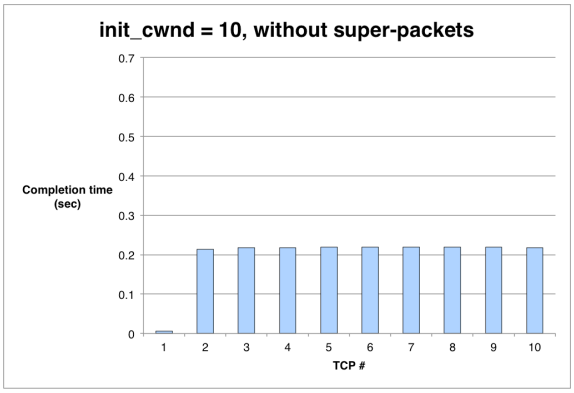

Figure 4a: TCP short flows and their completion time.

The result of this experiment is that the average latency of flows is 0.197 second when the senders do not use super-packets and 0.130 second when they do use super-packets of length 10. In other words, we see a 33.8% improvement in the average latency. The actual distribution of the latency of flows is shown in Figure 4a, 4b and 4c. As one can see from Figure 4a, when the senders do not use super-packets, the majority of the short flows experience packet loss that triggers a timeout and brings their *cwnd* down to 1 MSS. In fact 9 out of the 10 senders experience a timeout due to an insufficient number of duplicate ACKs.

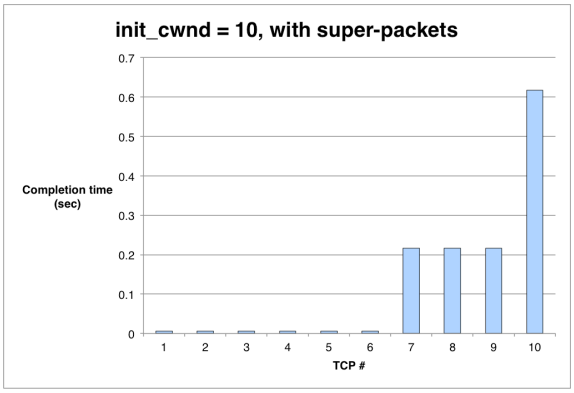

Figure 4b: TCP short flows as super-packets and their completion time.

On the other hand, when the senders use super-packets, the majority of the short flows get through the bottleneck link (shown in Figure 4b). Here, 6 out of 10 senders are able to successfully transmit their short flows in one round. There are 3 senders that experience one timeout and 1 very unlucky sender that experiences two timeouts. Despite this misfortune, the average latency is smaller because the majority of the flows are able to get through the bottleneck link on the first try.

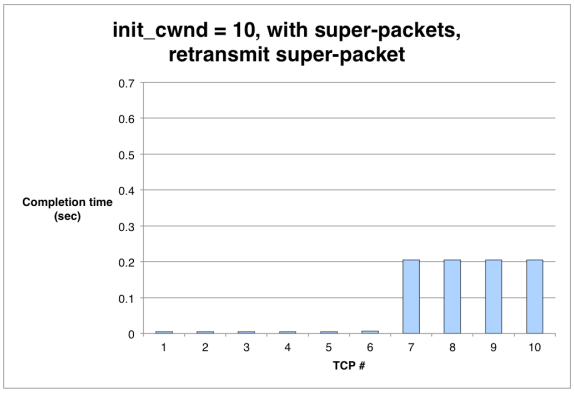

Figure 4c: TCP short flows as super-packets and their completion time if we use our ideal retransmission.

Furthermore, if the senders, whose super-packets are rejected the first time, do not fall back to the original TCP New Reno algorithm and instead send another super-packet with a *cwnd* of 10 after the timeout (i.e., if we use our ideal algorithm), then the average latency is 0.0857 second, or a 56% improvement. We obtained this result without having to implement our ideal algorithm by creating four additional senders that send out four brand new super-packets when the short flows labled 7 through 10 timeout for the first time. We tear down flows 7 through 10 before they are retransmitted as normal packets. In other words, we “replace” the four senders that timeout with four new

senders. Although a bit awkward, this mechanism produces the exact same result as that of our ideal algorithm.

