Towards a SPDY'ier Mobile Web?

Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, K.K. Ramakrishnan
AT&T Labs – Research
One AT&T Way, Bedminster, NJ, 07921
{erman,gvijay,rjana,kkrama}@research.att.com

ABSTRACT

Despite its widespread adoption and popularity, the Hypertext Transfer Protocol (HTTP) suffers from fundamental performance limitations. SPDY, a recently proposed alternative to HTTP, tries to address many of the limitations of HTTP (e.g., multiple connections, setup latency). With cellular networks fast becoming the communication channel of choice, we perform a detailed measurement study to understand the benefits of using SPDY over cellular networks. Through careful measurements conducted over four months, we provide a detailed analysis of the performance of HTTP and SPDY, how they interact with the various layers, and their implications on web design. Our results show that unlike in wired and 802.11 networks, SPDY does not clearly outperform HTTP over cellular networks. We identify, as the underlying cause, a lack of harmony between how TCP and cellular networks interact. In particular, the performance of most TCP implementations is impacted by their implicit assumption that the network round-trip latency does not change after an idle period, which is typically not the case in cellular networks. This causes spurious retransmissions and degraded throughput for both HTTP and SPDY. We conclude that a viable solution has to account for these unique cross-layer dependencies to achieve improved performance over cellular networks.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Applications*; C.4 [**Performance of Systems**]: Measurement techniques

Keywords

SPDY, Cellular Networks, Mobile Web

1. INTRODUCTION

As the speed and availability of cellular networks grows, they are rapidly becoming the access network of choice. Despite the plethora of 'apps', web access remains one of the most important uses of the mobile internet. It is therefore critical that the performance of the cellular data network be tuned optimally for mobile web access.

The Hypertext Transfer Protocol (HTTP) is the key building block of the web. Its simplicity and widespread support has catapulted it into being adopted as the nearly 'universal' application protocol, such that it is being considered the narrow waist of the future internet [11]. Yet, despite its success, HTTP suffers from fundamental limitations, many of which arise from the use of TCP as its transport layer protocol. It is well-established that TCP works best if a session is long lived and/or exchanges a lot of data. This is because TCP gradually ramps up the load and takes time to adjust to the available network capacity. Since HTTP connections are typically short and exchange small objects, TCP does not have sufficient time to utilize the full network capacity. This is particularly exacerbated in cellular networks where high latencies (hundreds of milliseconds are not unheard off [18]) and packet loss in the radio access network is common. These are widely known to be factors that impair TCP's performance.

SPDY [7] is a recently proposed protocol aimed at addressing many of the inefficiencies with HTTP. SPDY uses fewer TCP connections by opening one connection per domain. Multiple data streams are multiplexed over this single TCP connection for efficiency. SPDY supports multiple outstanding requests from the client over a single connection. SDPY servers transfer higher priority resources faster than low priority resources. Finally, by using header compression, SPDY reduces the amount of redundant header information each time a new page is requested. Experiments show that SPDY reduces page load time by as much as 64% on wired networks and estimate as much as 23% improvement on cellular networks (based on an emulation using Dummynet) [7].

In this paper, we perform a detailed and systematic measurement study on real-world production cellular networks to understand the benefits of using SPDY. Since most websites do not support SPDY – only about 0.9% of all websites use SPDY [15] – we deployed a SPDY proxy that functions as an intermediary between the mobile devices and web servers. We ran detailed field measurements using 20 popular web pages. These were performed across a four month span to account for the variability in the production cellular network. Each of the measurements was instrumented and set up to account for and minimize factors that can bias the results (e.g., cellular handoffs).

CoNEXT'13, December 9–12, 2013, Santa Barbara, California, USA.
Copyright 2013 ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535399.

Our main observation from the experiments is that, unlike in wired and 802.11 WiFi networks, SPDY *does not* outperform HTTP. Most importantly, we see that the interaction between TCP and the cellular network has the most impact on performance. We uncover a fundamental flaw in TCP implementations where they do not account for the high variability in the latency when the radio transitions from idle to active. Such latency variability is common in cellular networks due to the use of a radio resource state machine. The TCP Round-Trip Time (RTT) estimate and thus the time out value is incorrect (significantly under-estimated) after an idle period, triggering spurious retransmissions and thus lower throughput.

The TCP connection and the cellular radio connection for the end-device becomes idle because of users' web browsing patterns (with a "think time" between pages [9]) and how websites exchange data. Since SPDY uses a single long lived connection, the TCP parameter settings at the end of a download from one web site is carried over to the next site accessed by the user. HTTP is less affected by this because of its use of parallel connections (isolates impact to a subset of active connections) and because the connections are short lived (isolates impact going across web sites). We make the case that a viable solution has to account for these unique cross-layer dependencies to achieve improved performance of both HTTP and SPDY over a cellular network.

The main contributions of this paper include:

- We conduct a systematic and detailed study over more than four months on the performance of HTTP and SPDY. We show that SPDY and HTTP perform similarly over cellular networks.

- We show that the interaction between the cellular network and TCP needs further optimization. In particular, we show that the RTT estimate, and thus the retransmission time-out computation in TCP is incongruous with how the cellular network radio state machine functions.

- We also show that the design of web sites, where data is requested periodically, also triggers TCP timeouts. We also show that there exist dependencies in web pages today that prevent the browser from fully utilizing SPDY's capabilities.

2. BACKGROUND

We present a brief background on how HTTP and SPDY protocols work in this section. We use the example in Figure 1 to aid our description.

2.1 The HTTP Protocol

The Hypertext Transfer Protocol (HTTP) is a stateless, application-layer protocol for transmitting web documents. It uses TCP as its underlying transport protocol. Figure 1(a) shows an example web page which consists of the main HTML page and four objects referred in that page. When requesting the document, a browser goes through the typical TCP 3-Way handshake as depicted in Figures 1(b) and (c). Upon receiving the main document, the browser parses the document and identifies the next set of objects needed for displaying the page. In this example there are four more objects that need to be downloaded.

With the original versions of HTTP, a single object was downloaded per connection. HTTP version 1.1 introduced the notion of persistent connections that have the ability to reuse established TCP connections for subsequent requests and the concept of pipelining. With persistence, objects are requested sequentially over a connection as shown in Figure 1(b). Objects are not requested until the previous response has completed. However, this introduces the problem of *head-of-line* (HOL) blocking where subsequent requests get significantly delayed in waiting for the current response to come back. Browsers attempt to minimize the impact of HOL blocking by opening multiple concurrent connections to each domain — most browsers today use six parallel connections — with an upper limit on the number of active connections across all domains.

With pipelining, multiple HTTP requests can be sent to a server together without waiting for the corresponding responses as shown in Figure 1(c). The client then waits for the responses to arrive in the order in which they were requested. Pipelining can improve page load times dramatically. However, since the server is required to send its responses in the same order that the requests were received, HOL blocking can still occur with pipelining. Some mobile browsers have only recently started supporting pipelining.

2.2 The SPDY Protocol

Even though HTTP is widely adopted and used today, it suffers from several shortcomings (e.g., sequential requests, HOL blocking, short-lived connections, lack of server initiated data exchange, etc.) that impact web performance, especially on the cellular network.

SPDY [7] is a recently proposed application-layer protocol for transporting content over the web with the objective of minimizing latency. The protocol works by opening one TCP connection per domain (or just one connection if going via a proxy). SPDY then allows for unlimited concurrent streams over this single TCP connection. Because requests are interleaved on a single connection, the efficiency of TCP is much higher: fewer network connections need to be made, and fewer, but more densely packed, packets are issued.

SPDY implements request priorities to get around one object request choking up the connection. This is described in Figure 1(d). After downloading the main page, and identifying the objects on the page, the client requests all four objects in quick succession, but marks objects 3 and 4 to be of higher priority. As a result, server transfers these objects first thereby preventing the connection from being congested with non-critical resources (objects 2 and 5) when high priority requests are pending. SPDY also allows for multiple responses to be transferred as part of the same packet (e.g. objects 2 and 5 in Figure 1(d)) can fit in a single response packet can be served altogether. Finally, SPDY compresses request and response HTTP headers and Server-initiated data exchange. All of these optimizations have shown to yield up to 64% reduction in page load times with SPDY [7].

3. EXPERIMENTAL SETUP

We conducted detailed experiments comparing the performance of HTTP and SPDY on the 3G network of a commercial, production US cellular provider over a four month period in 2013.

