TECHNICAL UNIVERSITY OF KOŠICE

Faculty of Electrical Engineering and Informatics

Department of Computer and Informatics

Ing. Ivan Klimek

Wide Area Network Traffic Optimization

Written work to Doctoral Thesis

Study programme: *Informatics*

Study specialization: *Informatics*

Supervisor: *Assoc. Prof. Ing. František Jakab, PhD.*

Form of study: *Standard*

Košice, 2011

Acknowledgement

I would like to express my deep gratitude to a number of people who were essential for this work to become reality. The inspiring and positive attitude of Assoc. Prof. František Jakab has had an important role in guiding my studies practically from my first year at the Technical University of Košice. I would like to thank Cyril Measson, Aleksandar Jovicic, Tom Richardson and Mike Di Mare who significantly shaped my research interests and led to the selection of the problem addressed in this thesis. Last but not least, I am thankful to my family for their continual support. Once again, thank you, but this is just the beginning of a great journey …

Content

Introduction

It is not necessarily exaggerated to postulate that IT technologies have opened a new (cultural, professional, scientific, etc) era by revolutionizing the way people interact. There is something special about the Internet; it is probably one of the few high-tech technologies that are universally adapted in the same form worldwide. Every other technology has numerous variants, even such everyday things as combustion engines are not really standardized, every manufacturer is doing it a little bit different, in telecommunications there are various technologies, standards with dozens of their variants or even in the microchip industry every manufacturer has a special manufacturing process. Internet and its core concepts are except of a few details the same since its creation in the 1970s! Furthermore, a single one of the core protocols the Transmission Control Protocol (TCP) accounts for most of the Internet traffic. TCP provides guaranteed in-order delivery that enables higher layers to look on Internet as on a deterministic medium, a perfect FIFO pipe. As we will show TCP is not perfect, its inefficiency is based in its design, countless amounts of research tried using different methods to fix its problems, but with only limited success. In the first chapter we will look at the problems of TCP and show why without changing the core concepts behind it, it will not be possible to solve them. We will look at the Internet architecture and describe various currently applied methods that are trying to reduce the negative effects of its suboptimal design.

Since the beginning of Internet and TCP lots of things have changed. Technology progress in different areas brought new algorithms that did not exist when Internet's core concepts where designed. One of the most important paradigm shifts has been the advent of broadband wireless communications. Mobile phones are a phenomenon which was born when Internet already existed. The technologies that mobile devices carry and the innovations that they drove are based on state-of-the-art research in information, coding, and communications theory. For example, advances in channel coding and new algorithms permit to approach the fundamental limits of communications enabling high-speed wireless data transmissions. We will describe these algorithms in the second chapter, primarily focusing on Fountain codes that permit efficient Forward Error Correction (FEC). The third chapter will look at the FEC codes in the view of their application in the network protocols research.

The last chapter will summarize the current problems encountered in the Internet and sketch a realistic solution based on state-of-the-art but existing technologies. Our ambition is to consider such our solution, which we call the Universal Transport Protocol, as an incrementally deployable (and therefore realistic) successor of the Transmission Control Protocol.

1 The Internet Architecture

To make Internet more efficient we first need to understand its problems. For most people Internet is a synonym for World Wide Web which makes the Hypertext Transfer Protocol (HTTP) the most important protocol to focus optimization attempts on. The rich sites of Web 2.0 highlighted some fundamental problems of HTTP. We will look at them in the following text and also mention some proposed solutions. But HTTP is only the tip of the iceberg; its design follows the classic Internet status quo defined by TCP – it relies on an acknowledgment of every request, this scheme as we will show, without functioning parallelism creates performance problems. TCP builds its logic on ACKs, every aspect of its function is connected to it. We will describe TCP in detail and demonstrate that its logic is limiting its performance. Almost 95 percent[1] of Internet traffic was represented by TCP by the beginning of the 21st century [1], that is why most of the Internet architecture is built around it, we could say almost hard-coded for it. This fact is limiting other, possibly better schemes of gaining traction because they, as we will show, cannot be more aggressive than TCP simply because of this status quo. We will present some progressive new protocols and congestion control mechanisms featuring interesting technical approaches that could be incorporated into the design of the Universal Transport Protocol.

[1] Depends on the study, but generally number range between 80 to 95 percent.

1.1 Hypertext Transfer Protocol

Hypertext Transfer Protocol is the foundation of data communication for the World Wide Web. It functions as a request-response protocol in the client-server computing model and is designed to permit intermediate network elements to improve communications between clients and servers by means of caching frequently requested content. HTTP is an Application Layer protocol designed within the framework of the Internet Protocol Suite. The protocol definitions presume a reliable Transport Layer protocol for host-to-host data transfer [2]. The Transmission Control Protocol (TCP) is the dominant protocol in use for this purpose. However, HTTP has found application even with unreliable protocols, such as the User Datagram Protocol (UDP) in methods such as the Simple Service Discovery Protocol (SSDP) which is a part of the UPNP protocol stack [3, 4]. HTTP Resources are identified and located on the network by Uniform Resource Identifiers (URIs) - or, more specifically, Uniform Resource Locators (URLs) - using the http or https URI schemes. The original version of HTTP (HTTP/1.0) was revised in HTTP/1.1, instead of using a separate connection to the same server for every request-response transaction; HTTP/1.1 can reuse a connection multiple times, to download, for instance, images for a just delivered page. Hence HTTP/1.1 communications experience less latency as the establishment of TCP connections presents considerable overhead [5, 6, 7].

The core problem of HTTP is that it was not designed with latency issues in mind. Furthermore, the web pages transmitted today are significantly different from web pages 10 years ago and demand improvements to HTTP that could not have been anticipated when HTTP was developed – increase in the number of resources etc. The following are some of the features of HTTP that inhibit optimal performance [8]:

- *Single request per connection. Because HTTP can only fetch one resource at a time (HTTP pipelining helps, but still enforces only a FIFO queue), a server delay of 500 ms prevents reuse of the TCP channel for additional requests. Browsers work around this problem by using multiple connections. Since 2008, most browsers have finally moved from 2 connections per domain to 6.*
- *Exclusively client-initiated requests. In HTTP, only the client can initiate a request. Even if the server knows the client needs a resource, it has no mechanism to inform the client and must instead wait to receive a request for the resource from the client.*

- *Uncompressed request and response headers. Request headers today vary in size from ~200 bytes to over 2KB. As applications use more cookies and user agents expand features, typical header sizes of 700-800 bytes are common. For modems or ADSL connections, in which the uplink bandwidth is fairly low, this latency can be significant. Reducing the data in headers could directly improve the serialization latency to send requests.*
- *Redundant headers. In addition, several headers are repeatedly sent across requests on the same channel. However, headers such as the User-Agent, Host, and Accept* are generally static and do not need to be resent.*
- *Optional data compression. HTTP uses optional compression encodings for data. Content should always be sent in a compressed format.*

In the following text we will describe the two most interesting features of HTTP 1.1 that if widespread deployed, would dramatically improve HTTP performance. We will also analyze the impact of Bandwidth and Round Trip Time on HTTP.

1.1.1 HTTP persistent connection

HTTP persistent connection, also called HTTP keep-alive, or HTTP connection reuse, is the idea of using the same TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every single request/response pair. Under HTTP 1.0, there is no official specification for how keep-alive operates. If the browser supports keep-alive, it will add the *Connection: Keep-Alive* header to the request. If also the server supports it, it will add the same header to its response. The connection thus will not be dropped until one of the participating sites decides so. In HTTP 1.1 all connections are considered persistent unless declared otherwise [5, 9]. In contrast to HTTP 1.0, HTTP 1.1 persistent connections do not use separate keep-alive messages, they just allow multiple requests to use a single time-limited[2] connection. Recent measurements show that over 94% of HTTP connections to Google web servers use HTTP 1.1, but on average only 3.1 HTTP requests per connection are made [10, 11].

[2] Depending on the server configuration - Apache 2.0 uses by default 15 seconds, Apache 2.2 only 5 seconds.

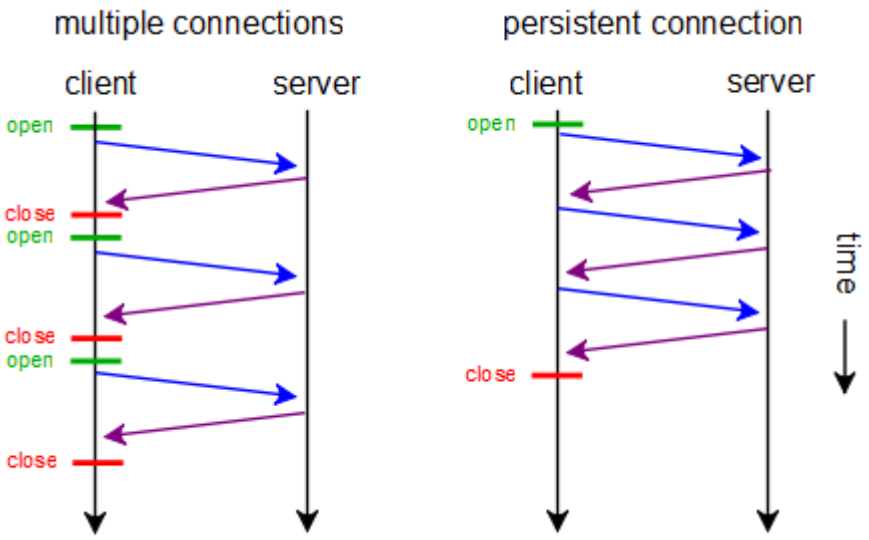

Figure 1: Persistent connection

1.1.2 HTTP pipelining

HTTP pipelining is a technique in which multiple HTTP requests are sent on a single HTTP connection without waiting for the corresponding responses [5]. Pipelining is only supported in HTTP/1.1. Pipelining dramatically improves the HTTP performance but as Web proxies and caching once improved the HTTP performance now they are limiting it. Most proxies do not support pipelining and therefore due to interoperability concerns all major browsers except of Opera have pipelining disabled by default resp. not implemented [12, 13, 14].

However, even if pipelining support would be deployed, there is still an unresolved issue: HTTP forces strict FIFO semantics for all the requests. A single slow dynamic request at the front of the queue will can create a situation known as TCP Head-of-line blocking (HOL blocking). Everyone else sharing that TCP channel will have to wait until it completes [15, 16].