4.3. Short Flows Competing with Long Flows

In order to see how super-packets would help short flows compete with long-running flows, we devised the topology shown in Figure 5. Host A sends ten long flows to the destination and fills the pipe completely (i.e., the two links connecting A to the router and the router to the destination are 100% utilized[*]). Host A's long flows each have an *init_cwnd* of 26 and an upper bound of 26 on the congestion window. Hosts B and C send short flows, each consisting of 10 packets, to the destination using a Poisson distribution. To minimize the effect of the SYN packet, we make the link between the router and the destination a bit faster than the link between each sender and the router, and instruct the router to always accept a SYN packet. Again, we set the *init_cwnd* of short flows to be 10 and set up two scenarios.

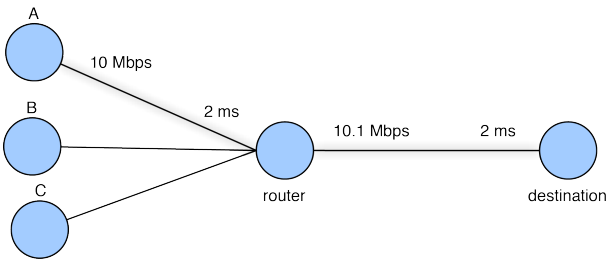

Figure 5

In the first scenario, the senders B and C send normal packets through a regular droptail router in the middle that does not support super-packets. The router has a buffer size of 20. In the second scenario, B and C send super-packets of length 10 through a special router with super-packet support in the middle. The router has a buffer size of 20, `t` of 10, and `EPOCH` of 0.08 second.

In the first scenario, the average latency of the flows is 0.0264 second. In the second scenario, the average latency of the flows is 0.0186 second. This is a 29.8% improvement.

4.4. Short Flows and the TCP Equation

One interesting side effect of using super-packets is that we are able to de-couple the maximum segment size (or MSS) from network constraints. It has been long known that the throughput of TCP is roughly proportional to $\frac{MSS}{RTT\sqrt{p}}$, where p is the drop rate. Indeed, the throughput depends on the MSS, which is limited by the maximum transfer unit (or MTU) since the former must be smaller than the latter. By using a super-packet, the sender is able to "enlarge" the MSS beyond the MTU.

To show this, we created the topology shown in Figure 6. There are 49 senders that simply send long flows. These long flows have an *init_cwnd* of 10 and an upper bound of 10 on the congestion window. One special sender sends a single long flow whose packets either have an enlarged MSS (scenario 1) or are wrapped inside super-packets of length 10 (scenario 2). In other words, for scenario 2, we break up the long flow into a sequence of super-packets.

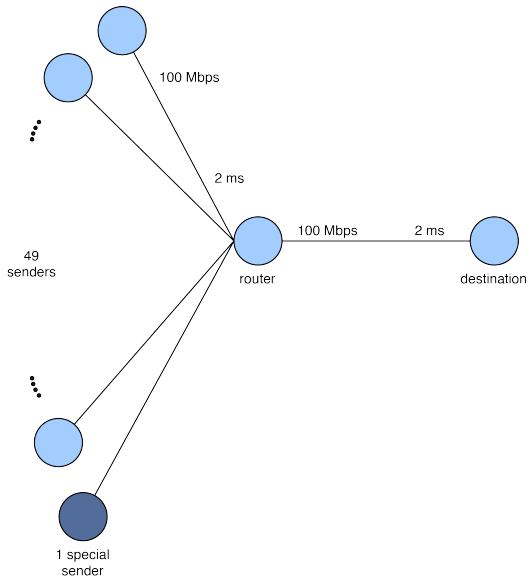

Figure 6

[*] The link from the router to the destination has a bit less than 100% utilization.

The experiment has a duration of 10 seconds. During the first second, the 49 senders begin their TCP connections to the destination. Starting at time 1, the special sender begins its TCP connection to the destination. We measure the throughput from when the special sender sends the second packet (after it receives the SYNACK).

In scenario 1, the special sender sends enlarged packets having 10 times the MSS of a normal packet (i.e., 14600 bytes) through a special router with super-packet support in the middle. The special sender's long flow has an *init_cwnd* of 5 and an upper bound of 5 on its congestion window. The router has a buffer size of 20 and `t` of 10.

In scenario 2, the special sender sends super-packets through the same special router with super-packet support in the middle. But this time, rather than sending enlarged packets, the special sender breaks up his long flow into a sequence of super-packets of length 10 (so a super-packet has a total MSS of 14600 bytes) and send them in succession. The router has a buffer size of 20 and `t` of 10 as before but since now we are dealing with actual super-packets, it must be mentioned that the router has an `EPOCH` of 0.08 second.