Figure 2 provides an overview of our test setup. Clients in our setup connect over the cellular network using HTTP

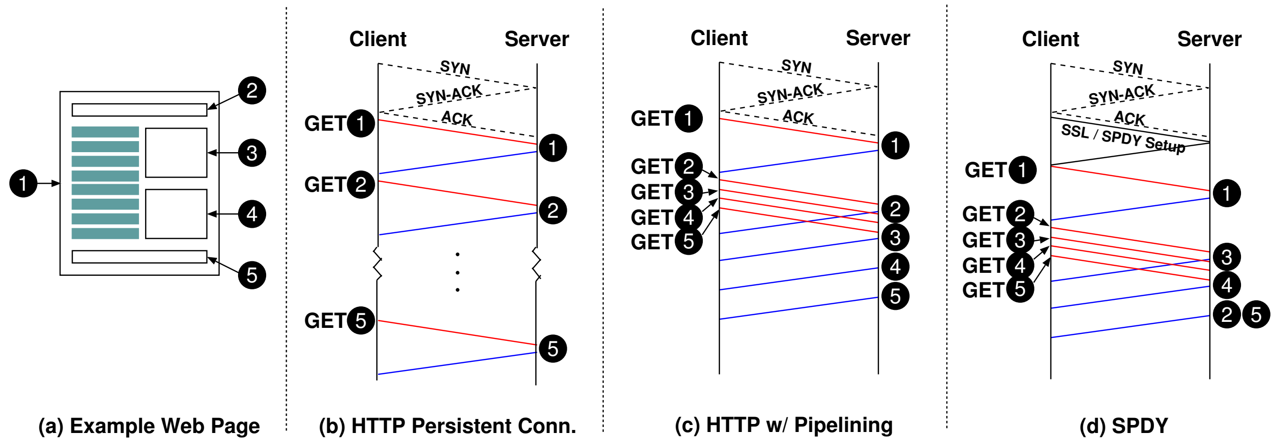

Figure 1: Example showing how HTTP and SPDY work.

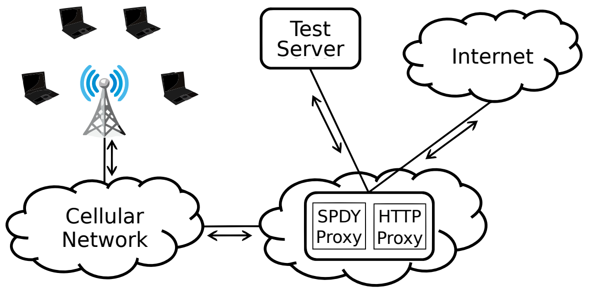

Figure 2: Our test setup

or SPDY to proxies that support that corresponding protocol. These proxies then use persistent HTTP to connect to the different web servers and fetch requested objects. We run a SPDY and an HTTP proxy on the same machine for a fair comparison. We use a proxy as an intermediary for two reasons: (a) We necessarily could not compare SPDY and HTTP directly. There are relatively few web sites that support SPDY. Moreover, a web server running SPDY would not support HTTP and vice versa. Thus, we would be evaluating connections to different servers which could affect our results (depending on their load, number of objects served, etc). (b) Most cellular operators in the US already use HTTP proxies to improve web performance. Running a SPDY proxy would allow operators to support SPDY over the cellular network even if the web sites do not.

Test Devices: We use laptops running Windows 7 and equipped with 3G (UMTS) USB cards as our client devices. We ran experiments with multiple laptops simultaneously accessing the test web sites to study the effect of multiple users loading the network. There are several reasons we use a laptop for our experiments. First, tablets and cellular-equipped laptops are on the rise. These devices request the regular web pages unlike smart phones. Second, and more importantly, we wanted to eliminate the effects of a slow processor as that could affect our results. For example, studies [16] have shown that HTML, Javascript, and CSS processing and rendering can delay the request of required objects and significantly affect the overall page load time. Finally, it has been observed [13] that having a slow processor increases the number of zero window advertisements, which significantly affects throughput.

Test Client: We used a default installation of the Google Chrome browser (ver 23.0) as the test client, as it supported traversing a SPDY proxy. Depending on the experiment, we explicitly configured Chrome to use either the HTTP or the SPDY proxy. When using a HTTP proxy, Chrome opens up to 6 parallel TCP connections to the proxy per domain, with a maximum of 32 active TCP connections across all domains. With SPDY, Chrome opens one SSL-encrypted TCP connection and re-uses this connection to fetch web objects. The connection is kept persistent and requests for different websites re-use the connection.

Test Location: Cellular experiments are sensitive to a lot of factors, such as signal strength, location of the device in a cell, the cell tower's backhaul capacity, load on the cell tower, etc. For example, a device at a cell edge may frequently get handed-off between towers, thereby contributing to added delays. To mitigate such effects, we identified a cell tower that had sufficient backhaul capacity and had minimal interference from other cell sites. For most of our experiments, we chose a physical location with an unobstructed view of the tower and received a strong signal (between -47 and -52 dBm). We configured the 3G modem to remain connected to that base station at that sector on a particular channel frequency and used a diagnostic tool to monitor the channel on that sector.

Proxies Used: We used a virtual machine running Linux in a compute cloud on the east coast of US to host our proxies. At the time of our experiments, there were no proxy implementations that supported both HTTP and SPDY. Hence we chose implementations that are purported to be widely used and the most competent implementations for the corresponding protocols. We used Squid [2] (v3.1) as our HTTP proxy. Squid supports persistent connections to both the client and the server. However, it only supports a rudimentary form of pipelining. For this reason, we did not run experiments of HTTP with pipelining turned on. Our comparisons are restricted to HTTP with multiple persistent connections. For SPDY, we used a SPDY server built by Google and made available as part of the Chromium source tree. This server was used in the comparison [7] of SPDY and HTTP and has since had extensions built in to support proxying.[1] We ran tcpdump to capture network level packet traces and `tcp-probe` kernel module to capture TCP congestion window values from the proxy to the mobile device.

[1]We also tested performance with a SOCKS proxy, but found the results to be worse than both HTTP and SPDY.

Website	Total Objs	Avg. Size (KB)	Avg. No. of Domains	Avg. Text Objs	Avg. JS/ CSS	Avg. Imgs/ Other
Finance	134.8	626.9	37.6	28.6	41.3	64.9
Entertainment	160.6	2197.3	36.3	16.5	28.0	116.1
Shopping	143.8	1563.1	15.8	13.3	36.8	93.7
Portal	121.6	963.3	27.5	9.6	18.3	93.7
Technology	45.2	602.8	3.0	2.0	18.0	25.2
ISP	163.4	1594.5	13.2	13.2	36.4	113.8
News	115.8	1130.6	28.5	9.1	49.5	57.2
News	157.7	1184.5	27.3	29.6	28.3	99.8
Shopping	5.1	56.2	2.0	3.1	2.0	0.0
Auction	59.3	719.7	17.9	6.8	7.0	45.5
Online Radio	122.1	1489.1	17.9	24.1	21.0	77.0
Photo Sharing	29.4	688.0	4.0	2.3	10.0	17.1
Technology	63.4	895.1	9.0	4.1	15.0	44.3
Baseball	167.8	1130.5	12.5	19.5	94.0	54.3
News	323	1722.7	84.7	73.4	73.6	176.0
Football	267.1	2311.0	75.0	60.3	56.9	149.9
News	218.5	4691.3	37.0	19.0	56.3	143.2
Photo Sharing	33.6	1664.8	9.1	3.3	6.7	23.6
Online Radio	68.7	2908.9	15.5	5.2	23.8	39.7
Weather	163.2	1653.8	48.7	19.7	45.3	98.2

Table 1: Characteristics of tested websites. The numbers are averaged across runs.

Web Pages Requested: We identified the top web sites visited by mobile users to run our tests (in the top Alexa sites). Of these, we eliminated web sites that are primarily landing pages (e.g., Facebook login page) and picked the remaining 20 most requested pages. These 20 web pages have a good mix of news websites, online shopping and auction sites, photo and video sharing as well as professionally developed websites of large corporations. We preferred the "full" site instead of the mobile versions keeping in mind the increasing proliferation of tablets and large screen smartphones. These websites contain anywhere from 5 to 323 objects, including the home page. The objects in these sites were spread across 3 to 84 domains. Each web site had HTML pages, Javascript objects, CSS and images. We tabulate important aspects of these web sites in Table 1.