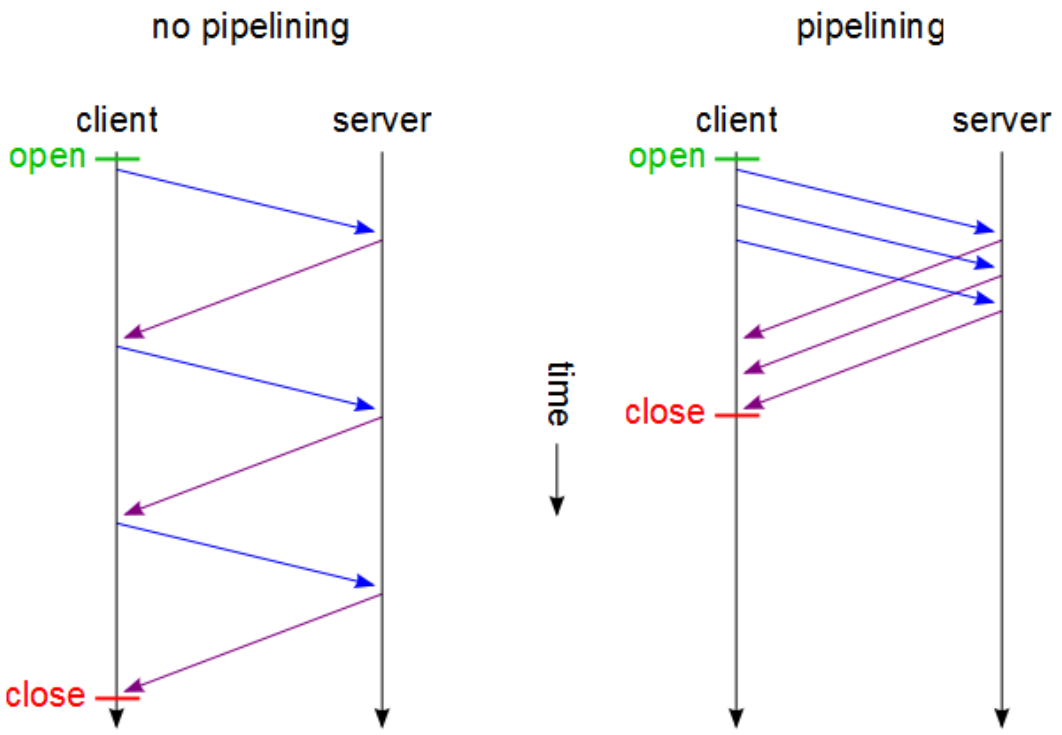

Figure 2: Pipelining

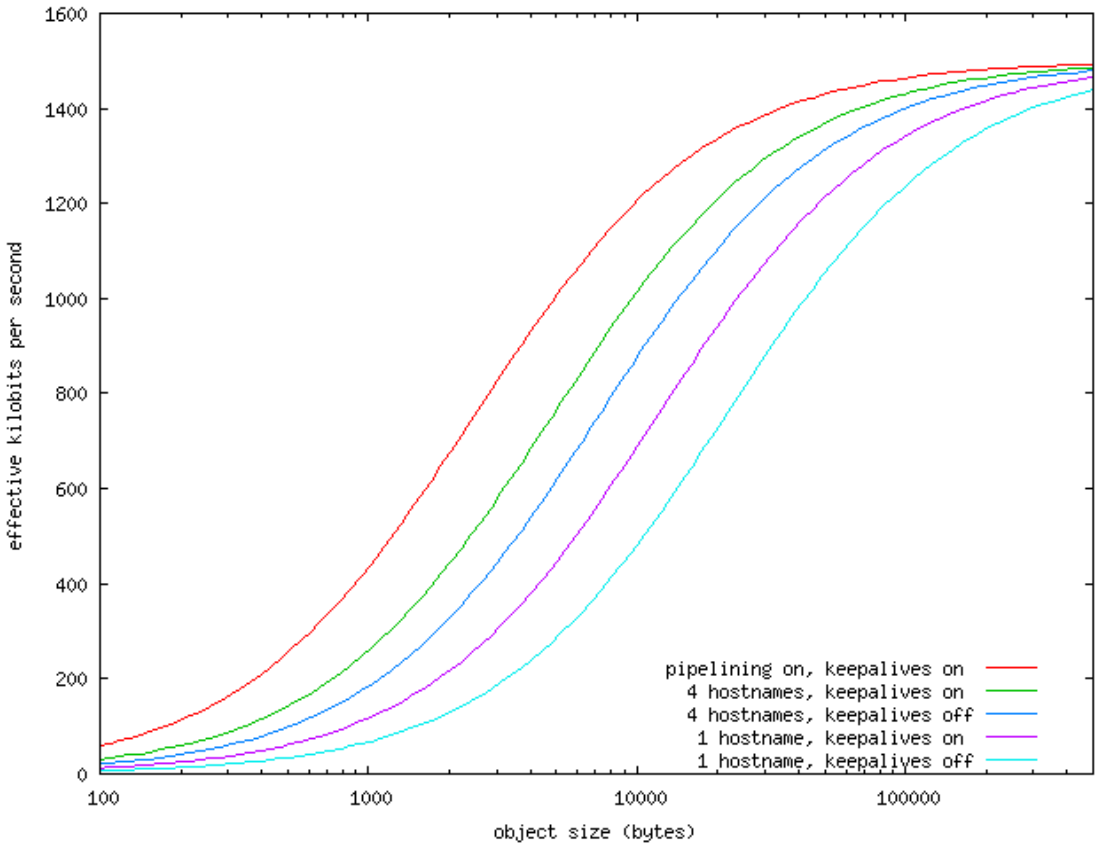

Figure 3: Effect of the number of hostnames, keepalives and pipelining using a simulated ADSL connection with 1.5Mbps downlink, 384kbps uplink, 100ms RTT and 0 %. packet drop [17].

1.1.3 Effect of Bandwidth and Round Trip Time on HTTP

Bandwidth is the limiting factor of the user experience only to a certain threshold, when crossed adding more bandwidth has only a very limited performance impact. The true limiting factor is the Round Trip Time (RTT), meaning the time it takes for a packet to get from the source to the destination and back. RTT is a physical hard limit set by the speed of light, so the only logical thing to do is to avoid doing exactly what is being done by HTTP without pipelining – request each web page resource separately. This situation is illustrated in Figures 4 and 5, adding more Bandwidth helps only until a certain point on the contrary decreasing Round Trip Time decreases Page Load Time almost linearly.

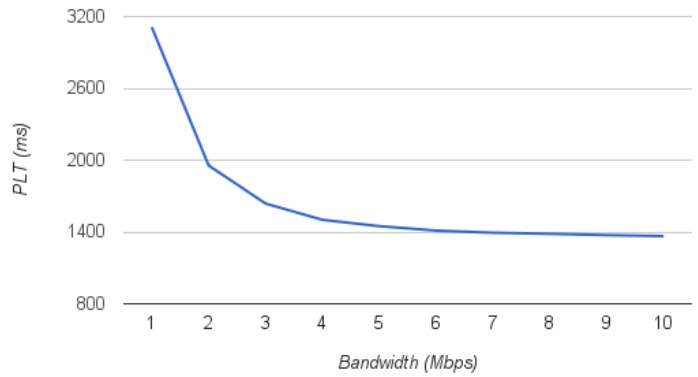

Figure 4: Effect of Bandwidth on Page Load Time with 60ms RTT

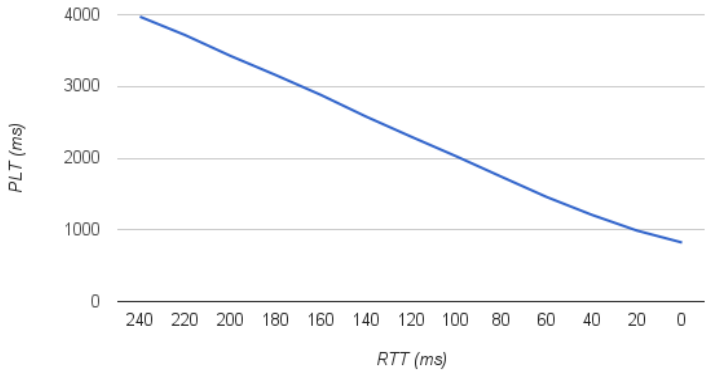

Figure 5: Effect of Round Trip Time on Page Load Time with Bandwidth fixed at 5Mbps

Hypertext Transport Protocol was designed back in the 1990s for accessing simple web pages which contained only a few resources like images or scripts. Nowadays most pages have at least 40 resources [18]. The maximal number of parallel sessions, and other parameters mentioned in the previous text create a situation known as the "waterfall loading" which is demonstrated in Figure 6.

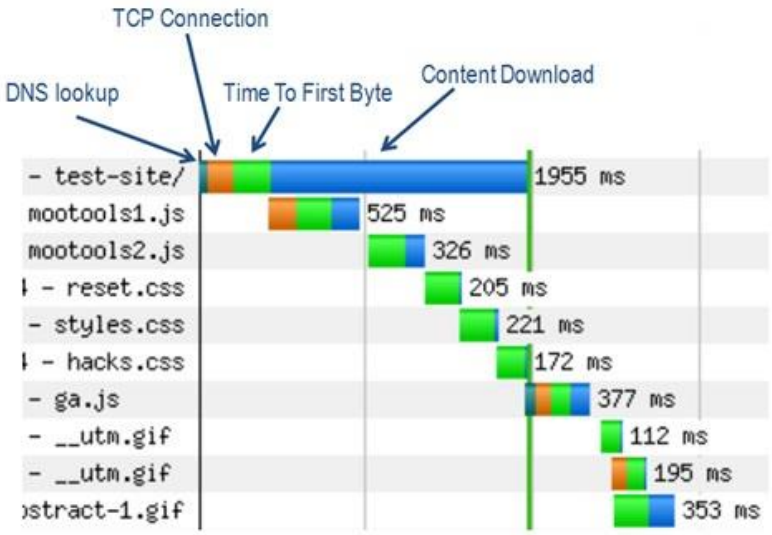

Figure 6: HTTP waterfall-shaped loading

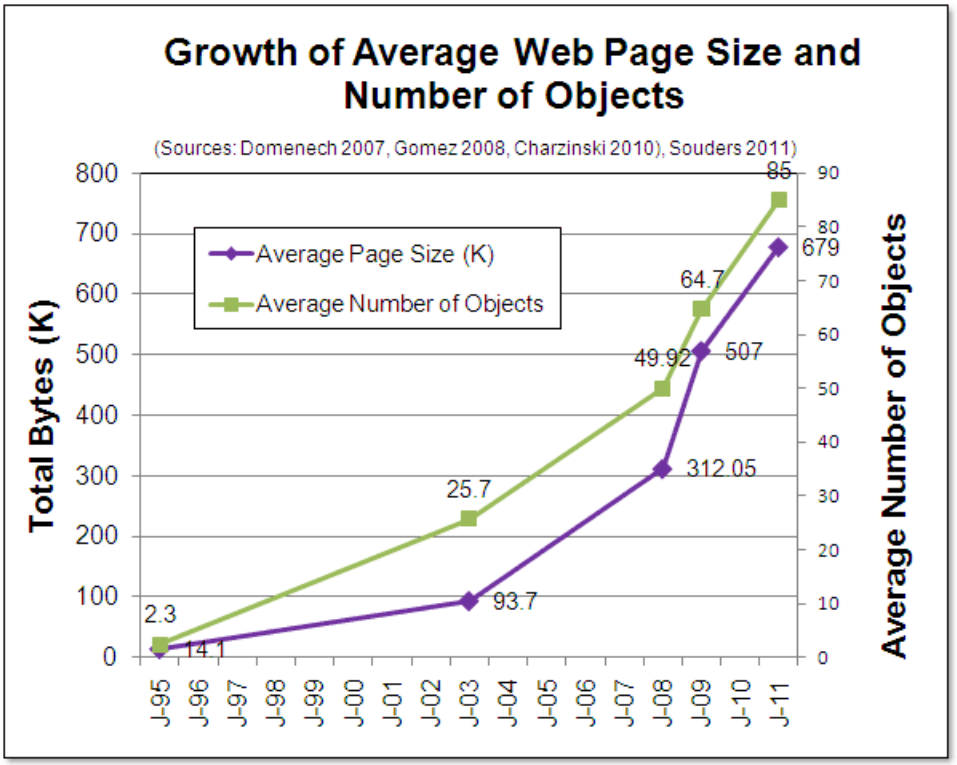

Figure 6: Growth of the average web page [19].

1.2 Google SPDY

Due to the sub-optimality of HTTP, Google decided to develop the SPDY protocol which is currently used by default on Google Chrome browsers with all Google services. SPDY enables true request pipelining (meaning without FIFO restrictions), message framing, and mandatory compression including headers, priority scheduling and even bi-directional communication. Following is a definition of its goal from the project homepage [8]:

The SPDY project defines and implements an application-layer protocol for the web which greatly reduces latency. The high-level goals for SPDY are:

- *To target a 50% reduction in page load time. Our preliminary results have come close to this target (see below).*
- *To minimize deployment complexity. SPDY uses TCP as the underlying transport layer, so requires no changes to existing networking infrastructure.*
- *To avoid the need for any changes to content by website authors. The only changes required to support SPDY are in the client user agent and web server applications.*
- *To bring together like-minded parties interested in exploring protocols as a way of solving the latency problem. We hope to develop this new protocol in partnership with the open-source community and industry specialists.*

Some specific technical goals are:

- *To allow many concurrent HTTP requests to run across a single TCP session.*
- *To reduce the bandwidth currently used by HTTP by compressing headers and eliminating unnecessary headers.*
- *To define a protocol that is easy to implement and server-efficient. We hope to reduce the complexity of HTTP by cutting down on edge cases and defining easily parsed message formats.*
- *To make SSL the underlying transport protocol, for better security and compatibility with existing network infrastructure. Although SSL does introduce a latency penalty, we believe that the long-term future of the web depends on a secure network connection. In addition, the use of SSL is necessary to ensure that communication across existing proxies is not broken.*

- *To enable the server to initiate communications with the client and push data to the client whenever possible.*

Unlike HTTP, each request in SPDY is assigned a stream ID, which allows using a single TCP channel in parallel. To support this, SPDY defines two types of frames in its binary protocol: control and data. Even if HTTP payload can be compressed, the HTTP headers are always plain-text. SPDY compresses all header data with a predefined dictionary. The support of true pipelining enables to assign priorities to each resource, for example: HTML first, JS second, images third. Bi-directional communication allows for so called server push – meaning that if the server knows which page the client is fetching, it can "hint" to the clients the resources it needs before the client even parses the HTML.

	DSL 2 Mbps downlink, 375 kbps uplink		Cable 4 Mbps downlink, 1 Mbps uplink	
	Average ms	**Speedup**	**Average ms**	**Speedup**
HTTP	3111.916		2348.188	
SPDY basic multi-domain* connection / TCP	2242.756	27.93%	1325.46	43.55%
SPDY basic single-domain* connection / TCP	1695.72	45.51%	933.836	60.23%
SPDY single-domain + server push / TCP	1671.28	46.29%	950.764	59.51%
SPDY single-domain + server hint / TCP	1608.928	48.30%	856.356	63.53%
SPDY basic single-domain / SSL	1899.744	38.95%	1099.444	53.18
SPDY single-domain + client prefetch / SSL	1781.864	42.74%	1047.308	55.40%

Table 1: SPDY speed-up [8].

The reasons why SPDY performs better when packet loss is encountered are that [8]:

- *SPDY sends ~40% fewer packets than HTTP, which means fewer packets affected by loss.*
- *SPDY uses fewer TCP connections, which means few changes to lose the SYN packet. In many TCP implementations, this delay is disproportionately expensive (up to 3 s)*
- *SPDY's more efficient use of TCP usually triggers TCP's fast retransmit instead of using retransmit timers.*

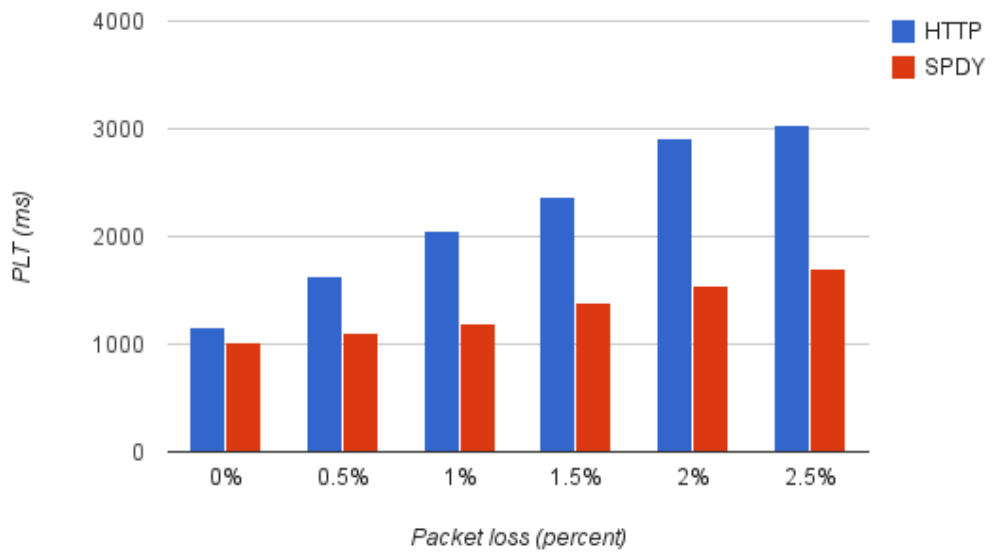

Figure 7: Packet loss effect on HTTP and SPDY, data from: [8].