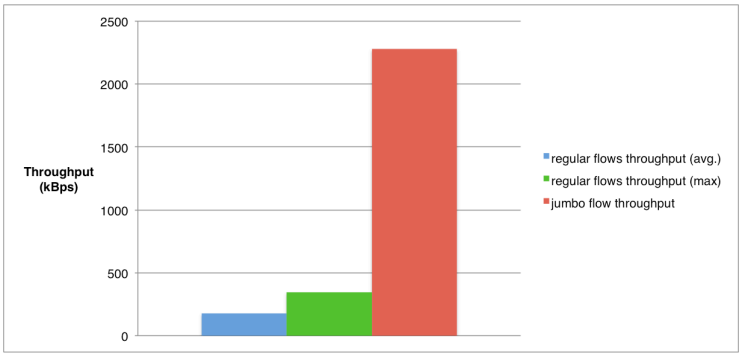

Figure 6a: Throughput of regular flows from the 49 senders vs. the throughput of the special sender sending enlarged packets. The flow that consists of these enlarged packets is termed "jumbo flow."

We measure and compare the throughput of the special sender in the two scenarios. The throughput graphs are shown in Figure 6a and 6b. The throughput of the special sender when it uses packets with 10 times the MSS of a normal packet is 2280 kBps, whereas the average throughput of the other 49 flows is 178 kBps (shown in Figure 6a). On the other hand, the throughput of the special sender when it uses super-packets of length 10 is 1577 kBps, whereas the average throughput of the other 49 flows is 223 kBps (shown in Figure 6b). As one can see, the throughput has gone up significantly in both cases.

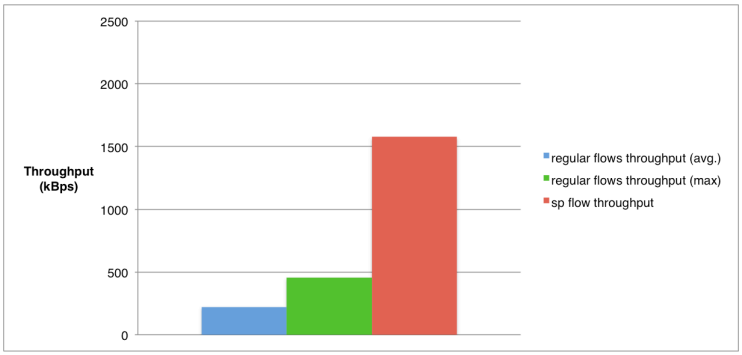

Figure 6b: Throughput of regular flows from the 49 senders vs. the throughput of the special sender sending super-packets. The flow that consists of these super-packets is termed "sp flow."

It should be noted that for various flows (regular flows, flows with enlarged MSS, flows with super-packets), a different set of values for the *init_cwnd* and the upper bound on the *cwnd* can change the outcome of any experiment. However, the benefits of using super-packets are definite regardless of the actual parameter values.

5. Related Work

It is widely known in the networking community that short flows have a disadvantage when it comes to obtaining a fair share of the network bandwidth. Google has proposed to increase the initial congestion window of TCP as a means to reduce latency for short flows and to give short flows a better chance to compete with long-running flows in the network for bandwidth [3]. This paper extends Google's proposal by introducing the notion of a super-packet. Super-packets help protect short flows from packet loss, allowing them to complete faster on average.

Nandita Dukkipati and Nick McKeown argue that flow-completion time is the right metric for congestion control [6]. Their work inspired us to focus on flow-completion time in our research. Specifically, we focused on improving the flow-completion time for short flows by protecting them from packet loss. Although packet loss does not significantly impact the flow-completion time percentage-wise for long flows, this is not the case for short flows. A few packet drops can result in the doubling or tripling of the flow-completion time for short flows because there are just not enough duplicate ACKs to trigger fast retransmit.

Forward Error Correction or FEC is another method used to improve the latency of flows by protecting them against packet loss. FEC encodes redundant information in the transmitted data to avoid costly retransmits or timeouts. The basic idea is that the sender sends n pieces of data, but the receiver only requires k, where $k < n$, pieces of data to reconstruct the original message. Therefore in the face of packet loss, as long as the receiver successfully receives k packets out of n, then there would be no need for retransmission. However, using FEC creates an overhead (i.e., an additional amount of data needs to be transmitted). Furthermore, FEC does not solve the incast problem. Therefore, we deem super-packets to be a more effective means of protecting flows, specifically short flows.