Test Execution: We used a custom client that talks to Chrome via the remote debugging interface and got Chrome to load the test web pages. We generated a random order in which to visit the 20 web sites and used that same order across all experiments. Each website was requested 60 seconds apart. The page may take much shorter time to load; in that case the system would be idle until the 60 second window elapsed. We chose 60 seconds both to allow for web pages to load completely and to reflect a nominal think time that users take between requests.

We used page load time as the main metric to monitor performance. Page load time is defined as the time it takes the browser to download and process all the objects associated with a web page. Most browsers fire a javascript event (`onLoad()`) when the page is loaded. The remote debugging interface provided us the time to download the different objects in a web page. We alternated our test runs between HTTP and SPDY to ensure that temporal factors do not affect our results. We ran each experiment multiple times during the typically quiet periods (e.g., 12 AM to 6 AM) to mitigate effects of other users using the base station.

4. EXPERIMENTAL RESULTS

We first compare the performance of SPDY and HTTP using data collected from a week's worth of experiments. Since there was a lot of variability in the page load times, we use a box plot to present the results in Figure 3. The x-axis shows the different websites we tested; the y-axis is the page load time in milliseconds. For each website, the (red) box on the left shows the page load times for HTTP, while the (blue) box on the right shows the times for SPDY. The box plot gives the standard metrics: the 25 percentile, the 75 percentile and the black notch in the box is the median value. The top and bottom of the whiskers shows the maximum and minimum values respectively. Finally, the circle in these boxes shows the mean page load time across all the runs.

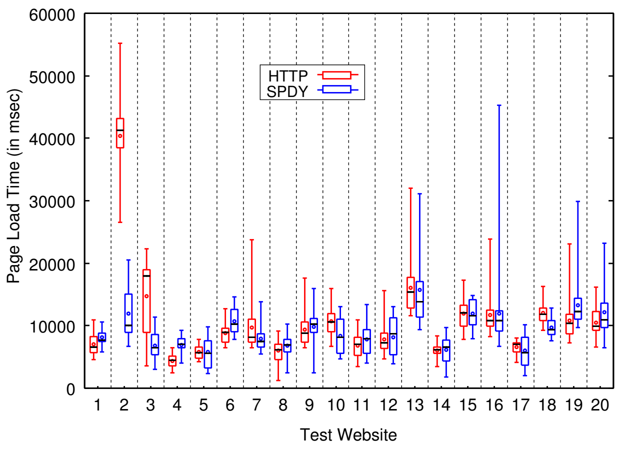

Figure 3: Page Load Time for different web sites with HTTP and SPDY.

The results from Figure 3, interestingly, do not show a convincing winner between HTTP and SPDY. For some sites, the page load time with SPDY is lower (e.g., 3, 7), while for others HTTP performs better (e.g., 1, 4). But for a large number of sites there isn't a significant difference.[2] This is in sharp contrast to existing results on SPDY where it has been shown to have between 27-60% improvement [7]. Importantly, previous results have shown an average of 23% reduction over emulated cellular networks [17].

4.0.1 *Performance over 802.11 Wireless Networks*

As a first step in explaining the result in Figure 3, we wanted to ensure that the result was not an artifact of our test setup or the proxies used. Hence, we ran the same experiments using the same setup, but over an 802.11g wireless network connected to the Internet via a typical residential broadband connection (15 Mbps down/ 2 Mbps up).

Figure 4 shows the average page load times and the 95% confidence intervals. Like previous results [7], this result also shows that SPDY performs better than HTTP consistently with page load time improvements ranging from 4% for website 4 to 56% for website 9 (ignoring website 2). Since the only difference between the two tests is the access network, we conclude that our results in Figure 3 is a consequence of how the protocols operate over the cellular network.

5. UNDERSTANDING THE CROSS-LAYER INTERACTIONS

We look at the different components of the application and the protocols that can affect performance. In the process we

[2]HTTP seems to perform quite poorly with site 2. Upon investigation, we found that the browser would occasionally stall on this site. These stalls happened more often with HTTP than with SPDY resulting in increased times.

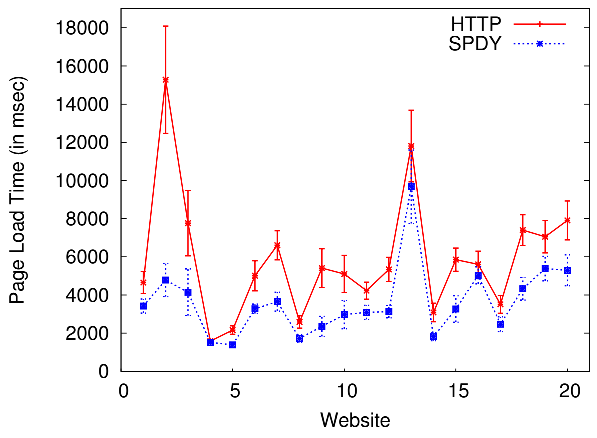

Figure 4: Average Page Load Time over an 802.11g/Broadband network.

observe that there are significant interdependencies between the different layers (from browser behavior and web page design, to TCP protocol implementations, to the intricacies of the cellular network) that affect overall performance.

5.1 Object download times

The first result we study is the break down of the page load time. Recall that, by default, the page load time is the time it takes the browser to process and download all the objects required for the web page. Hence, we look into the average download time of objects on a given page. We split the download time of the object into 4 steps: (a) the *initialization* step which includes the time from when the browser realizes that it requires the object to when it actually requests the object, (b) the *send* step which includes the time to actually send the request over the network, (c) the *wait* time which is the time between sending the request till the first byte of response, and finally (d) the *receive* time which is the time to receive the object.

We plot the average time of these steps for the different web sites in Figure 5. First, we see that the trends for average object download time are quite similar to that of page load times (in Figure 3). This is not surprising given that page load time is dependent on the object download times. Next, we see that the send time is almost invisible for both HTTP and SPDY indicating that sending the request happens very quickly. Almost all HTTP requests fit in one TCP packet. Similarly almost all SPDY requests also fit in a single TCP packet; even when the browser bundles multiple SPDY requests in one packet. Third, we see that receive times with HTTP and SPDY are similar, with SPDY resulting in slightly better average receive times. We see that the initialization time is much higher with HTTP because the browser has to either open a new TCP connection to download the object (and add the delay of a TCP handshake), or wait until it can re-use an existing TCP connection.

SPDY incurs very little initialization time because the connection is pre-established. On the other hand, it incurs a significant wait time. Importantly, this wait time is significantly higher than the initialization time for HTTP. This negates any advantages SPDY gains by reusing connections and avoiding connection setup. The wait times for SPDY are much greater because multiple requests are sent together or in close succession to the proxy. This increases delay as

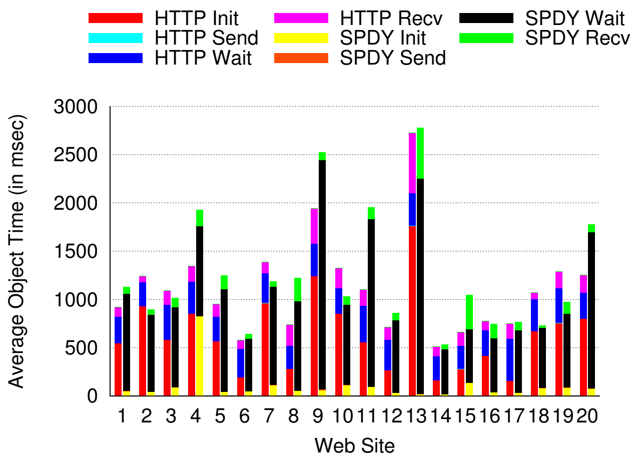

Figure 5: Split of average download times of objects by constituent components.

the proxy catches up in serving the requests to the client. Figure 7 discussed in the next section shows this behavior.

5.2 Web Page design and object requests

We now look at when different objects for a website are requested by the browser. One of the performance enhancements SPDY allows is for all objects to be requested in parallel without waiting for the response of outstanding objects. In contrast, HTTP has only one outstanding request per TCP connection unless pipelining is enabled.