We discovered that SPDY's latency savings also increased proportionally with increases in RTTs, up to a 27% speedup at 200 ms. The reason that SPDY does better as RTT goes up is because SPDY fetches all requests in parallel. If an HTTP client has 4 connections per domain, and 20 resources to fetch, it would take roughly 5 RTTs to fetch all 20 items. SPDY fetches all 20 resources in one RTT. [8]

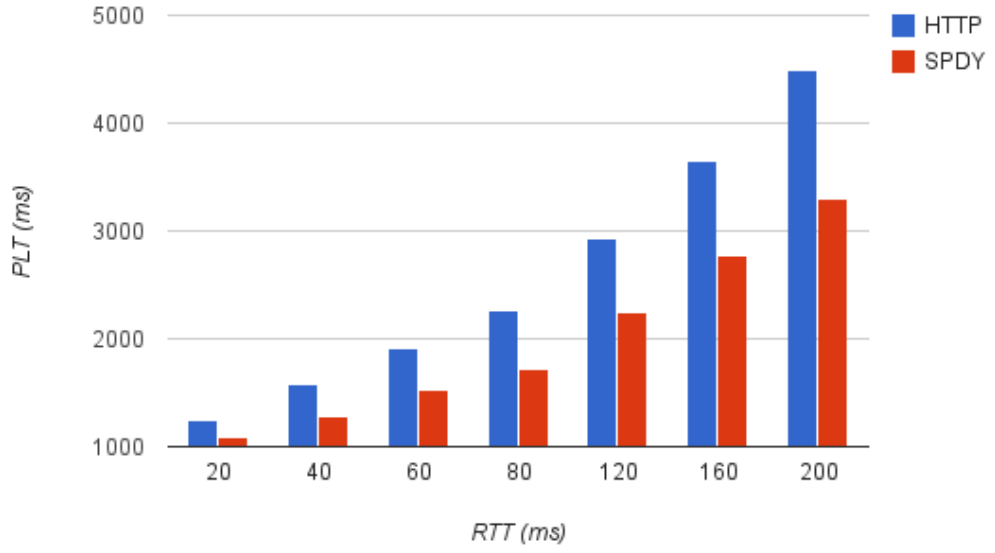

Figure 8: RTT effect on HTTP and SPDY, data from: [8].

1.3 Instant Page Load (IPL)

To better understand the problem of latency we created a “transparent” version of SPDY single stream approach. In contrast to SPDY - IPL is a network-based optimization system [20]. It uses an automated Man-in-the-middle attack to transparently intercept the client's HTTP request. Because the device performing this interception has a low RTT / large bandwidth Internet connection (preferably fiber) it can download the web page resources much faster than the client. The main idea is that most of the RTT happens at the last mile because of the technology used – WWAN, DSL. The web page is stripped at the intercepting device of all resources and a specially crafted JavaScript script is added. This modified page is transparently returned to the client as a reply to its intercepted request. Once the page gets loaded, it creates a HTML5 WebSocket connection to the IPL device. WebSocket is a part of the HTML5 standard; it is designed to provide for bi-directional, full-duplex communications channel, over a single Transmission Control Protocol (TCP) socket and has been already implemented in major web browsers and some web servers [21]. This WebSocket connection is then used to download all the resources in a single stream. IPL architecture and compassion with SPDY and HTTP is illustrated in Figure 9.

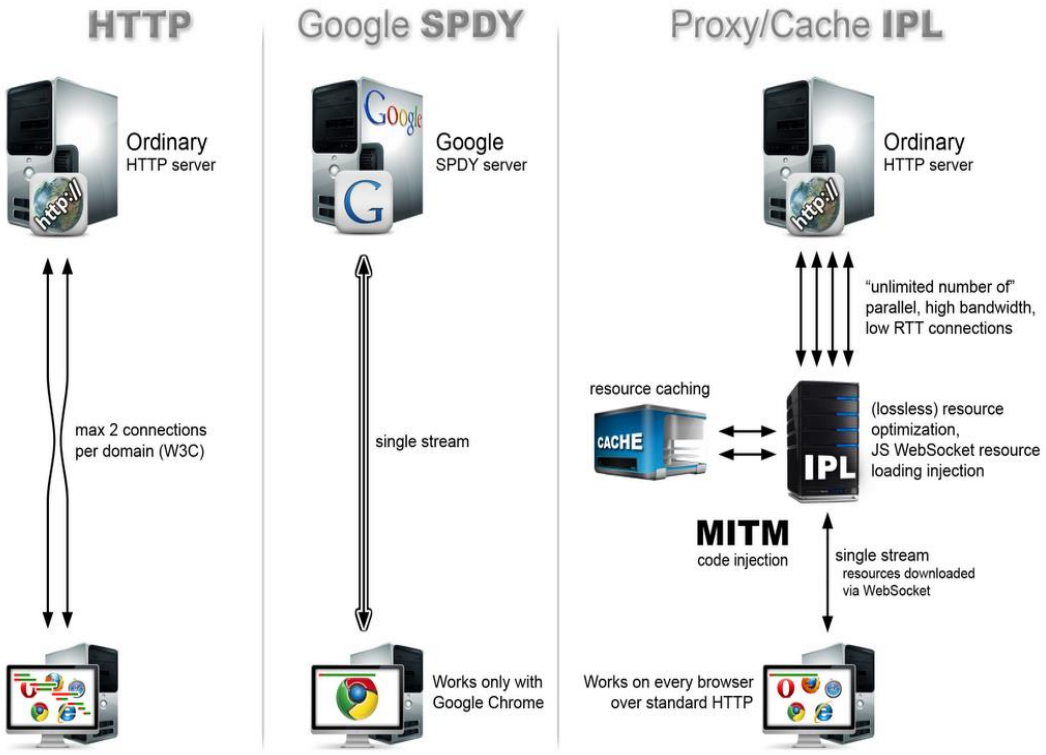

Figure 9: IPL architecture and comparison with SPDY and HTTP.

Figure 10 shows PLTs for adobe.com. Adobe`s main page had only a small number of resources (~40), loaded only from two sub-domains and was located in Europe all these factors made the difference between IPL and HTTP performance not that dramatic but still visible.

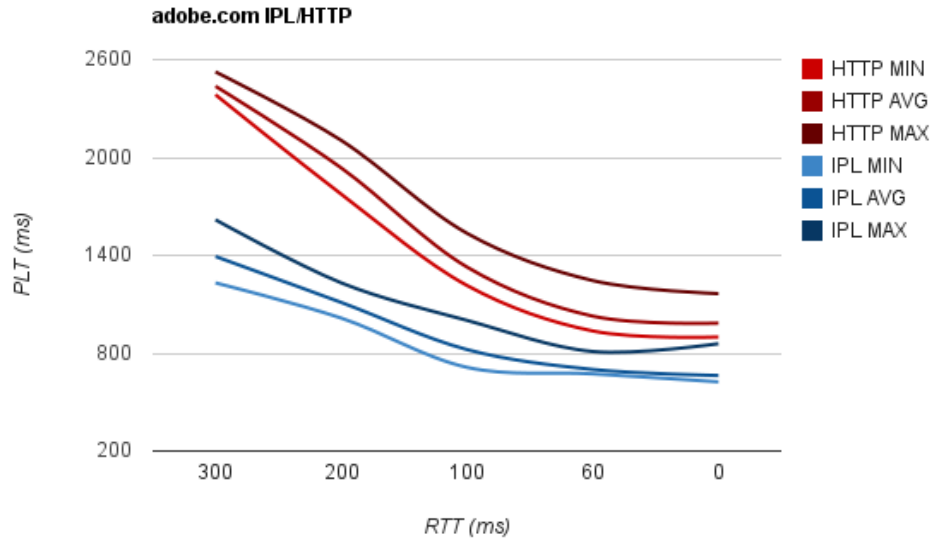

Figure 10: IPL vs. HTTP for adobe.com.

As an example of a large Website we selected BBC`s main page which has 100+ resources, loads from five domains and the server is located in Europe. Figure 11 demonstrates the massive reduction of PLT which is primarily caused by the vast amount of resources.

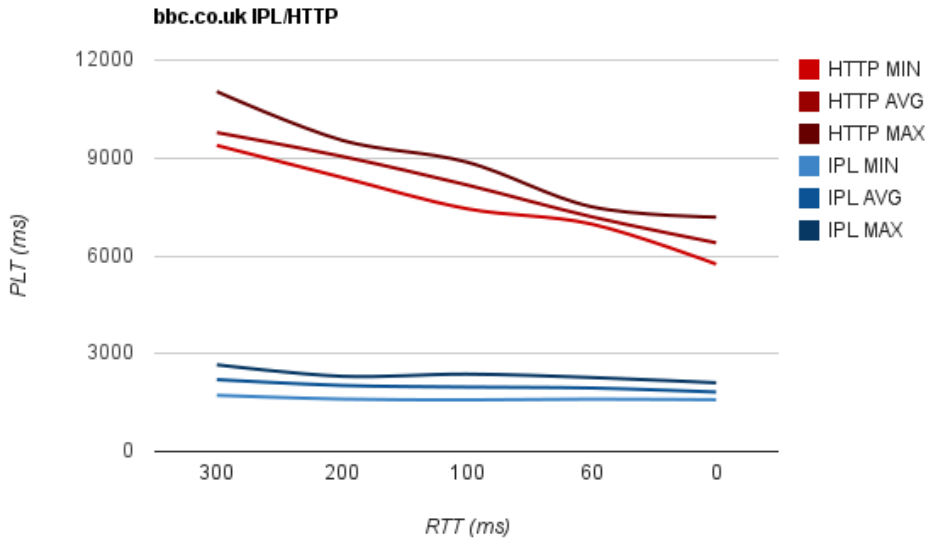

Figure 11: IPL vs. HTTP for bbc.co.uk.

The last example shows the PLT decrease for an average US Website (~70 resources, loading from 3 sub-domains). Because the data need to cross Atlantic further delay is added, this is why the difference between HTTP PLT and IPL is so visible in Figure 12.

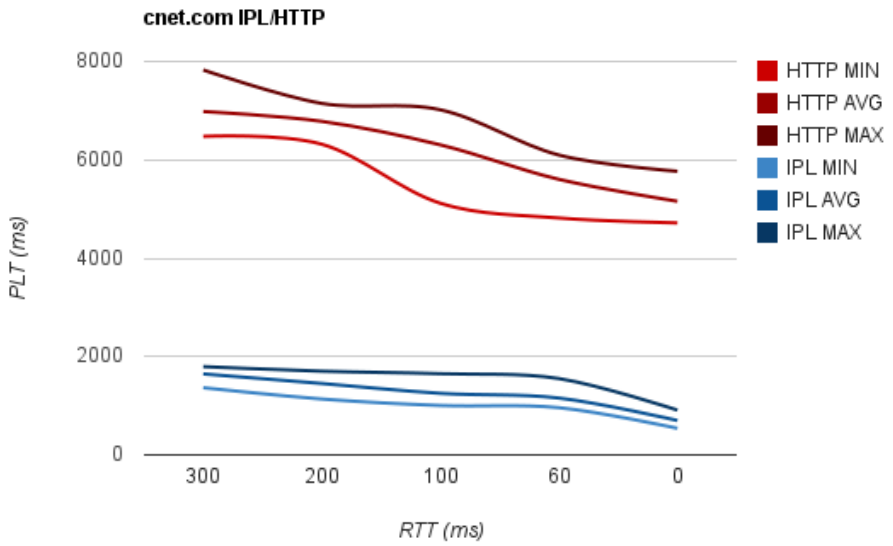

Figure 12: IPL vs. HTTP for cnet.com.

Both mentioned approaches (SPDY, IPL) are based on the same idea: Instead of downloading each resource separately they are downloaded in a single TCP stream thus the effect of RTT is limited when compared to HTTP. In contrast to SPDY – our presented IPL system takes a fully transparent middle-box approach. It utilizes an automated Man-in-the-middle attack on the HTTP protocol and injects JavaScript code into the session. Thru exploiting the standard features of HTML5 compatible browsers we were able to achieve comparable PLT decrease to SPDY.

Google SPDY or our own IPL clearly demonstrate that by avoiding the effect of RTT by downloading all the Web site resources in a single stream the Page Load Time can be dramatically improved. But still it is in some sense only re-inventing the wheel by re-doing the same as HTTP pipelining was designed to do, but because of the mentioned reasons failed to achieve.

1.4 Transmission Control Protocol

Transmission Control Protocol is one of the two original components of the Internet Protocol Suite together with the Internet Protocol, therefore the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered delivery of data streams. As already mentioned, TCP builds its logic on positive acknowledgments:

- Positive Acknowledgment with Retransmission (PAR) operates by re-transmitting data at an established period of time until the receiver acknowledges the reception [22].
- Selective Acknowledgment (SACK) – the receiver explicitly lists which segments in a stream are acknowledged, it is an optional feature that was not a part of the original TCP [23].
- Cumulative Acknowledgment is used by the TCP sliding window mechanism; the receiver acknowledges that it correctly received a segment which implicitly informs the sender that the previous packets were also received correctly. Cumulative ACKs were introduced as part of TCP Extensions for High Performance [24].
- Partial Acknowledgments (PA) - *In the case of multiple packets dropped from a single window of data, the first new information available to the sender comes when the sender receives an acknowledgement for the retransmitted packet (that is, the packet retransmitted when Fast Retransmit was first entered). If there is a single packet drop and no reordering, then the acknowledgement for this packet will acknowledge all of the packets transmitted before Fast Retransmit was entered. However, if there are multiple packet drops, then the acknowledgement for the retransmitted packet will acknowledge some but not all of the packets transmitted before the Fast Retransmit.* [25] Introduced with TCP NewReno.

