CapStart: A Path Capacity Adaptive TCP Slow Start for High Speed Networks

Dirceu Cavendish, Kazumi Kumazoe, Masato Tsuru, Yuji Oie
Department of Computer Science and Electronics
Kyushu Institute of Technology
Fukuoka, Japan 810-0004
Email: {dirceu,kuma,tsuru,oie}@ndrc.kyutech.ac.jp

Mario Gerla
Computer Science Dept.
University of California, Los Angeles
Los Angeles, USA 90050
Email: gerla@cs.ucla.edu

Abstract—In this document, we introduce CapStart TCP, an adaptive Slow Start scheme that consistently achieves fast TCP file transfer times regardless of high speed network scenarios. Once the TCP session is established, we estimate TCP session path capacity scenario, and tune the transport protocol to deliver fast transfer times. We show that our capacity estimator works well even when the session is subjected to commercial Internet cross traffic. We demonstrate significant transaction performance improvements of CapStart TCP, of as much as three times faster completion times in transcontinental high speed network experiments for various capacity scenarios.

Keywords—high speed networks; TCP slow start; Packet retransmissions; capacity estimation; path bottleneck.

I. INTRODUCTION

It is well recognized that the exponential increase of TCP congestion window may lead to large window sizes in high bandwidth delay product networks, which can cause large packet losses and costly retransmission time outs. In fact, IETF RFC 3742 [4] Limited Slow Start introduces a size threshold over which the exponential increase of the congestion window is replaced by a linear increase so as to cope with the problem.

In this paper, we first characterize under which network path conditions the slow start congestion window exponential ramp-up causes large retransmissions. We then develop a path scenario classifier, which in real time identifies the nature of the network path on which the TCP session operates, and hence selects the ramp-up mechanism that is most suited for best performance. This manuscript is an extended version of the paper [1].

The material is organized as follows. Related work discussion is provided on Section II. Section III reviews TCP Reno slow start, as well as Limited Slow Start ramp-up mechanisms. We identify path capacity scenarios that are useful for characterizing Slow Start ramp-up performance, and introduce analytical models to predict their backlog build up behavior. Section IV describes a high speed long distance experimental network and Web Server setting environment used to characterize Slow Start performance from an Web Server/client application perspective. We discuss path capacity scenarios that affect TCP performance, and show that some scenarios favor Reno Slow Start, whereas others favor Limited Slow Start. We then introduce Cap Start, a hybrid Reno/LSS Slow Start scheme, which is able to consistently deliver fast transaction completion times regardless of network scenarios. We verify CapStart high performance via experiments over a research Internet network, where we control the cross traffic, as well as experiments on paths crossing the commercial Internet, where interfering traffic is unpredictable. In Section V, we address directions for future work.

II. RELATED WORK

Adverse effects of TCP congestion window ramp up have been long recognized. A possible approach to deal with the problem is to slow down the transmission of a cwnd worth of packets via timers, pacing the injection of packets into the network according to some indication of the session available bandwidth [9], [15]. However, this approach comes with a cost to the kernel, with additional timers to manage. Slower than the exponential ramp-up of Reno’s congestion window has been proposed in past research work, as a way to avoid retransmissions. [7] proposes a ramp up scheme that is “less than exponential”, controlled by rtt measurements, whereas [2] switches between an exponential growth to a linear one, similar to congestion avoidance, if the current window size is larger than a value calculated as “the maximum window size sustainable at steady state in the bit pipe”. We argue that rtt measurement based cwnd adjustment is troublesome due to RTT high variability on realistic network scenarios. One way to avoid retransmissions is via tuning of the “ssthresh” parameter. [5] sets the ssthresh to an estimate of the session available bandwidth taken by observing the sequence of ACK arrival epochs. Another ssthresh adjustment scheme based on available bandwidth estimate derived from the ack stream is proposed by [18]. Although this approach prevents massive packet losses during SS to CA transition if accurate bandwidth estimation is achieved, it may lead to an early departure from SS, affecting the session throughput performance, especially in high speed network scenarios. These works tend to increase transaction completion time, as they decrease the opening of the control window. In contrast, our approach switches between two
well proven Slow Start algorithms, using a robust capacity classifier.