We plot the request time (i.e., the time the browser sends out a request) for both HTTP and SPDY for four websites (due to space considerations) in Figure 6. Two of these are news websites and two contain a number of photos and videos. SPDY, unlike what was expected, does not actually request all the objects at the same time. Instead for three of the four web sites, SPDY requests objects in steps. Even for the one website where all the objects are requested in quick succession, we observe a delay between the first request and the subsequent requests. HTTP, on the other hand, requests objects continuously over time. The number of objects it downloads in parallel depends on the number of TCP connections the browser opens to each domain and across all domains.

We attribute this sequence of object requests to how the web pages are designed and how the browsers process them to identify constituent objects. Javascript and CSS files introduce interdependencies by requesting other objects. Table 1 highlights that websites make heavy use of JavaScript or CSS and contain anywhere from 2 to 73 different scripts and stylesheets. The browser does not identify these further objects until these files are downloaded and processed. Further, browsers process some of these files (e.g., Javascripts) sequentially as these can change the layout of the page. This results in further delays. The overall impact to page load speeds depends on the number of such files in a web page, and on the interdependencies in them.

To validate our assertion that SPDY is not requesting all the objects at once because of these interdependencies and also to understand better the higher wait time of objects, we built two test web pages that consist of only a main HTML page and images which we placed on a test server (see Fig. 2). There were a total of 50 objects that

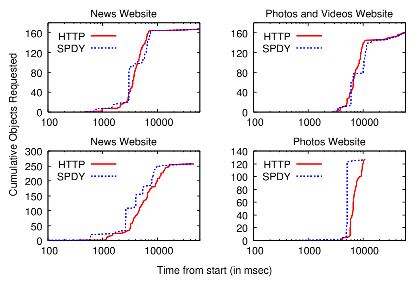

Figure 6: Object request patterns for different websites.

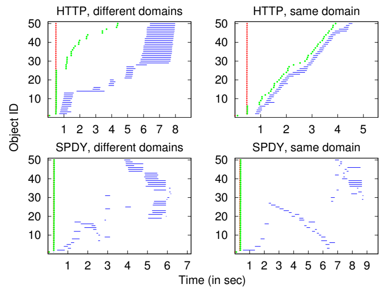

Figure 7: Object request and download with test web pages.

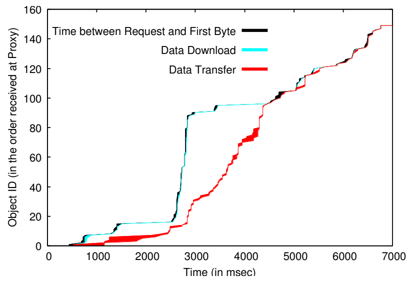

Figure 8: Queuing delay at the proxy

needed to be downloaded as part of the web page. We controlled the effect of domains by testing the two extremes: in one web page, all the objects came from different domains, while in the second extreme all the objects came from the same domain. Figure 7 shows the results of these two tests. Since there are no interdependencies in the web page, we see that the browser almost immediately identifies all the objects that need to be downloaded after downloading the main HTML page (shown using red dots). SPDY then requests all the images on the page in quick succession (shown in green dots) in both cases. HTTP on the other hand, is affected by these extremes. When all objects are on different domains, the browser opens one connection to each domain up to a maximum number of connection (32 in the case of Chrome). When all the objects are on the same domain, browsers limit the number of concurrent connections (6 in the case of Chrome) but reuse the connections.

Note that while the requests for SPDY are sent out earlier (green dots) than HTTP, SPDY has much more significant delay until the first byte of data is sent back to the client (start of blue horizontal line). Moreover, we also observe especially in the different domain case, that if multiple objects are downloaded in parallel the time to receive the objects (length of blue line) is increased. We find in this experiment that removing all the interdependencies for SPDY does not significantly improve the performance. In our tests, HTTP had an average page load time of 5.29s and 6.80s with single vs multiple domains respectively. Conversely, SPDY averages 7.22s and 8.38s with single or multiple domain tests. Consequently, prioritization alone is not a panacea to SPDY's performance in cellular networks.

5.3 Eliminating Server-Proxy link bottleneck

Figures 6 and 7 show that while today's web pages do not take full advantage of SPDY's capabilities, that is not a reason for the lack of performance improvements with SPDY in cellular networks. So as the next step, we focus on the proxy and see if the proxy-server link is a bottleneck.

In Figure 8 we plot the sequence of steps at the proxy for a random website from one randomly chosen sample execution with SPDY. The figure shows the objects in the order of requests by the client. There are three regions in the plot for each object. The black region shows the time between when the object was requested at the proxy to when the proxy receives the first byte of response from the web server. The next region, shown in cyan, represents the time it takes the proxy to download the object from the web server, starting from the first byte that it receives. Finally, the red region represents the time it takes the proxy to transfer the object back to the client.

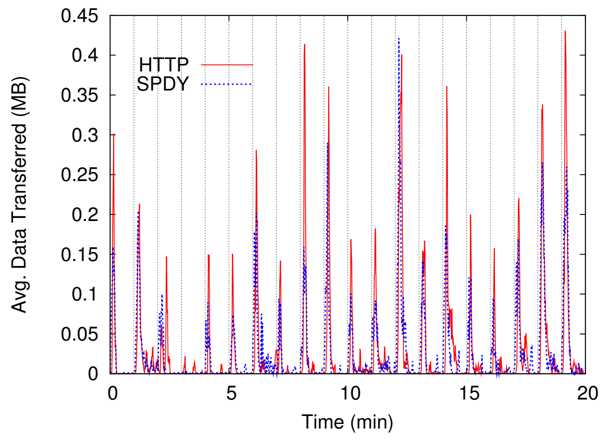

Figure 9: Average data transferred from proxy to device every second.

It is clear from Figure 8 that the link between the web server and proxy is not the bottleneck. We see that in most cases, the time between when the proxy receives the request from the client to when it has the first byte of data from the web server is very short (average of 14 msec with a max of 46 msec). The time to download the data, at an average of 4 msec, is also quite short. Despite having the data, however, we observe that the proxy is unable to send the data quickly to the client device. There is a significant delay between when the data was downloaded to the proxy to when it begins to send the data to the client.

This result shows that SPDY has essentially moved the bottleneck from the client to the proxy. With HTTP, the client does not request objects until the pending ones are downloaded. If these downloads take a while, the overall download process is also affected. In essence, this is like admission control at the client. SPDY gets rid of this by requesting all the objects in quick succession. While this works well when there is sufficient capacity on the proxy-client link, the responses get queued up at the proxy when the link between the proxy and the client is a bottleneck.

5.4 Throughput between client and proxy

The previous result showed that the proxy was not able to transfer objects to the client quickly, resulting in long wait times for SPDY. Here, we study the average throughput achieved by SPDY and HTTP during the course of our experiments. Since each website is requested exactly one minute apart, in Figure 9 we align the start times of each experiment, bin the data transferred by SPDY and HTTP each second, and compute the average across all the runs.

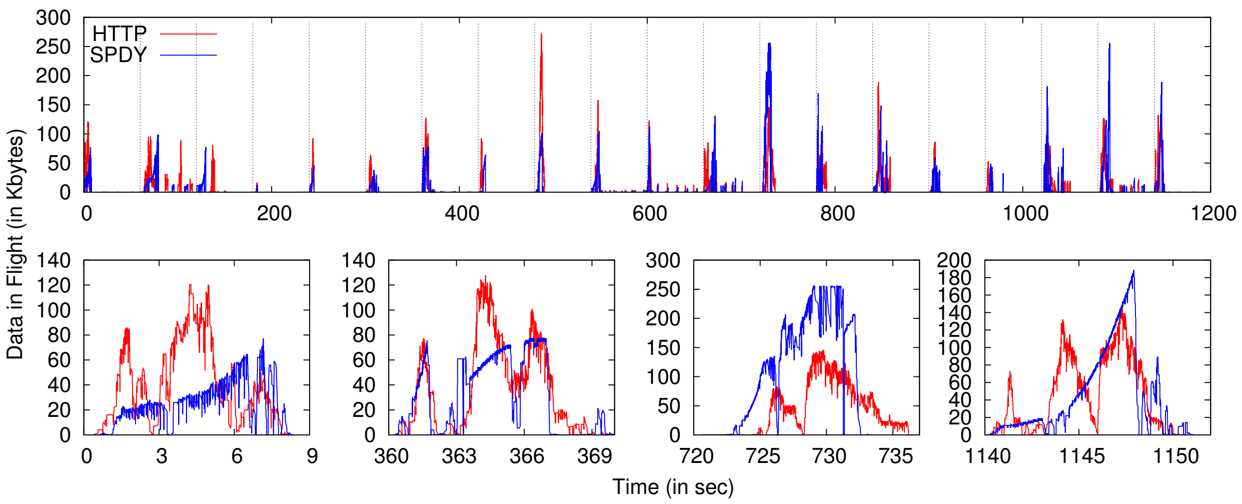

Figure 10: The number of unacknowledged bytes for a random run with HTTP and SPDY.