1.4.1 Packet loss detection

Acknowledgments are crucial for TCP, both its reliability and performance are dependent on them, the basic concept is that TCP assigns a sequence number to each transmitted packet and expects a positive acknowledgment for it. In the absence of an ACK from the receiver, the sender automatically assumes that the packet was lost in the transmission and retransmits it. This process is called loss recovery. TCP is a sliding-

window based protocol in which the size of the window determines the maximum number of outstanding unacknowledged packets in the network – also called in-flight. While TCP is waiting for an ACK for a lost packet, due to the window limit it does not send out new data, thus effectively stalling the data transfer. TCP assumes all loses are congestion events and in order to reduce the load on the network reduces the window size. Such behavior degrades the throughput and as we will show it is not always necessary.

Senders detect segment losses using two types of mechanisms that rely on the ACK stream: retransmission timeouts (RTOs) [26, 27, 28], and fast retransmit/recovery [28].

RTOs: *TCP sets a timer to expire after an RTO-amount of time when a segment is transmitted; if the ACK for a segment is not received before the timer expires, the sender concludes that the segment was lost. The value of RTO is determined using the relation:* $RTO = m * srtt + k * rttvar$*, where srtt is a moving average of the connection round-trip time (RTT), computed as:* $srtt = (1 - b) * srtt + b * rtt$*; rttvar is a moving average of the variability in RTT, computed as:* $rttvar = (1 - a) * rttvar + a * |srtt - rtt|$*; a, b, m, k, are positive constants and* $a, b \in [0, 1]$*. The value of RTO increases with m and k, whereas a and b determine the weight given to history when RTT is quite variable. The actual value of the RTO timer is set to a predetermined value, minRTO, if the value computed above is smaller than minRTO. The above formulation is intended to compute an RTO that is greater than the current RTT, in order to avoid inferring loss of segments for which the ACK is merely delayed. Since RTT variability can be high, the value of RTO can be high, especially with the recommended settings for the five parameters, a, b, m, k, minRTO - RTO-based loss detection can, therefore, be time-consuming.* [29]

FR/R: *FR/Rs are a faster means of detecting losses—if a segment is lost, segments with higher sequence numbers trigger duplicate (cumulative) ACKs for its preceding segment. Hence, when a sender receives duplicate ACKs for a segment, it can conclude that the next higher segment was lost. However, reordering of segments by the network can also trigger the generation of duplicate ACKs. In order to avoid erroneously inferring loss in such cases, TCP senders usually wait for* $D > 1$ *duplicate ACKs before concluding a segment loss.* [29]

Parameter	Linux	Windows	FreeBSD	Solaris
Timer granularity	10ms	100ms	10ms	10ms
Initial RTO (s)	3	3	3	3.375
$minRTO$ (ms)	200	200	1200	400
a	0.25	0.25	0.25	0.25
b	0.125	0.125	0.125	0.125
m	1	1	1	1.25
k	4	4	4	4
D	3	2	3	3
RTO	srtt + vartt	srtt + 4*rttvar	srtt+ 4*rttvar	1.25*srtt + 4*rttvar

Table 2: Values of key parameters in different TCP stacks [29].

Table 2 shows default TCP stack parameters of the most popular Operating Systems. Some important things to notice are the initial RTO which defines how long will the stack wait until timeout for the SYN packet in the three way handshake. Thus if the initial SYN is lost the OS will wait for 3 seconds (3.3375 seconds for Solaris). SYN packet loses are therefore the most costly ones. The RTO can never be smaller than minRTO, but even in today's WWAN networks the RTT can go below 100ms and in the upcoming 4G networks it can be even below 50ms. What can then happen is that a series of small RTTs brings the RTO down to minRTO, from that point any jitter, pushing RTT above certain dynamically calculated value larger than minRTO, will be considered a packet drop even if it is not. To explore the real-life RTT behavior of modern WWAN networks we did the following measurements on a HSDPA/HSUPA network while moving (walking) thru a highly populated area during midday. First graph shows a ping test without any additional load, one ICMP request per second:

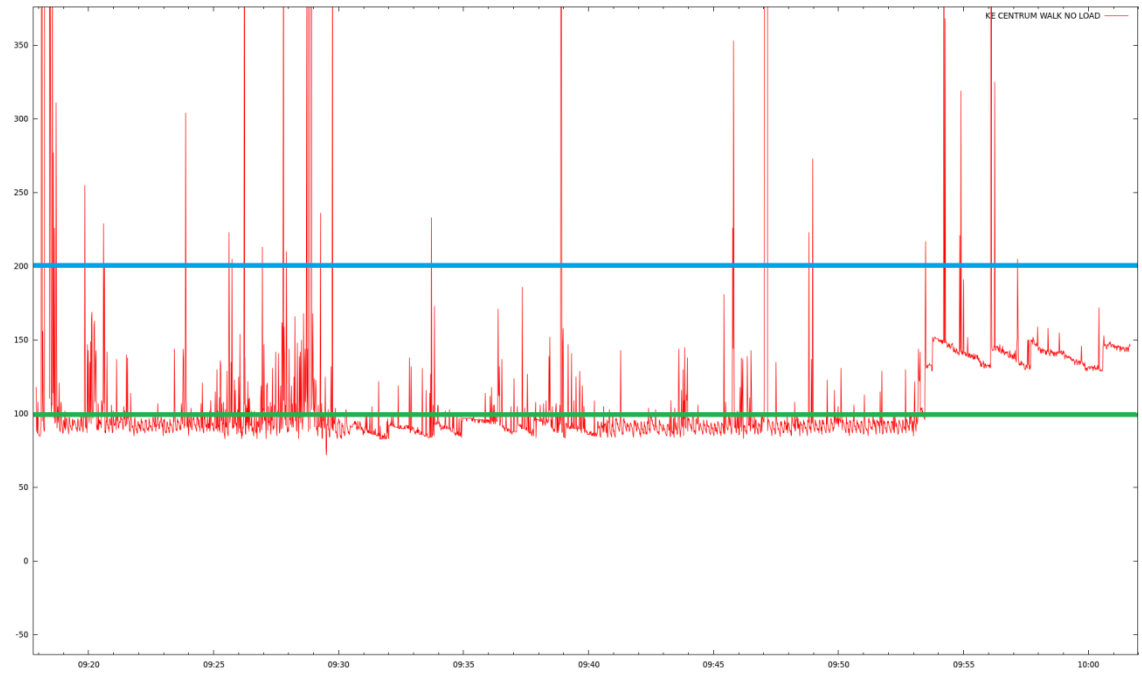

Figure 13: RTT ping measurement, green line marks 100ms, and blue line marks 200ms.

Even without any load spikes above 200ms can still be observed. To simulate a heavy usage scenario, for example a FTP upload, we used flood ping with 500 bytes of payload (maximum allowed by the network) and kept it running for more than 3 hours while continually moving at walking speed.

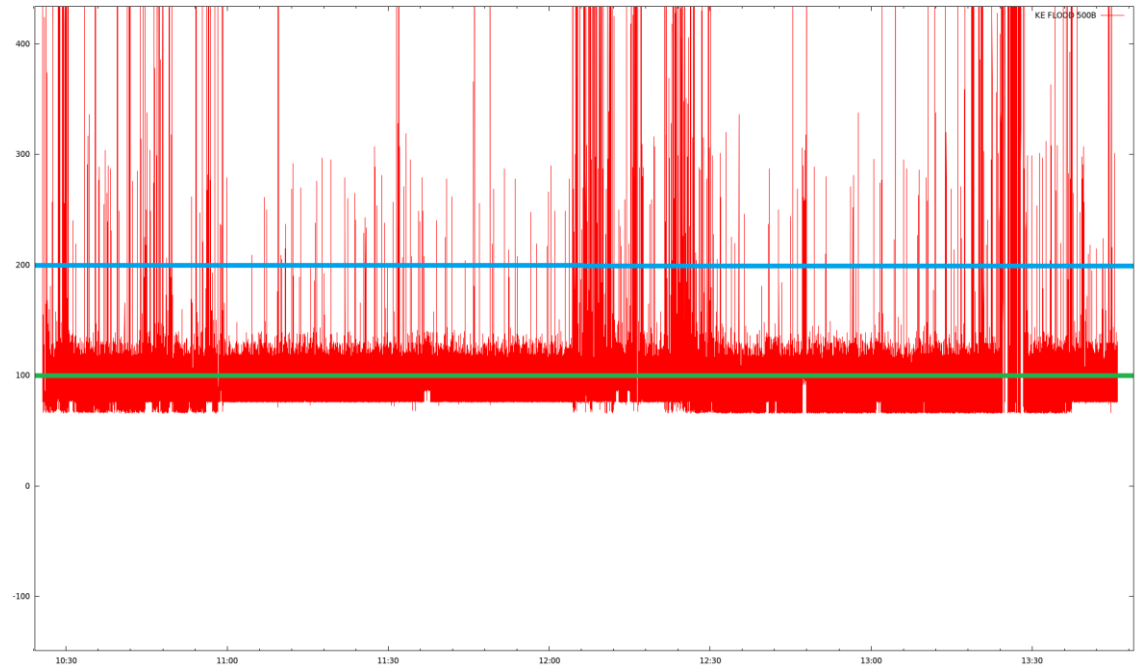

Figure 14: Flood ping, 500 bytes, green line marks 100ms, and blue line marks 200ms.

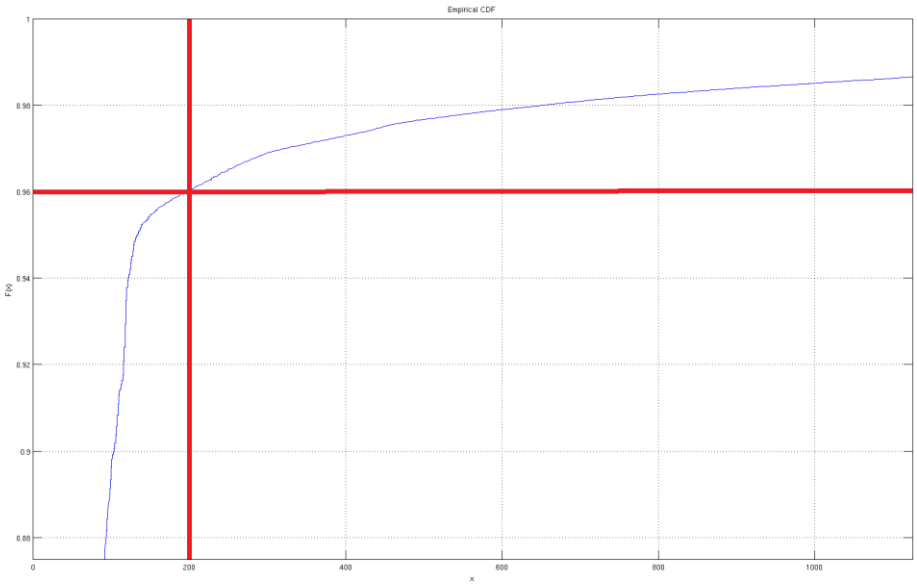

Figure 15: Empirical Cumulative Distribution Function of flood ping RTTs from Figure 14, 4 percent of all RTTs are larger than 200ms. Marked on X axis is 200ms, marked on Y axis 0.96.

Most of the spikes shown in Figure 14 would in fact be understood as RTO events by TCP by the mentioned equations even that they are not real RTO events but so-called spurious RTOs. Therefore, we can estimate that there could be in worst-case up to 4 percent packet drop during the test, if it would be a TCP connection instead of a flood ping. As we will show later, such level of packet drop would dramatically limit the TCP performance. It is well known that TCP has been optimized for wired networks. However, wireless links are known to experience sporadic and usually temporary losses due to fading, shadowing, hand off, and other radio effects, that cannot be considered congestion. Extensive research has been done on the subject of how to combat these harmful effects. Suggested solutions can be categorized as end-to-end solutions (which require modifications at the client or server) [30], link layer solutions (such as RLP in cellular networks) [31], or proxy based solutions (which require some changes in the network without modifying end nodes.) [30, 32].

Approach	Mechanism	Advantages	Disadvantages
Indirect TCP	splits TCP connection into two connections	isolation of wireless link, simple	loss of TCP semantics, higher latency at handover
Snooping TCP	“snoops” data and acknowledgements, local retransmission	transparent for end-to-end connection, MAC integration possible	problematic with encryption, bad isolation of wireless link
M-TCP	splits TCP connection, chokes sender via window size	Maintains end-to-end semantics, handles long term and frequent disconnections	Bad isolation of wireless link, processing overhead due to bandwidth management
Fast retransmit/ fast recovery	avoids slow-start after roaming	simple and efficient	mixed layers, not transparent
Transmission/ time-out freezing	freezes TCP state at disconnect, resumes after reconnection	independent of content or encryption, works for longer interrupts	changes in TCP required, MAC dependant
Selective retransmission	retransmit only lost data	very efficient	slightly more complex receiver software, more buffer needed
Transaction oriented TCP	combine connection setup/release and data transmission	Efficient for certain applications	changes in TCP required, not transparent

Figure 16: Comparison of different approaches to "mobile" TCP [33].

It is important to note that in modern WWAN networks packet drops due to radio effects are rare because the air interface usually implements some kind of Automatic Repeat Request (ARQ) protocol. Radio Link Protocol (RLP) is an example of such ARQ fragmentation protocol, if it detects packet losses it performs retransmission to bring down the packet loss down to 0.01 percent or even 0.0001 percent resp. the number of retransmissions retries is limited. UMTS, LTE or WiMAX use Hybrid ARQ (HARQ) that combines traditional ARQ with FEC to limit the amount of data that needs to be retransmitted because every retransmission adds delay [31, 34, 35, 36].

The presence of ARQ layer considerably improves TCP throughput. The price for this decreased loss rate is that ARQ adds latency in processing ARQ frames so that RTT is higher. Due to the strict link layer ordering, the communication end points observe a pause in packet delivery that can cause a spurious TCP RTO instead of getting out-of-order packets that could result in a false fast retransmit instead. Either way, interaction between TCP retransmission mechanisms and link-layer recovery can cause poor performance [37].