[17] addresses the reduction of Web Server transaction completion times by altering the way the Kernel serves multiple TCP sessions. Their work claims that by giving preference to soon to finish TCP sessions the number of simultaneous sessions held open decreases on average, bringing the average response time of each session down. Since our proposal relies on TCP protocol changes, it can be seen as a complementary approach to server-based improvements such as theirs.

III. SLOW START MECHANISMS AND PATH CAPACITY

We now address slow start behavior in specific path capacity scenarios.

A. Slow Start & Limited Slow Start

Ramping up quickly a TCP session is necessary in order to take advantage of network available bandwidth early in the life time of TCP sessions. Applications with short lived flows, such as Web Server transactions, are becoming ever more common, so a quick ramp up of the cwnd by TCP Slow Start mechanism is more than justified. The original slow start algorithm of Jacobson [10] starts with a congestion window - cwnd - size of one segment, and for each acknowledgement received, it increases the cwnd size by one extra segment. This logic causes the cwnd to double its size at every round trip time (RTT) of the TCP session, causing an exponential increase of the number of injected segments into the network per RTT. We use packets and segments interchangeably in this document.

The exponential growth of the cwnd may cause large segment losses for certain network scenarios. Limited Slow Start [4] aims at mitigating this effect by limiting the exponential growth up to a max_ssthresh parameter value, over which the cwnd is increased by a fraction of the current cwnd value. This effectively replaces the exponential growth with a less than linear cwnd increase once the congestion window size reaches max_ssthresh value.

B. Path Capacity Scenarios

We classify the various network scenarios into two categories, with respect to the relation between the TCP sender interface speed and the path bottleneck speed: capacity expansion, and capacity reduction. The two categories are illustrated in Fig. 1.

In what follows, we present an analysis of the dependency of the bottleneck queue build up with the congestion window cwnd size, for these two capacity scenarios, which helps explain experimental results presented later in the paper. Let C_{if} and C_i be the TCP sender interface capacity and router i outgoing interface capacity, respectively. A capacity expansion path scenario satisfies C_{if} < C_i, ∀i, whereas a capacity reduction path scenario satisfies C_{if} ≥ C_i, ∀i. We make the following simplifying assumptions:

- **Persistent sender**: Idle periods drain routers’ queues due to factors other than cwnd behavior. Persistent sender is then a worst case assumption.
- **Infinity router queue sizes**: Packet losses cause TCP to exit slow start. We focus on queue build ups during slow start.
- **Simple ack receive behavior**: We assume no delayed acks or ack behaviors other than immediate acking of just received segments, for simplicity. Delayed ack allows the receiver to impact the evolution of cwnd. We can extend the analysis to the case of k segments being acked together. Selective ack does not impact the analysis, as we assume infinite buffers, so packet loss does not occur.
- **No cross traffic**: We wish to isolate queue build ups caused by the cwnd size behavior from other factors, so that we can better understand the role of the sender on this build up. The analysis can be extended to the case of bottleneck cross traffic.

Fig. 2 depicts path capacity expansion and reduction scenarios.

![Figure 1: Types of path capacity scenarios](image)

![Figure 2: Path capacity scenarios’ analysis](image)
Fig. 2 a) shows router $i$ queue data arrival and departure patterns. Arrival segments are $MSS/C_{if}$ apart (segment inter-arrival time) because the sender interface speed is the slowest interface in the path. The arrival process shown assumes an intermediate router $h$ with higher capacity than the sender interface, hence the compression of the packet arrival (not inter-arrival) time. In case there is no router with capacity larger than $C_{if}$, there is no gap between segments arriving at router $i$. The interdeparture time of segments from router $i$ is driven by the sender interface capacity, and so does the ack segment arrival process back to the sender. The total number of outstanding segments in the path is equal to the $cwnd$ size $n$. Under these conditions, router $i$ goes idle for a fraction of time for every packet transmitted, hence there is no queue build up. If $cwnd$ is large enough, the entire path is “filled” by segments that are $MSS/C_{if}$ apart, beyond which the number of in flight segments become independent of $cwnd$. Beyond this point, the throughput is constrained by the sender interface capacity, and no longer driven by the $cwnd$ size. This point is: $cwnd_{fill} = RTT \times C_{if}/MSS$.