The figure shows the average amount of data that was transferred during that second. The vertical lines seen every minute indicate the time when a web page was requested. We see from the graph that HTTP, on average, achieves higher data transfers than SPDY. The difference sometimes is as high as 100%. This is a surprising result because, in theory, the network capacity between the client and the proxy is the same in both cases. The only difference is that HTTP uses multiple connections each of which shares the available bandwidth, while with SPDY the single connection uses the entire capacity. Hence, we would expect the throughput to be similar; yet they are not. Since network utilization is determined by how TCP adapts to available capacity, we shift our attention to how TCP behaves in the cellular network.

5.5 Understanding TCP performance

To understand the cause for the lower average throughput with SPDY, we look at how TCP behaves when there is one connection compared to when there are multiple connections. We start by looking at the outstanding bytes in flight between the proxy and the client device with HTTP and SPDY. The number of bytes in flight is defined as the number of bytes the proxy has sent to the client that are awaiting acknowledgment. We plot the data from one random run of the experiment in Figure 10.

Figure 10 shows that there are instances where HTTP has more unacknowledged bytes, and other instances where SPDY wins. When we looked at the correlation between page load times and the number of unacknowledged bytes, we found that whenever the outstanding bytes is higher, it results in lower page load times. To illustrate this, we zoom into four websites (1, 7, 13 and 20) from the same run and plot them in the lower half of Figure 10. For the first two websites, HTTP has more unacknowledged data and hence the page load times was lower (by more than one second), whereas for 13 and 20, SPDY has more outstanding data and hence lower page load times (faster by 10 seconds and 2 seconds respectively). We see that the trend applied for the rest of the websites and other runs. In addition, we see in websites 1 and 20 that the growth in outstanding bytes (i.e., the growth of throughput) is quite slow for SPDY. We have already established in Figure 8 that the proxy is not starved for data. Hence, the possible reasons for limiting the amount of data transferred could be either limits in the sender's congestion window or the receiver window.

5.5.1 *Congestion window growth*

We processed the packet capture data and extracted the receive window (`rwin`) advertised by the client. From the packet capture data, it was pretty clear that `rwin` was not the bottleneck for these experimental runs. So instead we focused on the proxy's congestion window and its behavior. To get the congestion window, we needed to tap into the Linux kernel and ran a kernel module (`tcp_probe`) that reports the congestion window (`cwnd`) and slow-start threshold (`ssthresh`) for each TCP connection.

Figure 11 shows the congestion window, `ssthresh`, the amount of outstanding data and the occurrence of retransmissions during the course of one random run with SPDY. First we see that in all cases, the `cwnd` provides the ceiling on the outstanding data, indicating that it is the limiting factor in the amount of data transferred. Next we see that both the `cwnd` and the `ssthresh` fluctuate throughout the run. Under ideal conditions, we would expect them to initially grow and then stabilize to a reasonable value. Finally, we see many retransmissions (black circles) throughout the duration of the run (in our plot, the fatter the circle, the greater the number of retransmissions.)

To gain a better understanding, we zoom into the interval between 40 seconds and 190 seconds in Figure 12. This represents the period when the client issues requests to websites 2, 3, and 4. The vertical dashed line represents time instances where there are retransmissions. From Figure 12 we see that, at time 60, when accessing website 2, both the `cwnd` and `ssthresh` are small. This is a result of multiple retransmissions happening in the time interval 0-60 seconds (refer Figure 11). From 60 to 70 seconds, both the `cwnd` and `ssthresh` grow as data is transferred. Since the `cwnd` is higher than the `ssthresh`, TCP stays in congestion avoidance and does not grow as rapidly as it would in 'slow-start'. The pattern of growth during the congestion avoidance phase is also particular to TCP-Cubic (because it first probes and then has an exponential growth).

After about 70 seconds, there isn't any data to transfer and then the connection goes idle until about 85 seconds. This is the key period of performance loss: At this time,

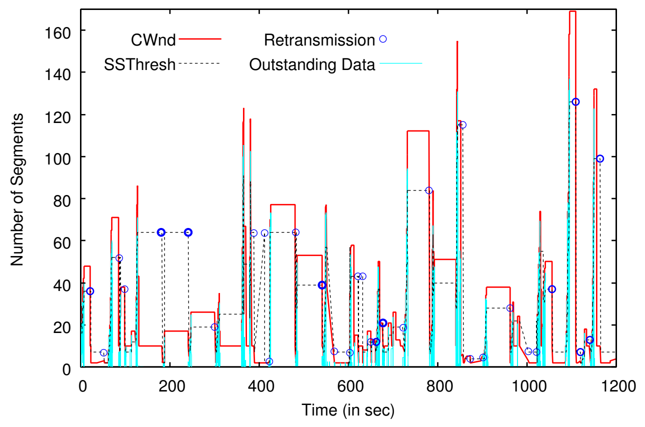

Figure 11: The cwnd, ssthresh, and outstanding data for one run of SPDY. The figure also shows times at which there are retransmissions.

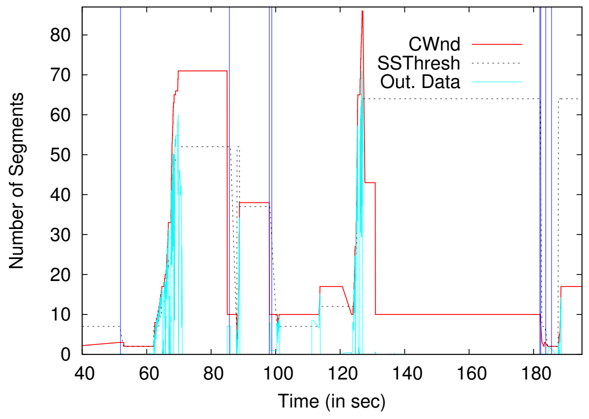

Figure 12: The cwnd, ssthresh, and outstanding data for three consecutive websites.

when the proxy tries to send data, multiple effects are triggered. First, since the connection has been idle, a TCP parameter (`tcp_slow_start_after_idle`) is triggered. Intuitively this parameter captures that fact that network bandwidth could have changed during the idle period and hence it makes sense to discard all the estimates of the available bandwidth. As a result of this parameter, TCP reduces the `cwnd` to the default initial value of 10. Note that the `ssthresh` and retransmission timeout (RTO) values are left unmodified; as a result the connection goes through slow start until `cwnd` grows beyond the `ssthresh`.

Cellular networks make use of a radio resource controller (RRC) state machine to manage the state of the radio channel for each device.[3] The radio on the device transitions between idle and active states to conserve energy and share the radio resources. Devices transfer the most data when they are in the active (or DCH) state. They transition to idle after a small period of inactivity. When going from idle to active, the state machine imposes a *promotion* delay, which is typically around 2 seconds [12]. This promotion delay results in a period in which TCP does not receive any acknowledgments either. Since TCP's RTO value is not reset after an idle period, and this RTO value is much smaller than the promotion delay, it results in a TCP time out and subsequent retransmissions (refer Figure 11). As a consequence, `cwnd` is reduced and the `ssthresh` is set to a value based on the `cwnd` (the specific values depend on the flavor of TCP). TCP then enters slow start and `cwnd` and `ssthresh` grow back quickly to their previous values (again this depends on the version of TCP, and in this case depends on the behavior of TCP-Cubic). As a result of an idle and subsequent retransmission, a similar process repeats itself twice, at 90 and 120 seconds with the `cwnd` and `ssthresh`. Interestingly, at 110 seconds, we do not see retransmissions even though there was an idle period. We attribute this to the fact that the RTO value is grown large enough to accommodate the increased round trip time after the idle time.

[3]Refer to Appendix A for a brief description of the RRC state machine.