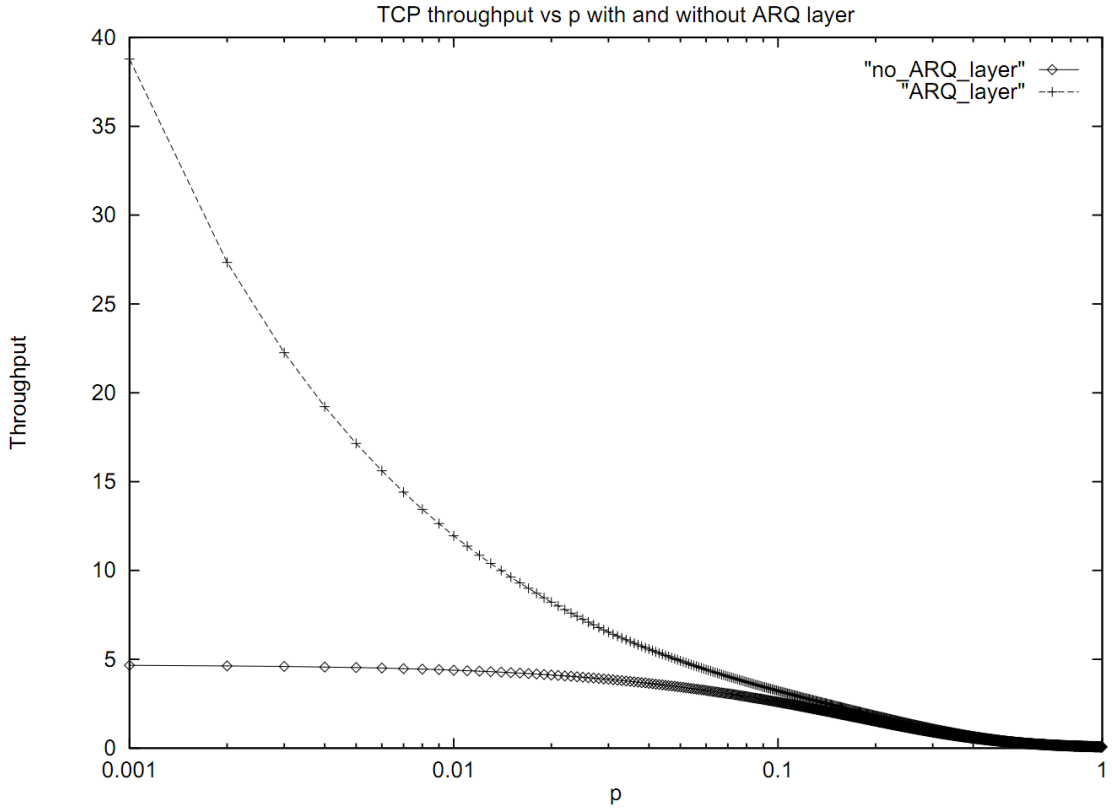

Figure 17: Effect of the ARQ layer on TCP performance, where p is the packet loss rate [38].

There are several proposed solutions for detecting spurious RTOs. The Eifel algorithm [39, 40] suggests that the TCP sender indicates whether a segment is transmitted for the first time or is a retransmission. When this information is sent back in the acknowledgement, the sender can determine if the original segment arrived at the receiver and declare the retransmission either correct or spurious. Knowing this, the sender either retransmits the unacknowledged segments in the conventional way, assuming the RTO was triggered by a segment loss e.g. it was not spurious, or if it was spurious - reverts the recent changes on the congestion control parameters and continues with transmitting new data. The latter alternative is likely to be the correct action to take when the original segment was acknowledged after the RTO, clearly indicating that the RTO was spurious. The Duplicate SACK (DSACK) enhancement [41, 42] suggests using the first SACK block to indicate arrival of duplicate segments. This alternative has its benefits over the Eifel algorithm, because the SACK option is more widely deployed than the TCP timestamps[3] required by Eifel, and the SACK blocks are appended to the TCP headers only when necessary. However, if the spurious RTO was triggered by a sudden delay and caused the unnecessary retransmissions, the

[3] MS Windows operating systems don't use TCP timestamps.

acknowledgements with the DSACK information will arrive at the sender after the acknowledgements of the original segments. Therefore, DSACK cannot avoid the unnecessary retransmissions following the spurious RTO, it can only revert the congestion control parameters to the state preceding the spurious retransmission. The most interesting proposal seems to be F-RTO [43] as it needs only modification at the sender[4]. *The guideline behind F-RTO is that an RTO either indicates a loss, or it is caused by an excessive delay in packet delivery while there still are outstanding segments in flight. If the RTO was due to delay, that is, the RTO was spurious, acknowledgements for non-retransmitted segments sent before the RTO should arrive at the sender after the RTO occurred. If no such segments arrive, the RTO is concluded to be non-spurious and the conventional RTO recovery with go-back-N retransmissions should take place at the TCP sender.* [44]

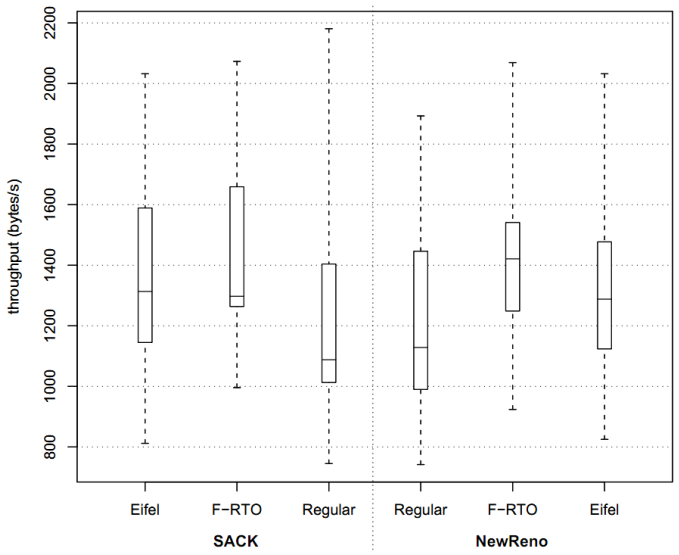

Figure 18: TCP performance with different variants with excessive delays on the link [44]

[4] DSACK needs both communicating nodes to support it.

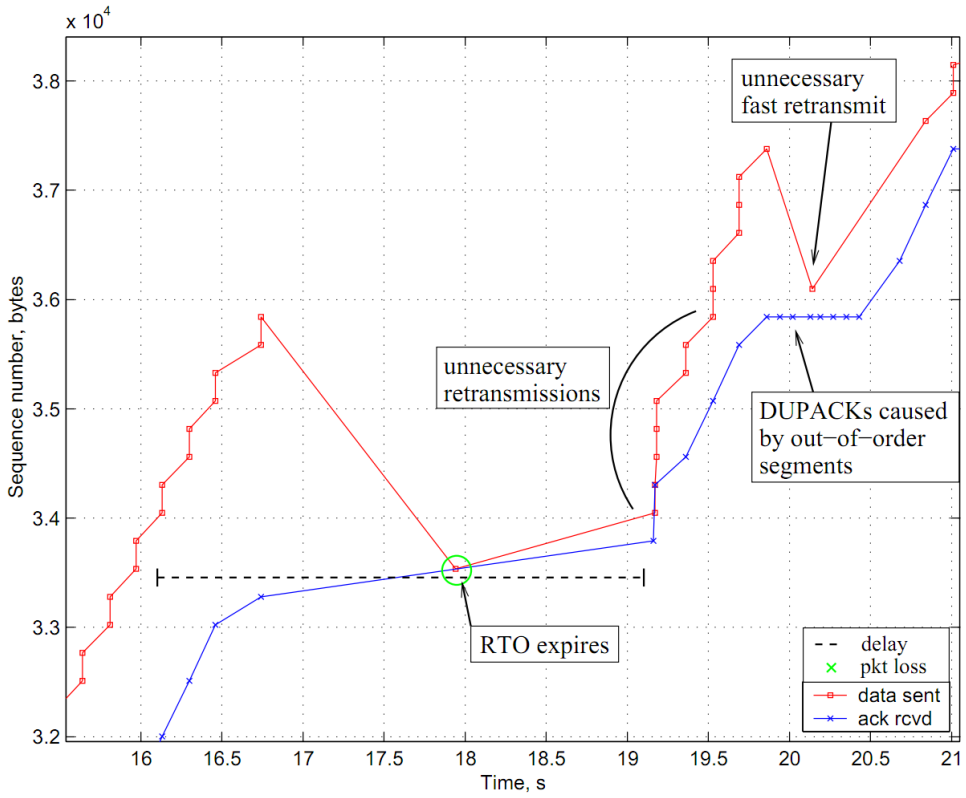

Figure 19: Conventional RTO recovery [44].

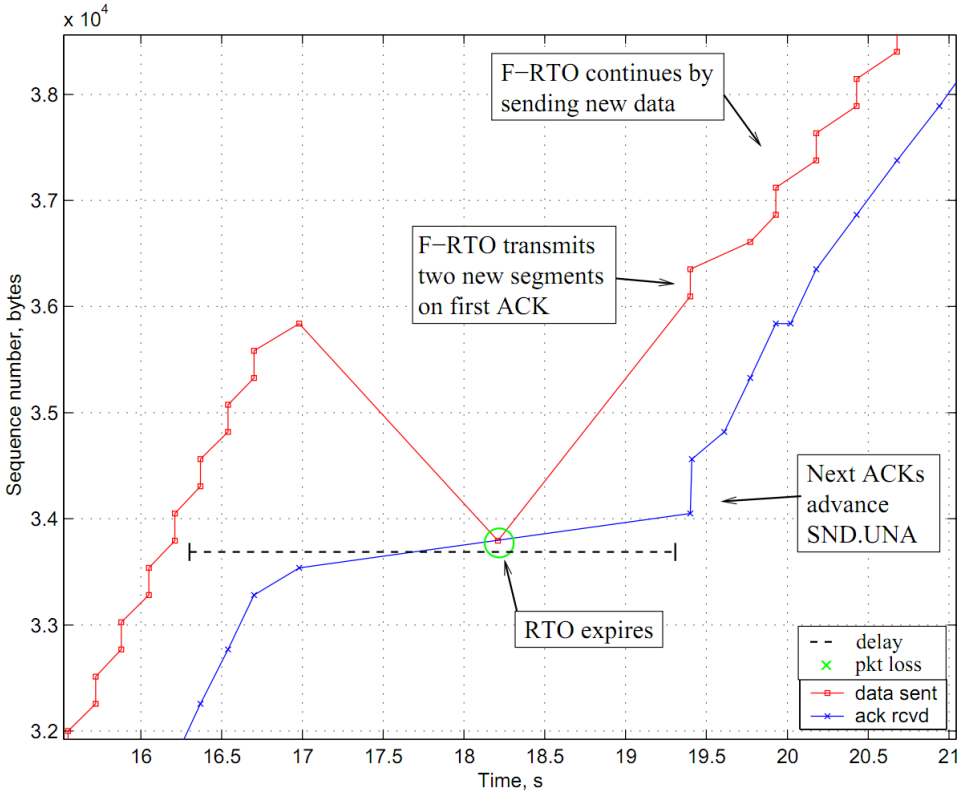

Figure 20: F-RTO recovery [44].

1.4.2 TCP Congestion and Receiver Windows

TCP performance is primarily controlled by two factors: the congestion and the receive window. The first one tries not to exceed the capacity of the network (congestion control) and the second one tries not to exceed the capacity of the receiver to process data (flow control). Each TCP segment contains the current value of the receive window. If for example a sender receives an ACK which acknowledges byte 4000 and specifies a receive window of 10000 (bytes), the sender will not send packets after byte 14000, even if the congestion window allows it [28].

TCP window scaling [24] is needed for efficient transfer when the bandwidth-delay product is greater than the initial receive window. For example the bandwidth-delay product of a HSDPA connection with 6Mbps and 100ms RTT would be calculated as:

$$B \times D = 6 \times 10^6 \text{ b/s} \times 10^{-1} \text{ s} = 6 \times 10^5 \text{ b, or 600 kb, or 75 kiB.}$$

Without increasing the default 64KB buffer size (assuming it is not Linux, see Table 3) the maximum speed would be utilized only to (64/75) = 85.3 percent. This way the receive window size may be increased up to a maximum value of 1,073,725,440 bytes; almost 1GB. Since Windows Vista practically most new operating systems have window scaling implemented and enabled by default, thus the receiver window should not be a limiting factor of TCP performance [45].

Operating System	Initial Receive Window
Linux 2.6.32	3*MSS (usually 5,840 bytes)
Linux 3.0.0	10*MSS (usually 14,600 bytes)
Windows NT 5.1 and up (XP and newer)	65,535 bytes
Mac OS X 10.5 and up (Leopard and newer)	65,535 bytes

Table 3: Initial Receive Windows for different Operating Systems

The congestion window is maintained by the sender and is calculated by estimating how much congestion there is between the sender and receiver. By default on connection setup the Slow-start phase is initiated, the congestion window is set to the 1 or 2 Maximum Segment Size (MSS)[5]. Further variance is defined by the Additive Increase/Multiplicative Decrease (AIMD) approach [46]. Let *w(t)* be the sending rate

[5] The general trend is to increase the initial cwnd, for example Linux currently uses initial cwnd 10

(e.g. the congestion window) during time slot *t*, *a (a > 0)* be the additive increase factor, and *b (0 < b < 1)* be the multiplicative decrease factor.

$$w(t+1) = \begin{cases} w(t) + a & \text{if congestion is not detected} \\ w(t) \times b & \text{if congestion is detected} \end{cases}$$

Formula 1: AIMD

The additive increase factor *a* is typically one MSS per RTT, and the multiplicative decrease factor *b* is typically 1/2. Since the receiver typically sends an ACK for every two segments due to delayed ACK, this behavior effectively doubles the window size each round trip of the network [47, 48]. The process continues until the congestion window size (cwnd) reaches the size of the receiver's advertised window or until a loss occurs, on which:

1. Congestion window is reset to 1 MSS
2. sshtresh is set to half the window size before packet loss was detected
3. Slow Start (exponential growth phase) is initiated
4. when cwnd >= sshtresh TCP goes into congestion avoidance mode

In congestion avoidance mode every ACK increases the congestion window by MSS*MSS/cwnd, resulting in a linear increase of cwnd [28].