Fig. 2 b) shows bottleneck router $bn$ queue data arrival and departure patterns. The queue departure process consists of back-to-back segments that are $MSS/C_{bn}$ apart, which drives the ack segment arrival process at the sender, and in its turn drives the segment injection process into the network. Because each ack generates the injection of two segments into the network at $C_{if}$ speed, there is a segment pair pattern, each segment $MSS/C_{if}$ apart from the other in a pair. Each segment pair is $MSS/C_{bn}$ apart from the next pair, because this distance is driven by the bottleneck departure process. Within an RTT period, there is a number $p$ of “segment pairs” before the bottleneck, and a number $q$ of segments after the bottleneck per $MSS/C_{bn}$ period, such that $cwnd = 2p + q$. As for each segment departure from the bottleneck router two segments arrive, there is a queue build up of $p$ segments. If the path RTT is long enough to have at least $p \times MSS/C_{bn}$ idle periods, this build up is completely drained out within one RTT period. The condition for the complete draining of bottleneck backlog is then $RTT \geq (2p + q)MSS/C_{bn}$. The maximum $cwnd$ size for which the bottleneck backlog is completely drained within one RTT results to be:

$$cwnd_{drain} = \frac{RTT \times C_{bn}}{MSS}$$  (1)

Beyond this size, $cwnd$ allows $cwnd - cwnd_{drain}$ extra segments within one RTT into the path that will not be drained from the bottleneck, because there are fewer idle $MSS/C_{bn}$ periods than periods with a pair of segments, as shown in Fig. 2 b). Finally, the worst case queuing build up occurs when $cwnd$ is large enough to fill the path with $MSS/C_{bn}$ periods, each of which having a pair of segments, or:

$$cwnd_{maxQ} = \frac{2RTT \times C_{bn}}{MSS}$$  (2)

Half of the segments allowed by $cwnd_{maxQ}$ accumulate in the router bottleneck. For $cwnd$ sizes larger than $cwnd_{maxQ}$, only $RTT C_{bn}/MSS$ segments accrue to the bottleneck build up per RTT. Table I summarizes our backlog analysis for path capacity reduction as a function of $cwnd$ size. One can see that queue build up is a function of the bottleneck capacity $C_{bn}$, but is not related with the sender interface capacity $C_{if}$, as long as $C_{if} \geq C_{bn}$. In other words, the scenarios 1G-100M and 10G-100M generate similar queue build up. This is a consequence of the bottleneck capacity driving the injection of segments at the sender. In addition, queue build up is proportional to the path RTT: the larger the RTT, the more build up is expected. Finally, the analysis can be easily extended to consider bottleneck cross traffic by defining routers’ available bandwidth $BW_{i}$ and replacing $C_{bn}$ with $BW_{bn}$, where $bn$ is the router with smallest available bandwidth. In this case, the results of Table I can be interpreted as for scenarios where the bottleneck router does not change and its available bandwidth remains constant within one RTT.

<table>
<thead>
<tr>
<th>$cwnd$</th>
<th>Backlog(per RTT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$RTT \times C_{bn}/MSS$</td>
<td>$cwnd \leq RTT \times C_{bn}/MSS$</td>
</tr>
<tr>
<td>$RTT \times C_{bn}/MSS$</td>
<td>$cwnd \leq 2RTT \times C_{bn}/MSS$</td>
</tr>
<tr>
<td>$RTT \times C_{bn}/MSS$</td>
<td>$cwnd &gt; 2RTT \times C_{bn}/MSS$</td>
</tr>
</tbody>
</table>

Table I: Capacity Reduction Backlog (segs/RTT) vs $cwnd$ size(segs)

As Table I predicts queue build ups for capacity reduction scenarios for large $cwnd$ sizes, less aggressive $cwnd$ increase schemes result in smaller queue build ups and better performance during slow start. In the experimental evaluation section of this paper, we show that TCP sessions facing capacity expansion path scenarios achieve better performance when using Jacobson’s exponential Slow Start, whereas sessions facing capacity reduction path scenarios achieve better performance when using Limited Slow Start. This motivates us to seek a mechanism to detect whether the TCP session faces one or the other path scenarios.