Jumbo frames are Ethernet frames with more than 1500 bytes of payload. In a sense, super-packets are like jumbo frames because they both bypass the MTU constraint of 1500 bytes. However, super-packets are better in that the packets that make up a super-packet do not need to be transmitted one immediately after another. This results in greater statistical multiplexing by the router. Also with super-packets, the length of a super-packet can be made arbitrarily large and is more flexible than that of a jumbo frame.

6. Future Research

In this section, we describe additional work that could be done in the area of super-packets for improving the performance of short flows in the Internet.

6.1. The Value of `t`

Recall that our algorithm uses a parameter `t`, which specifies when to stop accepting new super-packets. In our work, we used a static setting for `t`. Ideally, however, the value of `t` should be dynamically adjusted to accommodate a varying workload. For example, an exponential moving average that takes into account the number of recent super-packet arrivals may be appropriate. Alternatively, since we know the total number of outstanding packets of all super-packets that the router has accepted, we can use that information along with a deflation factor df in the range $(0, 1]$ to determine the value of `t` at any given time. That is, the value of `t` can make use of $df * \sum_{i=1}^{n} \Omega_i$, where n is the number of super-packets that the router has accepted and Ω_i is the `#_of_outstanding_packets` of super-packet i. The motivation behind the deflation factor is that some of the packets of a super-packet are forwarded before all of its packets arrive. Assume that the packets of a given super-packet are received by the router at a rate r_{in} and are forwarded at a rate r_{out}, where $r_{out} \leq r_{in}$, then we claim that $df = (1 - \frac{r_{out}}{r_{in}})$.

To see why, let $spLen$ be the length of the given super-packet. Then, it takes $t = \frac{spLen}{r_{in}}$ for the router to receive all the packets of the super-packet. During that time, the router forwards $t * r_{out} = spLen * \frac{r_{out}}{r_{in}}$ packets. Thus, when the last packet of the super-packet arrives, we have

$$spLen - spLen * \frac{r_{out}}{r_{in}} = spLen * \left(1 - \frac{r_{out}}{r_{in}}\right) = spLen * df$$

packets of the super-packet in the router queue.

Now, the value of df should be dynamically adjusted as well since the value of r_{out} depends on the workload.

Whatever method we choose, we have to ensure that there is enough incentive for users to choose super-packets over normal packets. Therefore, we can also think about an admission algorithm based on an incentive goal we want to achieve. For example, a good incentive would be to have the router reject a super-packet with roughly the same probability as the probability that it drops a normal packet. Based on that goal, one can try to devise an algorithm (or even adapt the algorithm we have) to meet that goal.

6.2. More Extensive Experiments

More experiments need to be done to solidify the theoretical foundation of super-packets. We would like to implement our ideal retransmission scheme of retransmitting the entire super-packet after a timeout if the super-packet is rejected. On the one hand, this circumvents the congestion avoidance of TCP. On the other hand, the all-or-nothing property of super-packets will act as a counter-force to this aggressive retransmission scheme.

6.3. Real Implementation

In order to move from theory to practice, we would like to do a complete implementation of super-packets in the Click modular router [8]. Ultimately, we would like to verify that super-packets could help reduce the latency of short flows in real world network traffic workloads.

7. Conclusion

Short flows in the network suffer severe performance degradation if one or more packets are dropped. An increased initial congestion window certainly allows for short flows to complete faster if there is no packet loss, but otherwise, it is not sufficient. In order to protect short flows from packet loss, we have developed the concept of a super-packet. A super-packet is a logical packet consisting of a finite number of consecutive packets in a flow. A super-packet-capable router makes a forwarding decision on the entire super-packet, either accepting or rejecting all packets of the super-packet. Super-packets help protect short flows from packet loss and help them compete better with long-running flows in the network. Interestingly, super-packets also enable hosts to decouple the maximum segment size from network constraints. In this paper, we have presented an efficient algorithm for super-packet admission in routers. The algorithm is keyed on the notion of the threshold value `t`. In practice, this value needs to take into account the workload experienced by the router running the admission algorithm. We have shown that the security implications of using super-packets mandate that any deployment should be carried out in a trusted environment like a datacenter. We ran a set of initial experiments, and they showed that super-packets have the potential to address the key shortcomings of short flows. By using super-packets, we can help protect short flows, reduce their latency on average, and ultimately make the Internet faster.