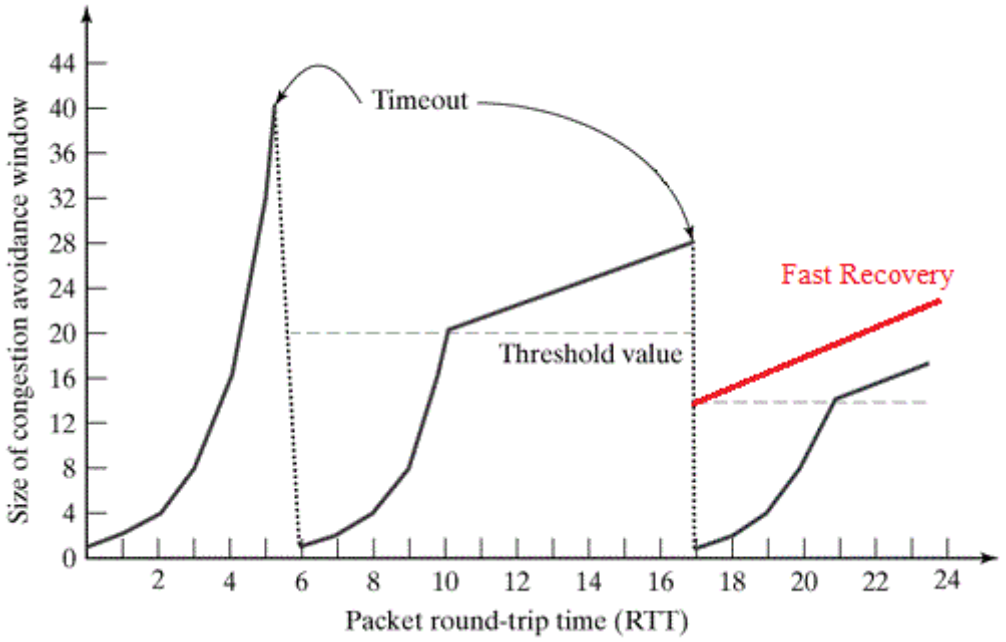

Figure 21: TCP Slow Start, Slow Start with congestion avoidance and Fast Recovery (TCP Reno)

Figure 21 illustrates the behavior of TCP Slow Start and compares it to Fast Retransmit/Recovery which would be triggered if duplicate ACKs would be detected

during the congestion avoidance mode. In FR/R the presumably dropped packet is retransmitted, the congestion window size is reduced to sshtresh and congestion avoidance mode is initiated. In [29] it was shown that "*50-80% of TCP loss detections are triggered by the costly RTO rather than FR/R. The main reason for the prominence of RTO is the lack of enough packets in flight to trigger a FR/R based detection. As expected RTO based detection is more time consuming that FR/R based detection. The time required for FR/R based detection is OS agnostic and is approximately equal to 1-2 RTTs in most of the cases. The median RTO based detection time for Windows and Linux is 4 RTTs while for Solaris and FreeBSD it is larger than 20 RTTs. RTO based detection for Windows and Linux differ significantly in its tail. While only 10% of Linux connections take longer than 10 RTT, 25% of Windows connections take longer than 10 RTTs"*

Some variants as for example the WideArea TCP (WATCP) try to optimize TCP performance by tweaking TCP parameters - reducing minRTO and initial RTO, transmitting critical packets multiple times (SYN-ACK) and by increasing the initial congestion window from the usual 1-2 MSS to as much as 16 MSS [49, 50]. Increasing the ICW results in reducing the amount of RTTs needed to transmit the data, thus improving the performance over WANs. In contrast to classic TCP where the congestion window slowly increases, in WATCP it aggressively starts large and decreases only when losses are detected. We speculate that such aggressive behavior when used with inefficient protocols as HTTP without pipelining might on current sites with dozens of resources create network congestion issues, other researchers expressed similar suspicions [51]. Authors of WATCP seem to be aware of this as they propose its usage in conjunction with SPDY instead of HTTP [49].

Many different TCP variants try to address the under-utilization problem caused most notably by the slow growth of TCP congestion window in large bandwidth-delay product networks, where it can take long amounts of time to reach the bandwidth capacity. (e.g., FAST [52], HSTCP [53], STCP [54], HTCP [55], SQRT [56], Westwood [57], BIC [58] and CUBIC [59]). For example to reach the sending rate of 1Gbps on a connection with 200ms RTT and 1500 bytes MTU it would take TCP almost 30 minutes [54]. Therefore, most of these protocols modify the window growth function of TCP in a more scalable fashion. CUBIC TCP is used by default in Linux kernels 2.6.19 and above. Its novelty lies in not relying on the receipt of ACKs to

increase the window size, instead it depends only on the last congestion event thus, making CUBIC window increase speed independent of RTT. This in fact makes the protocol fair to flows with different RTTs.

$$W_{cubic} = C(t-K)^3 + W_{\max} \quad \text{where} \quad K = \sqrt[3]{W_{\max}\beta/C}$$

Formula 2: CUBIC congestion window, C is a scaling factor, t is the elapsed time from the last window reduction, W_{max} is the window size just before the last window reduction, β is a constant multiplication decrease factor applied for window reduction at the time of loss event [59].

Most high-speed TCP variants achieve TCP friendliness by behaving exactly the same as TCP under certain specific circumstances, this behavior is called "TCP mode". For example BIC, HSTCP and STCP enter their TCP modes when the window size is less than a small constant, typically around 30 packets. TCP friendliness will be discussed in greater depth in chapter 1.5.

$$W_{tcp} = W_{\max}\beta + 3\frac{1-\beta}{1+\beta}\frac{t}{RTT}$$

Formula 3: CUBIC TCP mode, if W_{tcp} is larger than W_{cubic} then window size is set to W_{tcp} else W_{cubic}

It was shown that if the congestion duration is less than 1/sqrt(C*RTT), or if the packet loss rate is larger than 0.36*C*RRT^3, then CUBIC is TCP friendly. With C=0.4 and RTT=100ms, when the packet loss is greater than 0.000144, CUBIC is TCP friendly. Compared to HSTCP which is TCP friendly when the loss rate is larger than 0.001, CUBIC has a larger area of TCP friendliness. When RTT is very small, CUBIC is much more TCP friendly than HSTCP regardless of loss rates. More TCP friendliness results in CUBIC being less aggressive when competing against other high speed TCP variants [59, 60].

Both CUBIC and the standard TCP SACK-based loss recovery algorithm [61] were shown to deviate from their intended behavior in the real world due to the combined effect of short flows, application stalls, burst losses, ACK loss, reordering, and stretch ACKs. *Linux suffers from excessive congestion window reductions while RFC3517 transmits large bursts under high losses, both of which harm the rest of the flow and increase Web latency.* [11] Proportional Rate Reduction (PRR) was designed to solve this problem by recovering from losses quickly, smoothly and accurately by pacing out retransmissions acorns received ACKs. During measurements it was discovered that over 6% of HTTP responses served from Google.com experience losses. Because losses severely affect user experience further investigation was performed, the

results are summarized in Figure 22 and show that responses experiencing losses last 7-10 times longer than they would under ideal conditions while responses without retransmissions are very close to it.

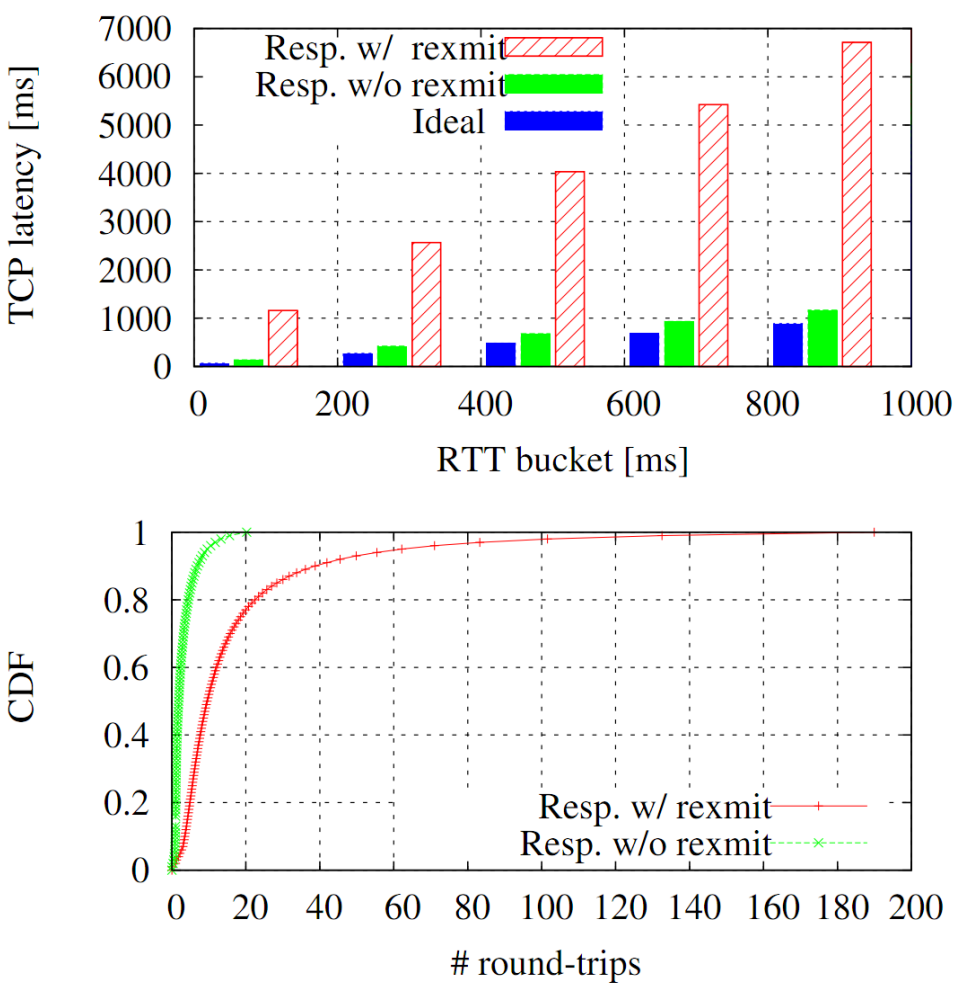

Figure 22: Top plot shows the average TCP latency of Google HTTP responses for different RTTs for ideal conditions, response without and without retransmit. Bottom CDF plot shows the number of RTTs needed to finish the request with and without response retransmission [11].

PRR has two main parts. The first part, the proportional part is active when the number of outstanding segments (pipe) is larger than ssthresh, which is typically true early during the recovery and under light losses. It gradually reduces the congestion window clocked by the incoming acknowledgments. The algorithm is patterned after rate halving, but uses a fraction that is appropriate for the target window chosen by the congestion control algorithm. For example, when operating with CUBIC congestion control, the proportional part achieves the 30% window reduction by spacing out seven new segments for every ten incoming ACKs (more precisely, for ACKs reflecting 10

segments arriving at the receiver). If pipe becomes smaller than ssthresh (such as due to excess losses or application stalls during recovery), the second part of the algorithm attempts to inhibit any further congestion window reductions. Instead it performs slow start to build the pipe back up to ssthresh subject to the availability of new data to send. [11]

PRR was shown to reduce the latency of short Web transfers by 3-10% compared to Linux recovery and proved to be smoother recovery for video traffic compared to the standard RFC3517. PRR is by default enabled on Linux kernels version 3.2 and above.

Features	RFC	Linux	Default
Initial cwnd	3390	p	10
Cong. control (NewReno)	5681	+	CUBIC
SACK	2018	+	on
D-SACK	3708	+	on
Rate-Halving		+	always on
FACK		+	on
Limited-transmit	3042	+	always on
Dynamic *dupthresh*		+	always on
RTO	2988	p	min=200ms
F-RTO	5682	+	on
Cwnd undo (Eifel)	3522	p	always on
TCP segmentation offload		+	determined by NIC
+ indicates the feature is fully implemented. **p** indicates a partially implemented feature.			

Table 4: Loss recovery related features in Linux.

1.4.3 TCP Performance Modeling

The macroscopic behavior of the TCP congestion window scaling described in the previous chapter can be theoretically modeled; we will first look at the so called Mathis et.al. formula [62] which was the first model to describe the TCP throughput as a function of connections RTT and loss rate. It assumes the connection always has data to send and that the losses (packet drop) are either periodic or random.

$$B(p) = \frac{MSS}{RTT}\frac{C}{\sqrt{p}}$$

Formula 4: Mathis et.al. formula.

The throughput B for the given loss rate of p is defined as seen in Formula 4. Where MSS is the Maximum Segment Size, RTT is the average RTT and C is a constant which depends on whether losses are periodic or random and whether delayed ACK is used or

not. In many practical situations C can be simplified to 1. It was shown that this model performs very well in low loss situations. But because the model assumes that all loses are recovered by FR/R in many cases it does not match the actual observed performance. To overcome the limitations of this model, Padhye et.al. [63] proposed a new model that models both FR/R and RTO based retransmissions.

$$B(p) = \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_o\ min(1, 3\sqrt{\frac{3bp}{8}})\ p(1+32p^2)}$$

Formula 5: Padhye et.al. formula.