C. Path Capacity Estimation

In order to determine whether the session faces a capacity expansion or capacity reduction scenario, besides knowing its own interface speed, the sender needs to know its path capacity. The network path capacity estimation method of our choice is based on packet pair dispersion [12] techniques. The idea is to measure the dispersion of the delay of packet pairs sent back to back. If both probing packets of size MSS of a packet pair sample do not suffer queueing delay, and
the dispersion between them is $d$, the slowest link capacity of the TCP session path can be estimated as:

$$\hat{C} = \frac{MSS}{d}$$

We estimate $\hat{C}$ by measuring the dispersion of rtt acks of segments that were sent back to back, at source interface link speed. Let $rtt_1$ and $rtt_2$ be the round trip time of segment 1 and 2 of a segment pair sample $i$. Our strategy is to seek the sample $j$ whose $rtt_1$ and $rtt_2$ are both minimum values among all samples ever seen by the session, which minimizes queueing delays on both segments.

We may write $rtt_1 = PD + q_1 + S_1$ and $rtt_2 = PD + q_2 + S_2$, where $PD$ is the propagation delay component of each segment rtt, $q_{1,2}$ is the queueing delay component, and $S_{1,2}$ is the service time component of each segment. We seek to find a segment pair $j$ for which $q_{1,2} = 0$. If either segment has a queue delay component, the dispersion measured may expand or contract, leading to under or over estimation of the slowest link capacity, respectively. A reasonable approach, taken by [12], is to keep running values of $\min_i(rtt_1)$ and $\min_i(rtt_2)$, and elect the segment pair sample $j$ for which $rtt_1 + rtt_2 = \min_i(rtt_1) + \min_i(rtt_2)$. [12] assumes a minimum number of samples, before a candidate sample is elected. We take an approach slightly different. Recalling that the objective is to use a segment pair sample for which $q_{1,2} = 0$, a conservative strategy is to select the sample $j$ which matches $\min_i(rtt_j)$ among all samples ever seen. This ensures that we select the sample with minimum $q_i$. If there is more than one such a sample, we select the one with least $rtt_2$, i.e., the one with minimum $q_2$ among the samples with minimum $q_1$.

A candidate sample, candRtt1, candRtt2, is kept at all times with the dispersion of the best capacity sample. At slow start, a sample is released at every rtt, as a guard time is needed between samples to avoid self interference in the estimation. Upon ack reception, sampleRtt1 and sampleRtt2 values are computed, and the sample is compared with the candidate pair so far. Fig. 3 illustrates the sample selection algorithm.

Accuracy of the packet pair dispersion technique depends on delays incurred by each of the segments of the packet pair [12]. If both segments of a capacity estimation segment pair do not experience queueing delays, the capacity estimation is precise. When they do experience queueing delays, the capacity estimated can be greater or smaller than the real bottleneck capacity, as illustrated in Fig. 4. If both segments experience exactly the same queueing delays, a rather rare event, the capacity measured is the same as the real bottleneck capacity. If the first segment experiences a larger queueing delay than the second one, the gap between the segments decreases, and the bottleneck capacity is over estimated. Conversely, if the second segment experiences a larger queueing delay than the first one, the gap between the segments increases, and the bottleneck capacity is under estimated. The algorithm of Fig. 3 is conservative, since it selects sample pairs whose first segment has experienced smallest delays, favoring capacity under estimation.

Figure 3: Capacity estimation flowchart

![Figure 3: Capacity estimation flowchart](image)

Figure 4: Capacity estimation accuracy

![Figure 4: Capacity estimation accuracy](image)

During congestion avoidance, a sample is released every time a current one is too old (acks have SNs larger than the current sample pair), provided that the window adjustment allows for at least two segments to be sent. So the capacity probing pacing remains one per rtt. In addition, few segment reception scenarios need to be handled. A duplicate ack within the range of sequence numbers expected indicates a possible segment loss of the second segment of a pair, or the loss of its ack. In either way, the sample is discarded, as its rtt values may be inflated by retransmissions. Selective acknowledgements [14] also interferes with the capacity estimator, so the sample is discarded. Finally, a duplicate SACK [3] is an indication of lost acks, so the sample is also discarded.

IV. HIGH SPEED EXPERIMENTAL EVALUATIONS

In this section, we evaluate the slow start impact on applications running on a high speed network infrastructure, and tune it to achieve high transaction performances. We
use httperf [6], a HTTP performance measurement tool, to characterize file transfer application transaction completion time over a high speed network infrastructure. Httperf is a load tool that measures web server performance. Httperf is able to produce a variety of web server workload requests, measuring each transaction statistics. In addition, we use netstat tool to track events at the TCP level.

We have conducted experiments on an intercontinental high bandwidth delay product network, with path capacities ranging from 100Mbps to 10Gbps. The experimental network has servers with PCI-133 buses and 10-Gbps network interface cards [11]. The access router has interfaces with 250 packet buffers.