References

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, 2002.

[2] *http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html*. Cisco Visual Networking Index: Forecast and Methodology, 2011-2016.

[3] N. Dukkipati *et al*. An Argument for Increasing TCP's Initial Congestion Window. In *ACM SIGCOMM CCR*, 2010.

[4] S. Ramachandran. Web metrics: Size and number of resources. *https://developers.google.com/speed/articles/web-metrics*, 2010.

[5] Y. Chen *et al*. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Workshop on Research in Enterprise Networks*, 2009.

[6] N. Dukkipati and N. McKeown, Why Flow-Completion Time is the Right Metric for Congestion Control. In *ACM SIGCOMM Computer Communication Review*, Volume 36, Number 1, 2006.

[7] Network simulator 2: *http://www.isi.edu/nsnam/ns/*

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263-297, 2000.

Appendix A1

Variable	Definition
`buffSize`	Size of the buffer in number of packets (assume all packets are the same size for now).
`pktCnt`	Number of packets currently in the buffer.
`t`	Buffer threshold, where `buffSize - t` is larger than the maximum expected super-packet size.
`pkt`	Reference to packet.
`isFirst`	Function that takes a `pkt` and returns true if `pkt` is the first packet of a super-packet and false otherwise. Same value as `first` in `pkt`.
`regular`	Function that takes a `pkt` and returns true if `pkt` is a normal packet (not a packet belonging to a super-packet). False otherwise.
`accepted`	Function that takes a `pkt` and returns true if `pkt` is a non-first packet of a super-packet that the router has accepted. False otherwise.

On packet arrival:

```
if (regular(pkt)) {
  if (pktCnt < t) {
    pktCnt = pktCnt + 1;
    enqueue pkt;
  }
  else {
    drop pkt;
  }
}
else {
  if (isFirst(pkt)) {
    if (pktCnt < t) {
      install state;
      pktCnt = pktCnt + 1;
```

```
      enqueue pkt;
    }
    else {
      drop pkt;
    }
  }
  else {
    if (accepted(pkt)) {
      if (pktCnt < buffSize) {
        update state;
        pktCnt = pktCnt + 1;
        enqueue pkt;
      }
      else {
        drop pkt;
      }
    }
    else {
      drop pkt;
    }
  }
}
```

Appendix A2

Variable	Definition
time	Function that gives the current time.
prev	The starting time of the current epoch.
h1	Reference to first hash table.
h2	Reference to second hash table.
b1	Reference to first bloom filter.
b2	Reference to second bloom filter.

Constant	Definition
EPOCH	A predefined period of time.

On packet arrival:

```
if (time() – prev > EPOCH) {
  perform epoch shift;
}
```

On epoch shift:

```
h1 = h2;
h2 = new HashTable();

b1 = b2;
b2 = new BloomFilter();

prev = time();
```

Appendix A3

On state installation:

```
b2.add((src,dst,sid));
h2.update((src,dst), (len,len - 1));
```

On state update:

```
bool inb1 = b1.test((src,dst,sid));
bool inb2 = b2.test((src,dst,sid));
bool inh1 = h1.contains((src,dst));
bool inh2 = h2.contains((src,dst));

if (!inb1 && !inb2) {
  drop pkt;
  exit;
}
else if (inb1 && !inb2) {
  if (!inh1) {
    //false positive
    drop pkt;
    exit;
  }
  else if (h1.get((src,dst))[1] <= 0) {
    //#_of_outstanding_packets is 0
    drop pkt;
    exit;
  }
  else {
    h1.update((src,dst), (0,-1));
  }
}
else if (!inb1 && inb2) {
  if (!inh2) {
    //false positive
    drop pkt;
    exit;
  }
  else if (h2.get((src,dst))[1] <= 0) {
    //#_of_outstanding_packets is 0
    drop pkt;
    exit;
  }
  else {
    h2.update((src,dst), (0,-1));
  }
}
else {
  //false positive
  if (!inh1 && !inh2) {
    drop pkt;
```

```
    exit;
  }
  else if (inh1 && h1.get((src,dst))[1] > 0) {
    h1.update((src,dst), (0,-1));
  }
  else if (inh2 && h2.get((src,dst))[1] > 0) {
    h2.update((src,dst), (0,-1));
  }
  else {
    drop pkt;
    exit;
  }
}
```