Throughput B for given loss rate p is defined as seen in Formula 5 where b is the delayed ACK threshold (2 if every second packet is ACKed), RTT is the average RTT, T is the RTO. This model was shown to perform well for a wide range of loss situations. Disregarding its slight sub-optimality, because of its simplicity Mathis formula became the de-facto standard approximation method of TCP throughput for given RTT and packet loss and is widely used throughout the Internet [64, 65].

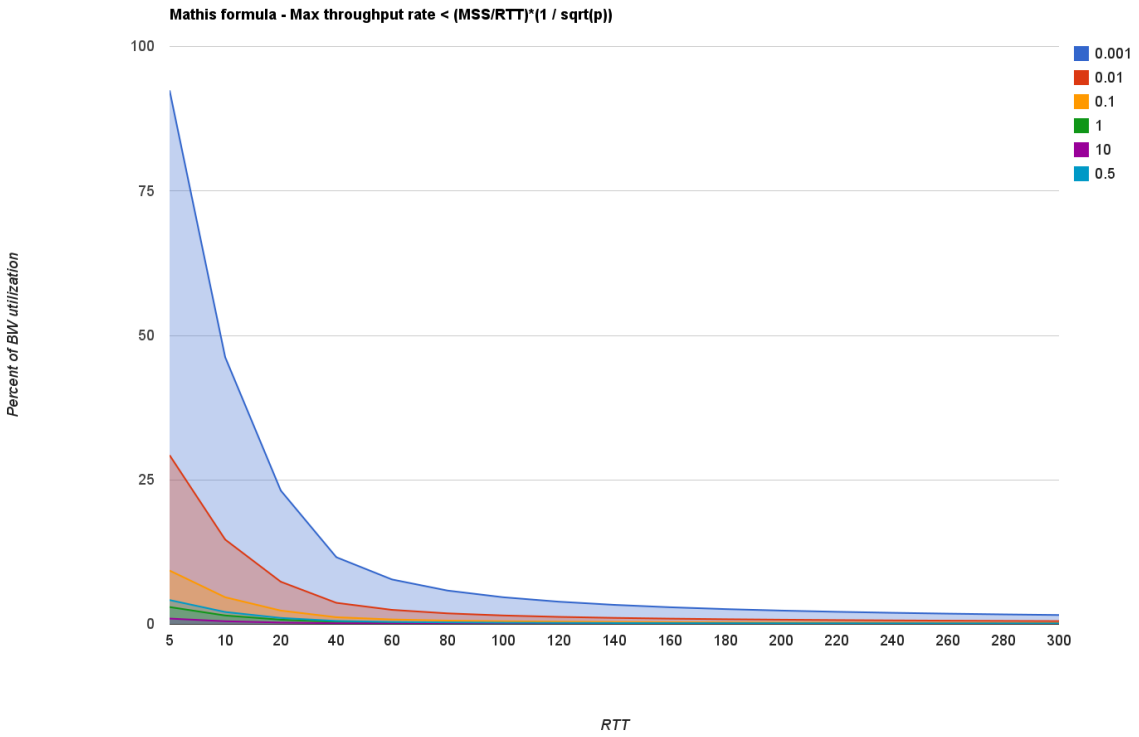

Figure 23: Mathis Formula, percent of Bandwidth utilization for given RTT (in ms) and loss rate (in percent).

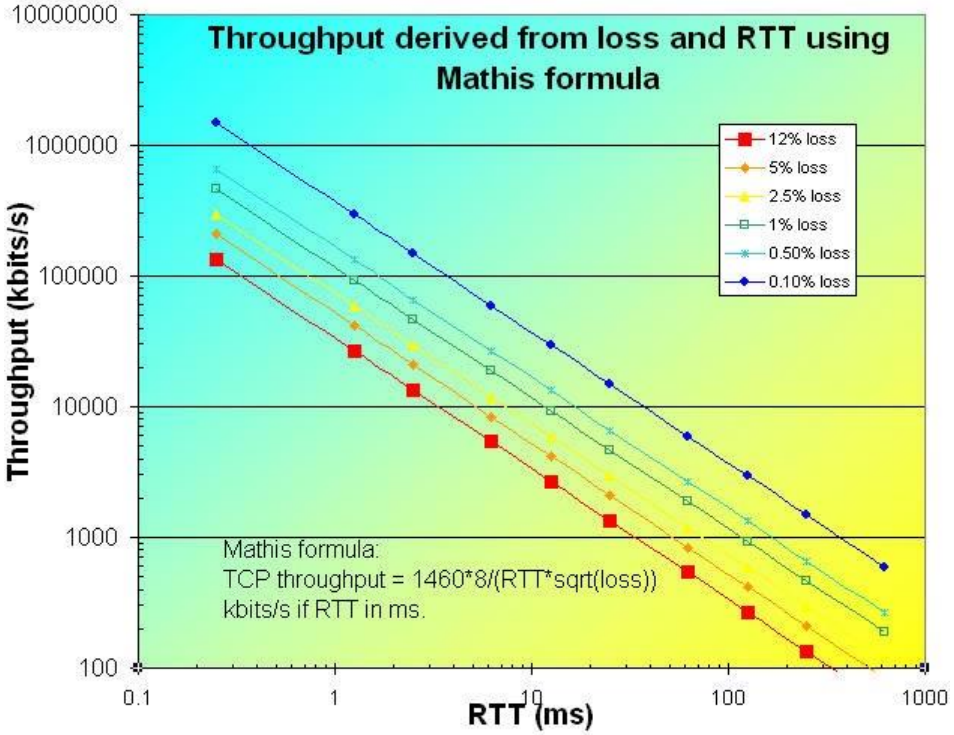

Figure 24: Mathis formula, Throughput for given RTT and loss rate [66].

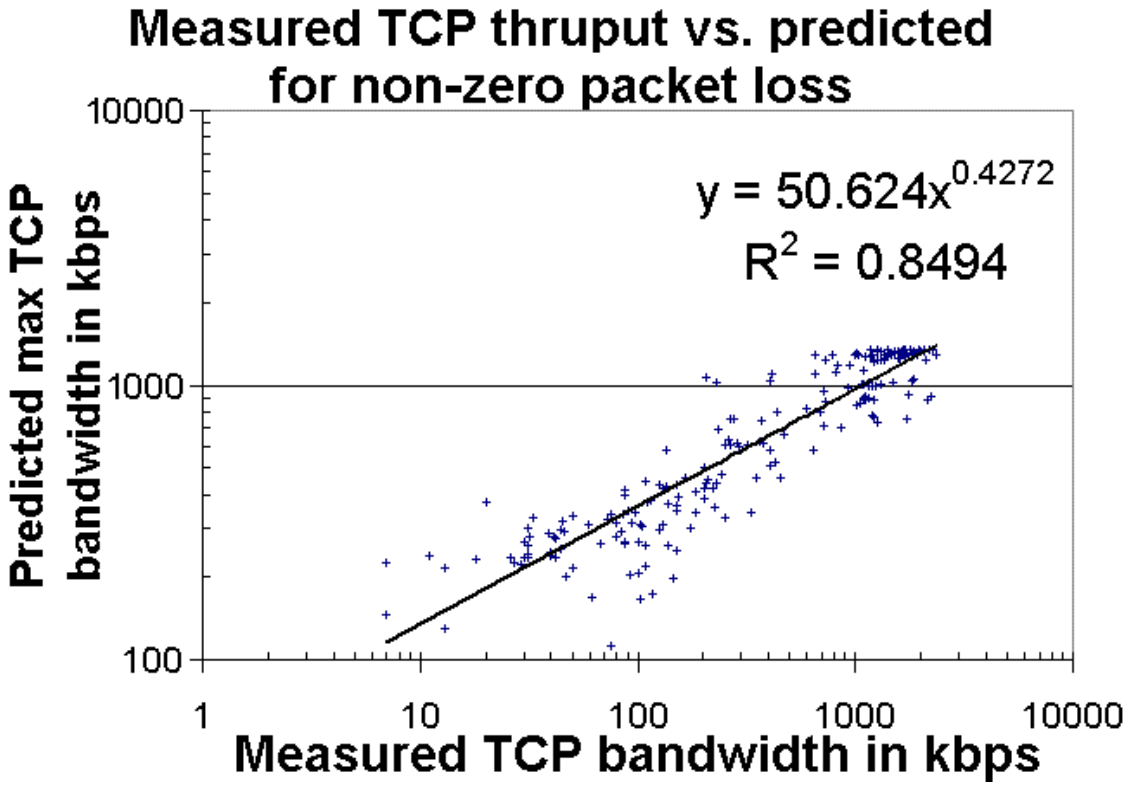

Figure 25: Mathis formula, predicted vs. measured throughput [66].

Mathis formula is interesting also because it tells us that even if the loss rate is 0, the RTT strongly affects the performance, similarly to HTTP, clearly indicating the ACK nature of the protocol. Referring back to the 500 byte flood ping example from 1.4.1,

the maximum throughput achievable by TCP with 100ms average RTT and 0 percent packet drop as stated by the Mathis formula would be 5.24Mbps, but with 4 percent packet drop merely 0.58Mbps.

The performance of TCP CUBIC can be analytically modeled using a Markovian model. But the complexity of the model makes it impractical. Still, it is important to mention that even if its congestion window scaling is independent from RTT, it was shown that its performance is still RTT dependent. Congestion loss happens when the transmission rate reaches the maximum capacity C of the bottleneck link. If we assume that the average RTT is stable, then the maximal congestion window size W is W = C * RTT. Equivalently, congestion loss occurs when window size reaches this maximum window size W [67, 68].

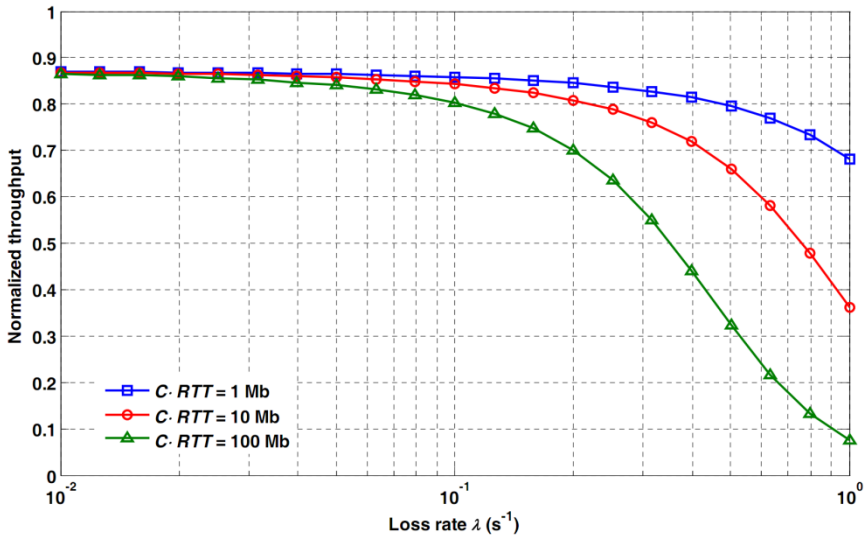

Figure 26: Normalized average TCP CUBIC throughput under different bandwidth-delay products (C * RTT) and loss rate λ [67].

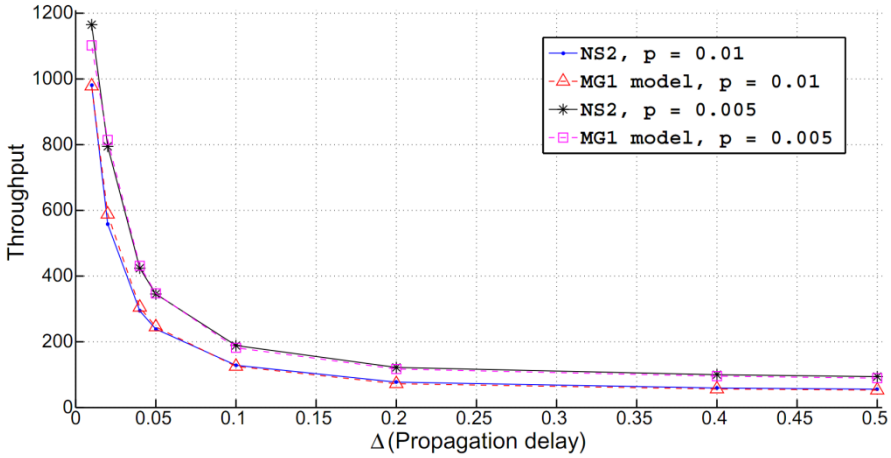

Figure 27: Throughput of TCP CUBIC (simulation and analytical model) in KBps vs. propagation delay in seconds, p is packet error rate, link capacity is 10Mbps [68].

1.5 Avoiding Congestion Collapse

Network congestion occurs when applications are sending more data than the network is able to forward, thus causing the buffers to fill up and possibly overflow. Creating a situation where an increase in data transmissions results in a proportionately smaller increase, or even a reduction, in throughput. When uncontrolled this may create a self-damaging state in which the congestion becomes so great that throughput drops to a level where little or no useful communication occurs. This state is called Congestion Collapse and it can be a stable state with the same load level that would by itself not produce congestion. The explanation for such behavior is that some aggressive/unresponsive streams can try to compensate for the packet loss that occurs as a result of congestion with retransmissions thus creating additional load.

Congestion Collapse is not just some fictive worst-case scenario, it did already occur in the history of Internet. Back in 1986, the NSFnet phase-I backbone dropped three orders of magnitude from its capacity of 32kbps to 40bps [26]. This led to the development of TCP congestion control mechanisms that are still used today. Because TCP is the Internet's most widely used protocol, its congestions control defines how every protocol is supposed to behave. This property is called TCP-Friendliness, and can be defined as: under high loss rate regions where TCP is well-behaving, the protocol must behave like TCP, and under low loss rate regions where TCP has a low utilization problem, it can use more bandwidth than TCP [69]. The fact that congestion collapse did not occur since 1980s does not mean it cannot happen again. Unresponsive streams are not only caused by TCP-unfriendly protocols but as we will show can be caused even by TCP under widely occurring circumstances and thus the risk is still here.