A. Characterizing transaction performance

![Image 1](image1.png)

**Figure 5:** Single Server Experimental Network Scenarios

The experimental network used for transaction performance characterization is depicted in Fig. 5. In order to evaluate the impact of transport protocols on applications performance, an HTTP server is used at the Kitakyushu/Japan end-point, for which we control the TCP slow start used to transfer 1Gbyte of data per HTTP transaction. An httperf [6] traffic generator is used at Chicago/USA and Tokyo/Japan end-points to generate requests towards the server in Kitakyushu. We use both httperf transaction measurement statistics, as well as netstat results at the Kitakyushu server site to generate performance data, such as http average transaction completion time, number of segments retransmitted and lost, as well as cwnd time evolution. We report on several path scenarios: i) expansion scenarios: 100M-100M, 100M-1G, 100M-10G, 1G-1G, 1G-10G; ii) reduction scenarios: 10G-1G, 10G-100M, 1G-100M. For each scenario, we ran Jacobson’s slow start (Reno), as well as Limited Slow Start, for capacity expansion scenarios. First we notice that for short RTT scenarios, there is little difference in performance between Reno and Limited Slow Start schemes. For these scenarios, there is no retransmissions, which is consistent with the no queue build up prediction of our analysis. For large RTTs, however, when the network interface is the path bottleneck, the exponential slow start delivers fastest transaction completion time. In addition, as the server network interface speed increases (1G-1G, 1G-10G scenarios), Limited Slow Start transaction completion time is significantly slower than Reno, up to five times slower. This is again consistent with our analytical predictions of no queue build up for capacity expansion scenarios - in the absence of queue build up and retransmissions, the faster cwnd opening scheme delivers higher performance. Fig. 7 reports slow start performance on capacity reduction scenarios. In capacity reduction scenarios of large RTT, LSS slower ramp-up results in significant reduction in transaction completion time, up to three and a half times faster than Reno. The cause for this is tracked to the amount of segments lost, with costly retransmissions. Notice how similar 10G – 100M and 1G – 100M performances are, which is consistent with the queue build up being independent of the server interface capacity predicted by our analytical model. It is clear that an efficient slow start should operate as Reno SS on path capacity expansion scenarios, and as Limited Slow Start at path capacity reduction scenarios.

![Image 2](image2.png)

**Figure 6:** Transaction Completion Time(sec): Cap. Expansion

![Image 3](image3.png)

**Figure 7:** Transaction Completion Time(sec): Cap. Reduction
B. Building a capacity scenario classifier

A naive path capacity scenario classifier consists in measuring the bottleneck capacity and comparing it with the network interface card capacity. The problem is that the capacity estimated is at most the server network interface speed. Moreover, the server interface card typically delivers less throughput than its nominal capacity. In order to best devise a classification scheme, we have implemented the capacity estimation algorithm of Fig. 3 in each slow start TCP code, and tracked its performance. Figs. 8, 9, 10, and 11 report on the estimation process for path capacity expansion and reduction scenarios, respectively. In each chart, the x-axis tracks the capacity estimation updating process, so it counts the number of samples used to adjust the capacity estimation for each of 5 trials. LSS typically has a larger number of samples for capacity expansion scenarios, whereas the converse is true for capacity reduction scenarios. In addition, trials may not present the same number of “accepted samples”, that is, the same number of capacity estimation updates. This is because the acceptance of a new sample is conditioned to the “quality” of the best sample “seen” so far by the TCP session. If by chance an early sample happens to have both segments with queueing delay zero, for example, no further sample is ever accepted during the lifetime of that session, as the previous sample has the best quality one can hope for.

We adopt the following classification logic. If the capacity measured is less than 10% of the nominal server network interface card, we declare the path scenario to be a capacity reduction one, and hence use Limited Slow Start; Otherwise, we declare the path scenario to be capacity expansion, and use the exponential increase Reno Slow Start. We believe this to be an accurate classification scheme not only due to the capacity data collected, but also because Ethernet link speeds change by a factor of 10. So, any detected reduction by a factor greater than 10 is likely to be a capacity reduction scenario.