When website 3 is requested at time 120, the `cwnd` and `ssthresh` grow as data is transferred. The website also transfers small amounts of data at around 130 seconds, after a short idle period. That causes TCP to reduce its `cwnd` to 10. However the idle period is short enough that the cellular network does not go idle. As a result, there are no retransmissions and the `ssthresh` stays at 65 segments. The `cwnd` remains at 10 as no data was transferred after that time. When website 4 is requested at 180 seconds, however, the `ssthresh` falls dramatically because there is a retransmission (TCP as well as the cellular network become idle). Moreover, there are multiple retransmissions as the RTT estimates no longer hold.

5.5.2 *Understanding Retransmissions*

One of the reasons for both SPDY and HTTP's performance issues is the occurrence of TCP retransmissions. Retransmissions result in the collapse of TCP congestion window, which in turn hurts throughput. We analyze the occurrence of retransmissions and its cause in this section.

There are on average 117.3 retransmissions for HTTP and 67.3 for SPDY. We observed in the previous section that most of the TCP retransmissions were spurious due to an overly tight RTO value. Upon close inspection of one HTTP run, we found all (442) retransmissions were in fact spurious. On a per connection basis, HTTP has fewer retransmissions (2.9) since there are 42.6 concurrent TCP connections open on average. Thus, the 67.3 retransmits for SPDY results in much lower throughput. We also note from our traces that the retransmissions are bursty in nature and typically affect a few (usually one) TCP connections. Figure 13 shows that even though HTTP has a higher number of retransmissions, when one connection's throughput is compromised, other TCP connections continue to perform unaffected. Since HTTP uses a 'late binding' of requests to connections (by allowing only one outstanding request per connection), it is able to avoid affected connections, and maintain utilization of the path between the proxy and the end-device. On the other hand, since SPDY opens only one TCP connection, all these retransmissions affect its throughput.

5.6 Cellular network behavior

5.6.1 *Cellular State Machine*

In this experiment we analyze the performance improvement gained by they device staying in the DCH state. Since there is a delay between each website request, we run a continual ping process that transfers a small amount of data every few seconds. We choose a payload that is small enough

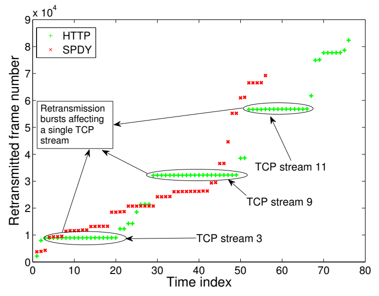

Figure 13: Retransmission bursts affecting a single TCP stream

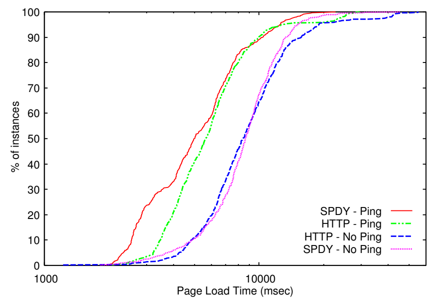

Figure 14: Impact of cellular RRC state machine.

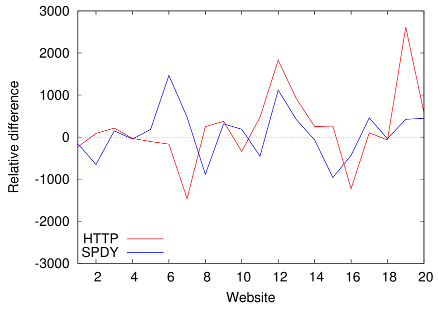

Figure 15: Page load times with & w/o `tcp_slow_start_after_idle`

to not interfere with our experiments, but large enough that the state machine keeps the device in DCH mode.

Figure 14 shows the CDF of the page load times for the different websites across the different runs. Unsurprisingly, the result shows that having the cellular network in DCH mode through continuous background ping messages significantly improves the page load time of both HTTP and SPDY. For example, more than 80% of the instances load in less than 8 seconds when the device sends continual ping messages, but only between 40% (SPDY) and 45% (HTTP) complete loading without the ping messages. Moreover, SPDY performs better than HTTP for about 60% of the instances with the ping messages. We also looked into the number of retransmissions with and without ping messages; not surprisingly, we observed that the number of retransmissions reduced by ∼91% for HTTP and ∼96% for SPDY indicating that TCP RTT estimation is no longer impacted by the cellular state machine. While this result is promising, it is not practical to keep the device in DCH state as it wastes cellular resources and drains device battery. Hence, mechanisms need to be built into TCP that account for the cellular state machine.

5.6.2 Performance over LTE

We analyze the performance of HTTP and SPDY over LTE in this section. LTE adopts an improved RRC state machine with a significantly smaller promotion delay. On the other hand, LTE also has lower round-trip times compared to 3G, which has the corresponding effect of having much smaller RTO values. We perform the same experiments using the same setup as in the previous 3G experiments, but connect to an LTE network with LTE USB laptop cards.

Figure 16 shows the box plot of page load times for HTTP and SPDY over LTE. As expected, we see that both HTTP and SPDY have considerably smaller page load times compared to 3G. We also see that HTTP performs just as well as SPDY, if not better, for the initial few pages. However, SPDY's performance is better than HTTP after the initial set of web pages. We attribute this to the fact that LTE's RRC state machine addresses many of the limitations present in the 3G state machine, thereby allowing TCP's congestion window to grow to larger values and thus allowing SPDY to transfer data more quickly. We also looked at the retransmission data for HTTP and SPDY – the number of retransmissions reduced significantly with an average of 8.9 and 7.52 retransmissions per experiment with HTTP and SPDY (as opposed to 117 and 63 with 3G) respectively.

While the modified state machine of LTE results in better performance, we also wanted to see if it eliminated the issue of retransmission as a result of the state promotion delay. We focus on a short duration of a particular, randomly selected, run with SPDY in Figure 17. The figure shows the congestion window of the TCP connection (in red), the amount of data in flight (in cyan) and the times when there are retransmissions (in black). The thicker retransmission lines indicate multiple retransmissions. We see from the figure that retransmissions occur after an idle period in LTE also. For example, at around 600 seconds, the proxy tries to send data to the device after an idle period; timeouts occur after the transmission of data, leading to retransmissions; and the congestion window collapses. This result leads us to believe that the problem persists even with LTE, albeit less frequently than with 3G.

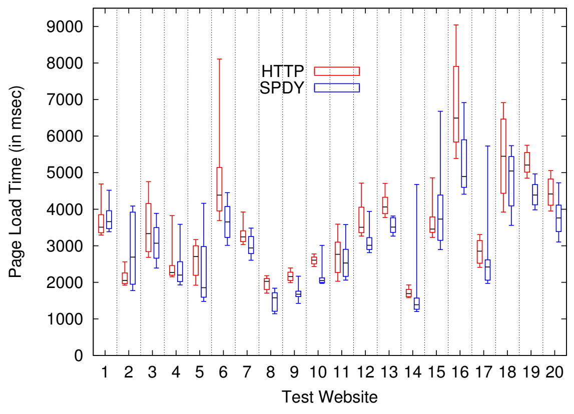

Figure 16: Page Load Time of HTTP and SPDY over LTE

5.7 Summary and Discussion

We see from these results how the interaction between the different layers affects performance. First we see websites sending and/or requesting data periodically (ads, tracking cookies, web analytics, page refreshes, etc.). We also observe that a key factor affecting performance is the independent reaction of the transport protocol (i.e., TCP) and the cellular network to inferred network conditions.

TCP implementations assume their `cwnd` statistics do not hold after an idle period as the network capacity might have changed. Hence, they drop the `cwnd` to its initial value. That in itself would not be a problem in wired networks as the `cwnd` will grow back up quickly. But in conjunction with the cellular network's idle-to-active promotion delay, it results in unintended consequences. Spurious retransmissions occurring due to the promotion delay cause the `ssthresh` to fall to the `cwnd` value. As a result, when TCP tries to recover, it goes through slow start only for a short duration, and then

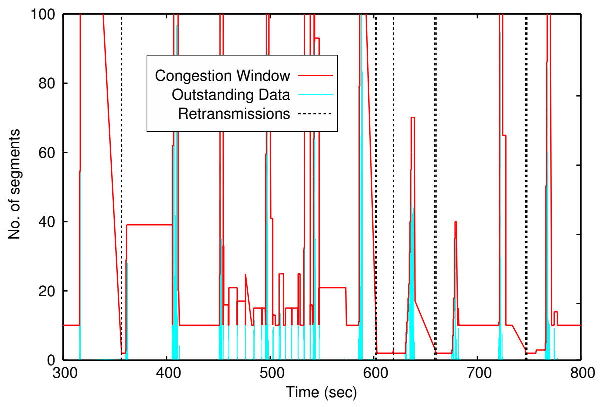

Figure 17: SPDY's Congestion window and retransmissions over LTE.

switches to congestion avoidance, even for small number of segments. From a TCP standpoint, this defeats the design intent where short transfers that do not have the potential of causing congestion (and loss) should be able to rapidly acquire bandwidth, thus reducing transfer time. This difficulty of transport protocols 'shutting down' after an idle period at just the time when applications wake up and seek to transfer data (and therefore requiring higher throughput) is not new and has been observed before [8]. However, the process is further exacerbated in cellular networks with the existence of a large promotion delay. These interactions thus degrade performance, including causing multiple (spurious) retransmissions that have significant undesirable impacts on the individual TCP connection behavior.