Miss-classification may occur in two ways:

- In capacity expansion scenarios, the capacity estimator outputs a value less than 10% of the server network interface speed. Only a very low performance network interface card could deliver such poor interface throughput. We have not encountered such a case in our experiments.
- In capacity reduction scenarios, the capacity estimator outputs a value that is more than 10% of the server network interface speed. Given that in this case the bottleneck speed is at least one order of magnitude smaller than the nominal server interface speed, the capacity estimator has to overestimate the bottleneck capacity by at least 110%. We have not encountered such a case either.

The accuracy of the path capacity classifier was then 100%. We have implemented the path capacity classifier on a Linux TCP stack. The current implementation of the path capacity classifier is accurate to up to 1 Gbps bottleneck speeds, due to a limitation on Linux kernel timer accuracy. We also had to modify the TCP socket to include network interface card speed used by the session associated with the socket, not originally available in the socket. Our capacity estimator implementation for Linux is limited to path capacities up to 1Gbps, due to timer accuracy issues. 10Gbps interfaces require sub-micro second accuracy for 1500 byte segments.

C. Adaptive Slow Start Experiments - No cross traffic

Figure 12 reports on the transaction response times of CapStart, a hybrid Reno & Limited Slow Start algorithm using our proposed path capacity scenario classifier for large RTT scenarios (Kitakyushu-Chicago: 180msecs) and no...
cross traffic. The algorithm initiates as Reno, and switches to LSS if its path capacity classifier detects a capacity reduction scenario. The absence of cross traffic ensures that the algorithm does not leave slow start phase at an arbitrary time, due to random packet loss. We can see that its average transaction completion time is the least among all slow start algorithms for all path capacity scenarios. For the 1G-100M experiment, CapStart better behavior than LSS is because the algorithm starts as Reno, hence leading to more throughput than LSS before switching to LSS behavior. This can be verified by cwnd traces, such as the one of Fig. 13. For capacity expansion scenario, cwnd evolution of CapStart follows closely Reno (SS), whereas for capacity reduction scenario CapStart follows Reno in the first seconds of the session, and then parallels the increase of the LSS curve. Similarity between results for scenarios 10G – 100M and 1G – 100M is consistent with our analytical results.

Next, we introduce 20% of bottleneck capacity worth of traffic through the bottleneck link, in order to evaluate the impact of cross traffic on the path capacity classifier, and hence on CapStart performance. Firstly, capacity estimation results similar to the ones reported in Figs. 10 and 11 were obtained, demonstrating that the estimation accuracy did not get significantly affected by UDP traffic. It is not surprising, therefore, that transaction completion time performance, reported in Fig. 14 is qualitatively similar to the scenarios with no UDP traffic, except for one type of path capacity scenario. For capacity scenarios where the server interface speed and bottleneck speed are the same, extra UDP traffic has the qualitative effect of turning a capacity expansion scenario into a capacity reduction one. This is consistent with the extension of our analytical model to cross traffic, where available bandwidth rather than bottleneck capacity is considered. Since the bottleneck capacity estimation is unaffected by the extra UDP traffic, the classifier still decides for a capacity expansion scenario, causing a worse performance than Reno. For all other path capacity scenarios, performance is unaffected by the extra traffic, as the amount of extra traffic is not enough to turn a capacity expansion scenario into a capacity reduction one.

Finally, Fig. 16 presents multi-server experimental results for the topology shown in Fig. 15. Access router interfaces have 64 packet buffers. Two flows originating and terminating at different server/client share a common bottleneck. Flow 1 starts transmission slightly earlier than flow 2. For path capacity expansion scenarios, CapStart behaves like Reno, and does not impact the behavior of the other TCP protocol. For path capacity reduction scenarios, Reno and CapStart favor the flow that initiates first, whereas LSS does not. In addition, CapStart behaves similarly to Reno for most scenarios, except when sharing the bottleneck with Reno. In that case, even though CapStart flow 1 starts first, Reno flow 2 finishes faster, showing that Reno is more aggressive than CapStart. In general, CapStart does not incur in fairness issues when coexisting with other TCP protocols, since it either uses the exact same slow start mechanism as other TCPs, or it uses the Limited Slow Start, which slows down

![Figure 11: Capacity Estimation(Gbps): 1G Cap. Reduction](image1)

![Figure 12: Kitakyushu-Chicago Transaction Completion Time(sec): Cap Start, Reno (SS), Limited Slow Start (LSS)](image2)

![Figure 13: Kitakyushu-Chicago Cwnd Dynamics: Cap Start, Reno (SS), Limited Slow Start (LSS)](image3)

![Figure 14: Kitakyushu-Chicago Transaction Completion Time(sec) with UDP](image4)
congestion window ramp up as compared with regular Slow Start, injecting less traffic than other protocols.

Figure 15: Two Server Experimental Network Scenarios

![Diagram showing two server experimental network scenarios](image)

Figure 16: Kitakyushu-Chicago Transaction Completion Time(sec) with two servers: Cap Start, Reno (SS), Limited Slow Start (LSS)

![Bar charts showing transaction completion time with different protocols](image)

D. Adaptive Slow Start Experiments - Internet Path Scenario

The previous experiments were executed in a clean end-to-end path, except for well controlled UDP traffic case. We now present some results of CapStart performance on a real Internet path scenario, depicted in Fig. 17. Figures 18 and 19 shows the cwnd dynamics of Reno, Limited, and CapStart slow start schemes for two trials. Notice that each protocol is affected randomly by packet losses due to cross traffic. For 18 run, limited slow start is not hit by packet losses, and hence is able to finish 1GByte transfer quickly (the end of the lines show completion time). Notice that LSS and CapStart protocols exhibit less retransmissions than Reno, which is evidence of a smoother slow start mechanism. For 19 run, Reno is not hit by packet losses, and hence is able to finish 1GByte transfer more quickly than the other protocols, despite LSS and CapStart having less retransmissions in the beginning of their sessions. Since transfer time is heavily dependent on the specific packet loss pattern experienced by each protocol, a completion time comparison is not meaningful on a real Internet path scenario. Notice, however, that CapStart still delivers best performance consistently across all capacity path scenarios, under the same cross traffic.

Figure 17: Kitakyushu-Stuttgart Internet path scenario

![Diagram showing Kitakyushu-Stuttgart Internet path scenario](image)

Figure 18: Control window/retransmissions dynamics: Kitakyushu-Stuttgart Internet path trial 1

![Graphs showing control window/retransmissions dynamics](image)

Figure 19: Control window/retransmissions dynamics: Kitakyushu-Stuttgart Internet path trial 2

We now investigate CapStart’s accuracy in detecting path capacity reduction scenarios, as capacity estimation accuracy can be measured regardless of random packet losses. We have run CapStart on 1G-100M path capacity scenario, and collected data about whether capacity reduction was detected and how soon. Out of 100 trials, CapStart was able to detect path capacity reduction on 90 instances. Figures 20 and 21 show how fast CapStart is able to detect capacity reduction for two client access speeds of 100Mbps and 1Gbps, respectively, and hence switch to a conservative cwnd increase. The x-axis shows the number of RTTs taken for detection, whereas the y-axis show the number of trials on a particular detection time. It is quite remarkable that CapStart is able to detect path capacity reduction very early in the session lifetime.

E. Lessons Learned

Our experimental investigation has shown the following:
V. CONCLUSION

We have characterized path capacity scenarios that substantially affect TCP sessions from a transaction completion time application performance standpoint. We have shown that, for high bandwidth delay product networks, faster network interface cards may not result in better web server performance with Reno protocol. For path capacity reduction scenarios, Limited Slow Start delivers faster transaction completion times, whereas for path capacity expansion scenarios, Reno Slow Start delivers superior performance. Finally, we have proposed a path capacity scenario classifier as an automated way to merge Reno and Limited Slow Start into a robust Slow Start algorithm that consistently delivers fast transaction completion times.

This work can be extended into considering estimators for specific types of networks, with various characteristics, such as satellite [13] and various wireless networks (e.g., [16], as well as specific applications, such as VoIP [8]. Depending on the specific case, path estimators can be used not only to slow down the ramp up of the cwnd, but also to speed it up, in case of lightly loaded and large rtt networks. The ultimate goal is to control segment losses while satisfying applications’ transmission needs. As evidenced by real Internet path experimental results presented, reducing retransmissions during slow start only does not ensure better TCP performance, because the slow start phase of a session can be arbitrarily short, depending on how soon a packet loss is experienced. We are currently designing a new congestion avoidance algorithm to reduce retransmissions, improving total transfer time.

ACKNOWLEDGMENT

Research partially supported by the Japan Society for the Promotion of Science, Grant-in-Aid No. 18100001.

REFERENCES