Our results also point to a *fundamental flaw in TCP implementations*. Existing implementations discard the congestion window value after an idle period to account for potential changes in the bandwidth during the idle period. However, information about the latency profile (i.e., RTT estimates) are retained. With the cellular state machine, the *latency profile also changes* after an idle period; since the estimates are inaccurate, it results in spurious retransmissions. We notice that LTE, despite an improved state machine, is still susceptible to retransmissions when coming out of the idle state. When we keep the device in active mode continuously, we transform the cellular network to behave more like a traditional wired (and also a WiFi) network in terms of latency profile. Consequently, we see results similar to the ones seen over wired networks.

6. POTENTIAL IMPROVEMENTS

Having identified the interactions between TCP and the cellular network as the root cause of the problem, in this section, we propose steps that can minimize their impact.

6.1 Using multiple TCP connections

The observation that using a single TCP connection causes SPDY to suffer because of retransmissions suggests a need to explore the use of multiple TCP connections. We explore this option by having the browser use 20 SPDY connections to a single proxy process listening on 20 different ports.[4] However, the use of multiple TCP connections *did not* help in improving the page load times for SPDY. This is primarily because with SPDY, requests are issued to each connection up front. As a result, if a connection encounters retransmissions, pending objects requested on that connection are delayed. What is required is a late binding of the response to an 'available' TCP connection (meaning that it has a open congestion window and can transmit data packets from the proxy to the client at that instant) and avoiding a connection that is currently suffering from the effects of spurious timeouts and retransmissions. Such a late binding would allow the response to come back on any available TCP connection, even if the request was sent out on a different connection. This takes advantage of SPDY's capability to send the requests out in a 'burst', and allows the responses to be delivered to the client as they arrive back, avoiding any 'head-of-the-line blocking'.

[4]On the browser, we made use of a proxy auto config (PAC) file that dynamically allocate the proxy address and one of the 20 ports for each object requested.

6.2 TCP Implementation Optimizations

6.2.1 Resetting RTT Estimate after Idle

There is a fundamental need to decay the estimate of the available capacity of a TCP connection once it goes idle. The typical choice made today by implementations is to just reset `cwnd` to the initial value. The round trip time (RTT) estimate, however, is left untouched by implementations. The RTT estimate drives the retransmission timeout (RTO) value and hence controls when a packet is retransmitted. Not resetting the RTT estimate may be acceptable in networks that have mostly 'stable' latency characteristics (e.g., a wired or WiFi network), but as we see in our observations with the cellular network this leads to substantially degraded performance. The cellular network has vastly varying RTT values. In particular, the idle to active transition (promotion) can take a few seconds. Since the previous RTT estimate derived when the cellular connection was active may have been of the order of tens or hundreds of milliseconds, there is a high probability of a spurious timeout and retransmission of one or more packets after the idle period. These retransmissions have the cascading effect of reducing the `cwnd` further, and also reducing `ssthresh`. Therefore, when the `cwnd` starts growing, it grows in the congestion avoidance mode, which further reduces throughput. Thus the interaction of TCP with the RRC state machine of the cellular network has to be properly factored in to achieve the best performance. Our recommended approach is to reset the RTT estimate as well, to the initial default value (of multiple seconds). This causes the RTO value to be larger than the promotion delay for the 3G cellular network, thus avoiding spurious timeouts and unnecessary retransmissions. This, in turn, allows the `cwnd` to grow rapidly, ultimately reducing page load times.

6.2.2 Benefit of Slow Start after Idle?

One approach we also considered was whether avoiding the 'slow start after idle' would improve performance. We examined the benefit or drawback of the TCP connection transitioning to slow start after idle. We disabled the slow start parameter and studied the improvement in page load time. Figure 15 plots the relative difference between the average page load time of the different websites with and without this parameter enabled. A negative value on the Y-axis indicates that disabling the parameter is beneficial, while a positive value indicates that enabling it is beneficial.

We see that the benefits vary across different websites. Our packet traces indicate that the amount of outstanding data (and hence throughput) is quite similar in both the cases. The number of retransmitted packets seem similar under good conditions, but disabling the parameter runs the risk of having lots of retransmissions under congestion or poor channel conditions since the `cwnd` value is inaccurate after an idle period. In some instances, `cwnd` grows so large with the parameter disabled, that the receive window becomes the bottleneck and negates the benefit of a large congestion window at the sender.

6.2.3 Impact of TCP variants

We replaced TCP Cubic with TCP Reno to see if modifying the TCP variant has any positive impact on performance. We find in Table 2 that there is little to distinguish between Reno and Cubic for both HTTP and SPDY over 3G. We see that the average page load time across all the runs of all pages is better with Cubic. Average throughput is quite similar with Reno and Cubic, with SPDY achieving the highest value with Cubic. While this seemingly contradicts the result in Figure 9, note that this result is the average across all times (ignoring idle times), while the result in Figure 9 considers the average at that one second instant. Indeed the maximum throughput result confirms this: HTTP with Cubic achieves a higher throughput than SPDY with Cubic. SPDY with Reno does not grow the congestion window as much as SPDY with Cubic. This probably results in SPDY with Reno having the worst page load time across the combinations.

	Reno		Cubic	
	HTTP	SPDY	HTTP	SPDY
Avg. Page Load (msec)	9690.84	9899.95	9352.58	8671.09
Avg. Throughput (KBps)	121.88	119.55	115.36	129.79
Max. Throughput (KBps)	1024.74	528.88	889.33	876.98
Avg. `cwnd` (# segments)	10.45	24.16	10.59	52.11
Max. `cwnd` (# segments)	22	48	22	197

Table 2: Comparison of HTTP and SPDY with different TCP variants.

6.2.4 Cache TCP Statistics?

The Linux implementation of TCP caches statistics such as the slow start threshold and round trip times by default and reuses them when a new connection is established. If the previous connection had statistics that are not currently accurate, then the new connection is negatively impacted. Note that since SPDY uses only one connection, the only time these statistics come into play is when the connection is established. It can potentially impact HTTP, however, because HTTP opens a number of connections over the course of the experiments. We conducted experiments where we disabled caching. Interestingly, we find from our results that both HTTP and SPDY experience reduced page load times. For example, for 50% of the runs, the improvement was about 35%. However, there was very little to distinguish between HTTP and SPDY.

7. RELATED WORK

Radio resource management: There have been several attempts to improve the performance of HTTP over cellular networks (e.g. [10,12]). Specifically, TOP and TailTheft study efficient ways of utilizing radio resources by optimizing timers for state promotions and demotions. [5] studies the use of caching at different levels (e.g., nodeB, RNC) of a 3G cellular network to reduce download latency of popular web content.

TCP optimizations: With regards to TCP, several proposals have tried to tune TCP parameters to improve its performance [14] and address issues like Head of Line (HOL) blocking and multi-homing. Recently, Google proposed in an IETF RFC 3390 [4] to increase the TCP initial congestion window to 10 segments to show how web applications will benefit from such a policy. As a rebuttal, Gettys [6] demonstrated that changing the initial TCP congestion window can indeed be very harmful to other real-time applications that share the broadband link and attributed this problem to one of ”buffer bloat”. As a result Gettys, proposed the use of HTTP pipelining to provide improved TCP congestion behavior. In this paper, we investigate in detail how congestion window growth affects download performance for HTTP and SPDY in cellular networks. In particular, we demonstrate how idle-to-active transition at different protocol layers results in unintended consequences where there are retransmissions. Ramjee et al. [3] recognizes how challenging it can be to optimize TCP performance over 3G networks exhibiting significant delay and rate variations. They use an ACK regulator to manage the release of ACKs to the TCP source so as to prevent undesired buffer overflow. Our work inspects in detail how SPDY and HTTP behave and thereby TCP in cellular networks. Specifically, we point out a fundamental insight with regards to avoiding spurious timeouts. In conventional wired networks, bandwidth changes but the latency profile does not change as significantly. In cellular networks, we show that spurious timeout is caused by the fact that TCP stays with its original estimate for the RTT and a tight retransmission timeout (RTO) estimate derived over multiple round-trips during the active period of a TCP connection is not only invalid, but has significant performance impact. Thus, we suggest using a more conservative way to manage the RTO estimate.

8. CONCLUSION

Mobile web performance is one of the most important measures of users' satisfaction with their cellular data service. We have systematically studied, through field measurements on a production 3G cellular network, two of the most prominent web access protocols used today, HTTP and SPDY. In cellular networks, there are fundamental interactions across protocol layers that limit the performance of both SPDY as well as HTTP. As a result, there is no clear performance improvement with SPDY in cellular networks, in contrast to existing studies on wired and WiFi networks.

Studying these unique cross-layer interactions when operating over cellular networks, we show that there are fundamental flaws in implementation choices of aspects of TCP, when a connection comes out of an idle state. Because of the high variability in latency when a cellular end device goes from idle to active, retaining TCP's RTT estimate across this transition results in spurious timeouts and a corresponding burst of retransmissions. This particularly punishes SPDY which depends on the single TCP connection that is hit with the spurious retransmissions and thereby all the cascading effects of TCP's congestion control mechanisms like lowering `cwnd` etc. This ultimately reduces throughput and increases page load times. We proposed a holistic approach to considering all the TCP implementation features and parameters to improve mobile web performance and thereby fully exploit SPDY's advertised capabilities.

9. REFERENCES

[1] 3GPP TS 36.331: Radio Resource Control (RRC). `http://www.3gpp.org/ftp/Specs/html-info/36331.htm`.

[2] Squid Caching Proxy. `http://www.squid-cache.org`.

[3] Chan, M. C., and Ramjee, R. TCP/IP performance over 3G wireless links with rate and delay variation. In *ACM MobiCom* (New York, NY, USA, 2002), MobiCom '02, ACM, pp. 71–82.

[4] Chu, J., Dukkipati, N., Cheng, Y., and Mathis, M. Increasing TCP's Initial Window. `http://tools.ietf.org/html/draft-ietf-tcpm-initcwnd-08.html`, Feb. 2013.

[5] Erman, J., Gerber, A., Hajiaghayi, M., Pei, D., Sen, S., and Spatscheck, O. To cache or not to cache: The 3g case. *IEEE Internet Computing 15*, 2 (2011), 27–34.

[6] Gettys, J. IW10 Considered Harmful. `http://tools.ietf.org/html/draft-gettys-iw10-considered-harmful-00.html`, August 2011.

[7] Google. SPDY: An experimental protocol for a faster web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[8] Kalampoukas, L., Varma, A., Ramakrishnan, K. K., and Fendick, K. Another Examination of the Use-it-or-Lose-it Function on TCP Traffic. In *ATM Forum/96-0230 TM Working Group* (1996).

[9] Khaunte, S. U., and Limb, J. O. Statistical characterization of a world wide web browsing session. Tech. rep., Georgia Institute of Technology, 1997.

[10] Liu, H., Zhang, Y., and Zhou, Y. Tailtheft: leveraging the wasted time for saving energy in cellular communications. In *MobiArch* (2011), pp. 31–36.

[11] Popa, L., Ghodsi, A., and Stoica, I. HTTP as the narrow waist of the future internet. In *Hotnets-IX* (2010), pp. 6:1–6:6.

[12] Qian, F., Wang, Z., Gerber, A., Mao, M., Sen, S., and Spatscheck, O. TOP: Tail Optimization Protocol For Cellular Radio Resource Allocation. In *IEEE ICNP* (2010), pp. 285–294.

[13] Shruti Sanadhya, and Raghupathy Sivakumar. Adaptive Flow Control for TCP on Mobile Phones. In *IEEE Infocom* (2011).

[14] Stone, J., and Stewart, R. Stream Control Transmission Protocol (SCTP) Checksum Change. `http://tools.ietf.org/html/rfc3309.html`, September 2002.

[15] W3techs.com. Web Technology Surveys. `http://w3techs.com/technologies/details/ce-spdy/all/all.html`, June 2013.

[16] Wang, X. S., Balasubramanian, A., Krishnamurthy, A., and Wetherall, D. Demystifying Page Load Performance with WProf. In *Usenix NSDI'13* (Apr 2013).

[17] Welsh, M., Greenstein, B., and Piatek, M. SPDY Performance on Mobile Networks. `https://developers.google.com/speed/articles/spdy-for-mobile`, April 2012.

[18] Winstein, K., Sivaraman, A., and Balakrishnan, H. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Usenix NSDI'13* (Apr 2013).

APPENDIX

A. CELLULAR STATE MACHINES

The radio state of every device in a cellular network follows a well-defined state machine. This state machine, defined by 3GPP [1] and controlled by the radio network controller (in 3G) or the base station (in LTE), determines when a device can send or receive data. While the details of the

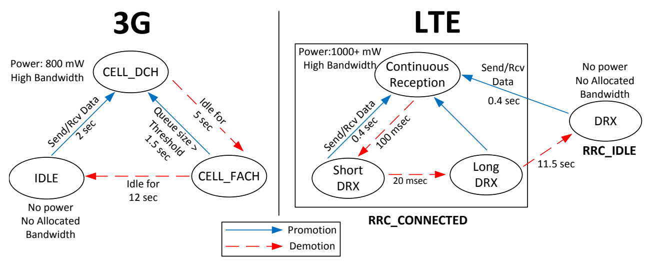

Figure 18: The RRC state machines for 3G UMTS and LTE networks

states, how long a device remains in each state, and the power it consumes in a state differ between 3G and LTE, the main purpose is similar: the occupancy in these states control the number of devices that can access the radio network at a given time. It enables the network to conserve and share available radio resources amongst the devices and for saving the device battery at times when the device does not have data to send or receive.

3G state machine: The 3G state machine, as shown in Figure 18, typically consists of three states: $IDLE$, Forward access channel ($CELL\_FACH$) and Dedicated channel ($CELL\_DCH$). When the device has no data to send or receive, it stays in the $IDLE$ state. The device does not have radio resource allocated to it in $IDLE$. When it wants to send or receive data, it has to be *promoted* to the $CELL\_DCH$ mode, where the device is allocated dedicated transport channels in both the downlink and uplink directions. The delay for this promotion is typically ~2 seconds. In the $CELL\_FACH$, the device does not have a dedicated channel, but can transmit at a low rate. This is sufficient for applications with small amounts or intermittent data. A device can transition between $CELL\_DCH$ and $CELL\_FACH$ based on data transmission activity. For example, if a device is inactive for ~5 seconds, it is *demoted* from $CELL\_DCH$ to $CELL\_FACH$. It is further demoted to $IDLE$ if there is no data exchange for another ~12 secs. Note that these state transition timer values are not general and vary across vendors and carriers.

LTE state machine: LTE employs a slightly modified state machine with two primary states: RRC_IDLE and $RRC\_CONNECTED$. If the device is in RRC_IDLE and sends or receives a packet (regardless of size), a state promotion from RRC_IDLE to $RRC\_CONNECTED$ occurs in about 400 msec. LTE makes use of three sub-states within $RRC\_CONNECTED$. Once promoted, the device enters Continuous Reception state where it uses considerable power (about 1000mW) but can send and receive data at high bandwidth. If there is a period of inactivity (e.g., for 100 msec), the device enters the short Discontinuous Reception ($Short\_DRX$) state . If data arrives, the radio returns to the Continuous Reception state in ~400 msec. If not, the device enters the long Discontinuous Reception ($Long\_DRX$) state. In the $Long\_DRX$ state, the device prepares to switch to the RRC_IDLE state, but is still using high power and waiting for data. If data does arrive within ~11.5 seconds, the radio returns to the Continuous Reception state; otherwise it switches to the low power (< 15 mW) RRC_IDLE state. Thus, compared to 3G, LTE has significantly shorter promotion delays. This shorter promotion delay helps reduce the number of instances where TCP experiences a spurious timeout and hence an unnecessary retransmission(s).