Despite the common belief, congestion is a normal state of a well utilized network. It becomes problematic only when unmanaged. The most commonly noticed effect of congestion is its microscopic form – jitter[6].

1.5.1 Bufferbloat

Bufferbloat is a phenomenon whereby excessive buffering of packets inside the network causes high latency and jitter, most times reducing the overall network throughput. Network equipment manufacturers use to set the buffer size statically for the fastest possibly achievable speed resp. try to avoid dropping packets at all cost.

[6] There are various sources of Jitter, congestion is only one of them.

Generally network equipment supports a wide range of operation speeds, for example DOCSIS based services commonly provide speeds greater than 100Mbps. If these same modems are used on slower connections, their buffer designed to be large enough for much higher speeds, become too big [51, 70].

Most routers drop packets only if their buffer is full, this approach is called tail-drop. On older routers the buffer size was fairly small so packets began to drop shortly after the link became saturated, allowing for only a few millisecond of buffering and giving TCP the ability to adjust [71]. On newer routers buffers became large enough to hold several megabytes of data translating to several seconds' worth of delay at a few Mbps line rate. Because most deployed versions of TCP use packet drop as the single congestion indicator - TCP does not slow down. Bufferbloat therefore effectively defeats TCP congestion control mechanisms and on congested links makes any interactive applications unfeasible [51, 70].

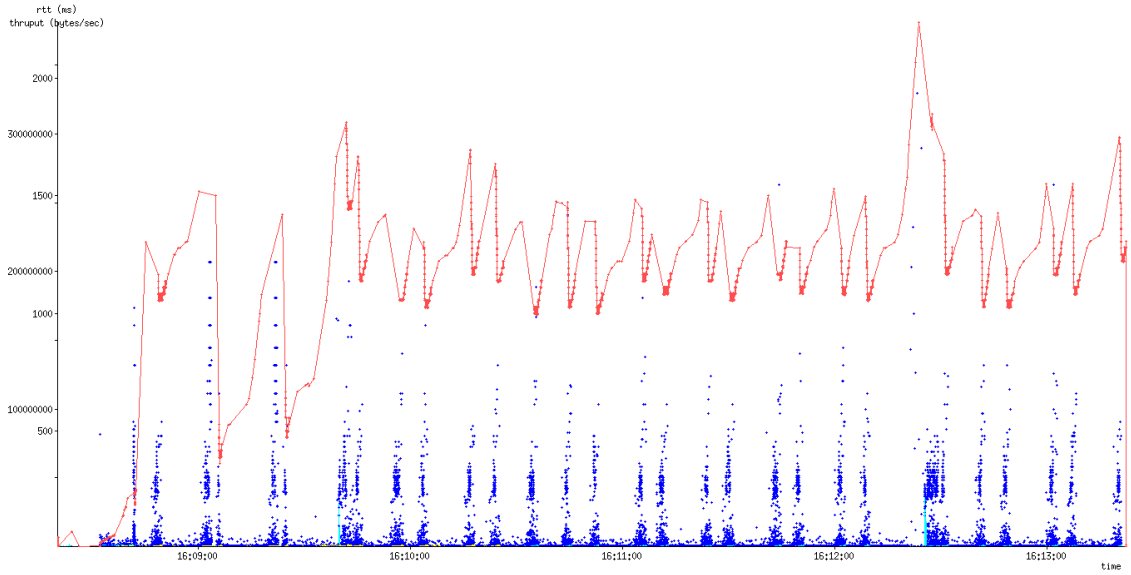

Figure 28: Effect of Bufferbloat on RTT (red) and throughput (blue), RTT increases from a few ms to several seconds and keeps there while under load, the throughput spikes occur when the buffer is freed, as soon as the buffer is again filled throughput drops [72].

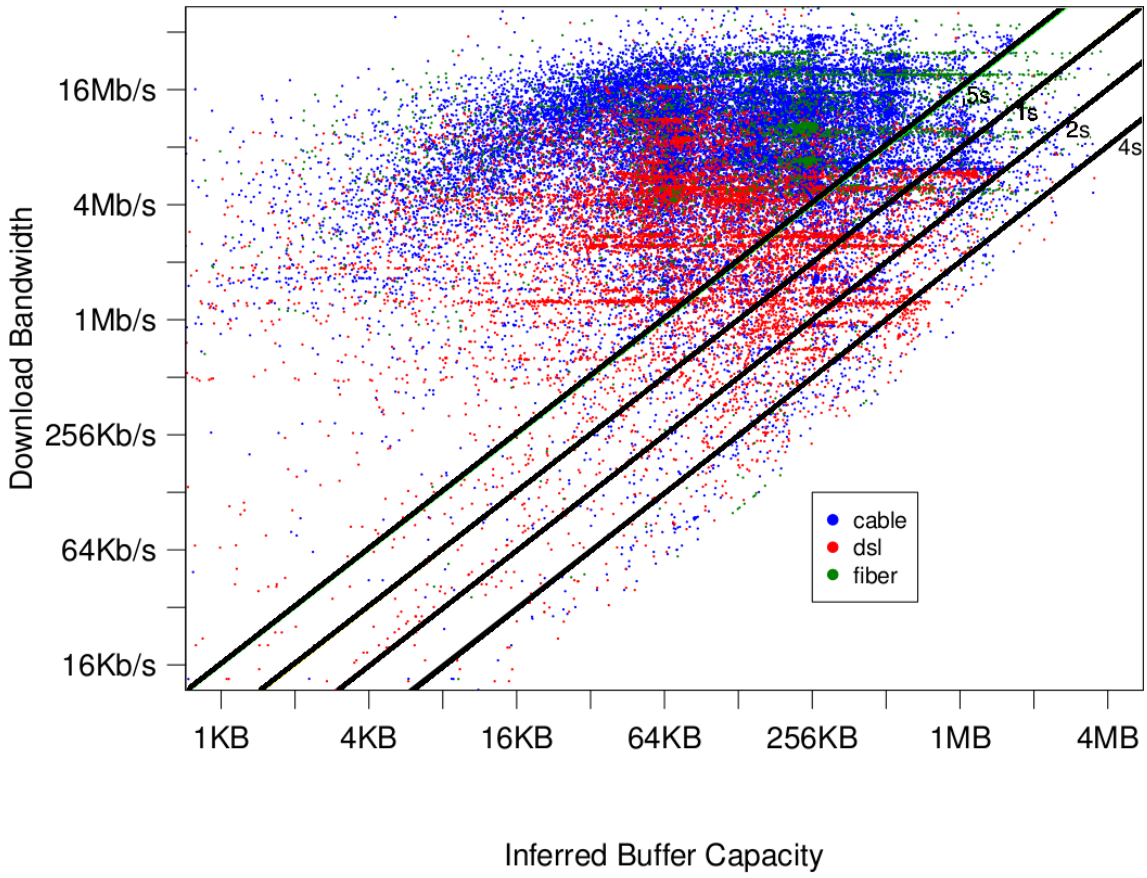

Figure 29: Inferred downlink buffer capacity of residential routers, black lines represent delay (0.5s, 1s, 2s, 4s) added by the buffer size at the given rate [72].

Bufferbloat is not limited to SOHO routers; its aggregate form was noticed even in WWAN networks with reported RTT spikes of up to 30 seconds that were not caused by the link layer/ARQ [73]. It is a design issue that can be found practically in all network drivers/OSs/HW as they mostly implement fixed length packet buffers [74, 75]. Meaning it will be visible everywhere where aggregation occurs and Active Queue Management is not deployed, creating unmanaged congestion which can lead to congestion collapse [51, 70, 76].

1.5.2 Delay Based Congestion Control

Most TCP stacks today rely upon packet loss in order to detect network congestion and to respond by drastically reducing the sending rate. Packet loss therefore significantly degrades the connection performance. An alternative strategy is delay-based congestion control (DBCC), which attempts to avoid packet losses by:

1. Detecting congestion early through increase in the RTT.

2. Reducing the connection sending rate in order to alleviate congestion before packet losses can occur.

Estimator	Metric Used	Compared to	Condition Used To Estimate Congestion
CARD [Jai89]	Delay Gradient	Previous RTT	$\frac{RTT-prevRTT}{RTT+prevRTT} * \frac{Window+prevWindow}{Window-prevWindow} > 0$
Tri-S [WC91]	Throughput Gradient	Initial RTT	$\frac{firstRTT}{Window-prevWindow} * (\frac{Window}{RTT} - \frac{prevWindow}{prevRTT}) < 0.5$
Dual [WC92]	Delay	Min and Max RTT	$\frac{minRTT+maxRTT}{2} < RTT$
Vegas [BOP94]	Throughput	Min RTT	$window * (1 - \frac{minRTT}{RTT}) > 3$
BFA [AR98]	Delay variability	fixed thresholds	$signed - rtt - variability > 0.01$
CIM [MNR03]	Delay	Previous 20 RTTs	$avg(RTT_2) > avg(RTT_{20}) + RTT_{std-dev}$
FAST [WJLH06]	Delay	Min RTT	$1 - \frac{100}{window} <= \frac{minRTT}{avgRTT}$
DECA [YQC04]	Delay	Min and Max RTT	$\frac{(maxRTT+minRTT)}{2} - perFlightMaxRTT < 0$
DAIMD [LSM+07]	Delay	Min RTT	$(pktInFlight > x)\&(sRTT - minRTT) > 0.02$

Table 5: Delay Based Congestion Estimator Descriptions: References in the first column provide more details on the estimators considered [29].

As we already have shown in 1.4.1 RTT variance due to radio effects cannot be avoided and also in most cases cannot be considered a congestion event. In general, delay based congestion control algorithms are not widely used primarily because when compared to loss-only based implementations perform sub-optimally [77, 78, 79]. But behavior such as Bufferbloat can be effectively detected only by a delay based congestion estimator as there is simply no packet drop for long periods of buffering. Clearly no loss-only or delay-only solution can solve the problem. Several proposals seek to combine them. But even such hybrid Delay/Loss-based congestion control – as implemented in TCP Illinois and Compound TCP were shown to exhibit poor scaling as path bandwidth-delay product increases [78].

Micro Transport Protocol (µTP) is as far as the authors are aware of the only wide scale deployed protocol (µTP is used as the primary transport protocol in the BitTorrent P2P network) that has some kind of delay-based congestion control. It focuses on not creating additional buffer load at SOHO routers by monitoring the RTT and throttling down the transmission rate whenever the threshold of 100ms is reached, effectively yielding to TCP traffic [80, 81].

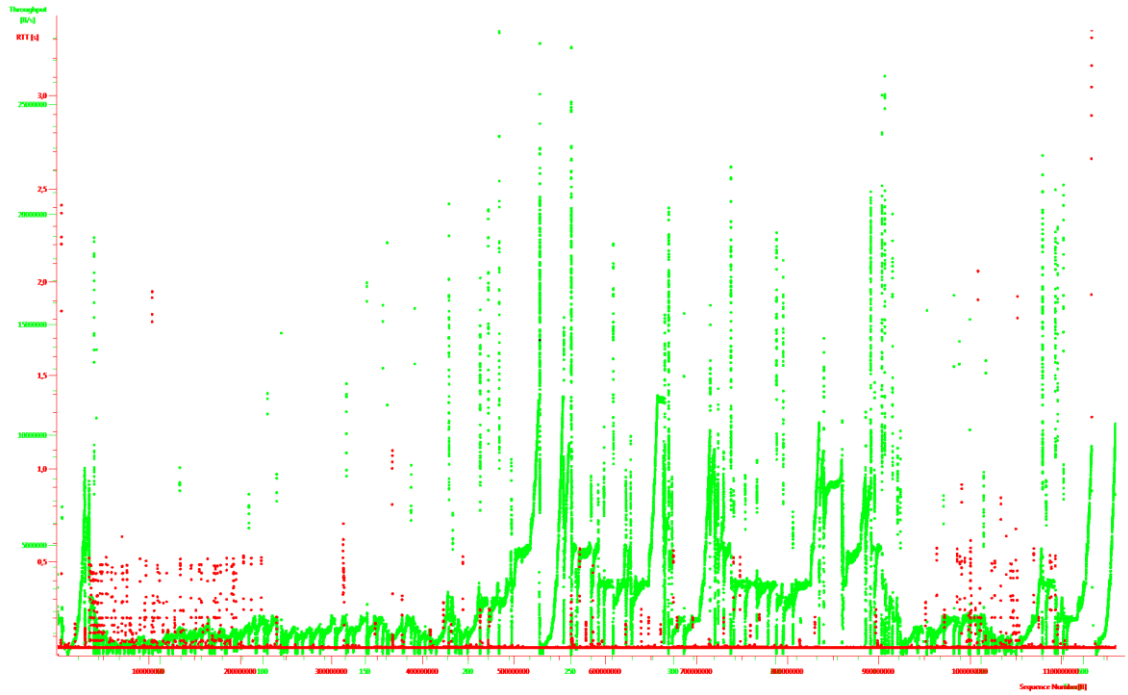

Figure 30: TCP iperf between two servers with 40ms RTT, 14 hops away, red is the plot of the TCP RTTs, green is the TCP throughput. Both nodes were running Linux without a delay-based congestion estimator, thus the rate was lowered only on loss events. From the plots it is clear that there is a connection between the RTT and congestion events as RTT anomalies always occurred before/thru congestion events/rate reduction.

1.5.3 Active Queue Management

Active Queue Management is a technique that employs preventive packet dropping or ECN marking before the network buffer is full. This approach often sounds counter-intuitive to many network engineers as perfectly good packets are being dropped even when there is still free buffer space and may be one of the reasons of the very sparse deployment of AQM [82]. The first AQM scheme was Random Early Detection (RED) [83]. It was developed because the classic tail-drop scheme penalized bursty traffic. And also led to global synchronization due to all TCP connections being forced by drop to “hold back” simultaneously, and then step forward simultaneously [84].

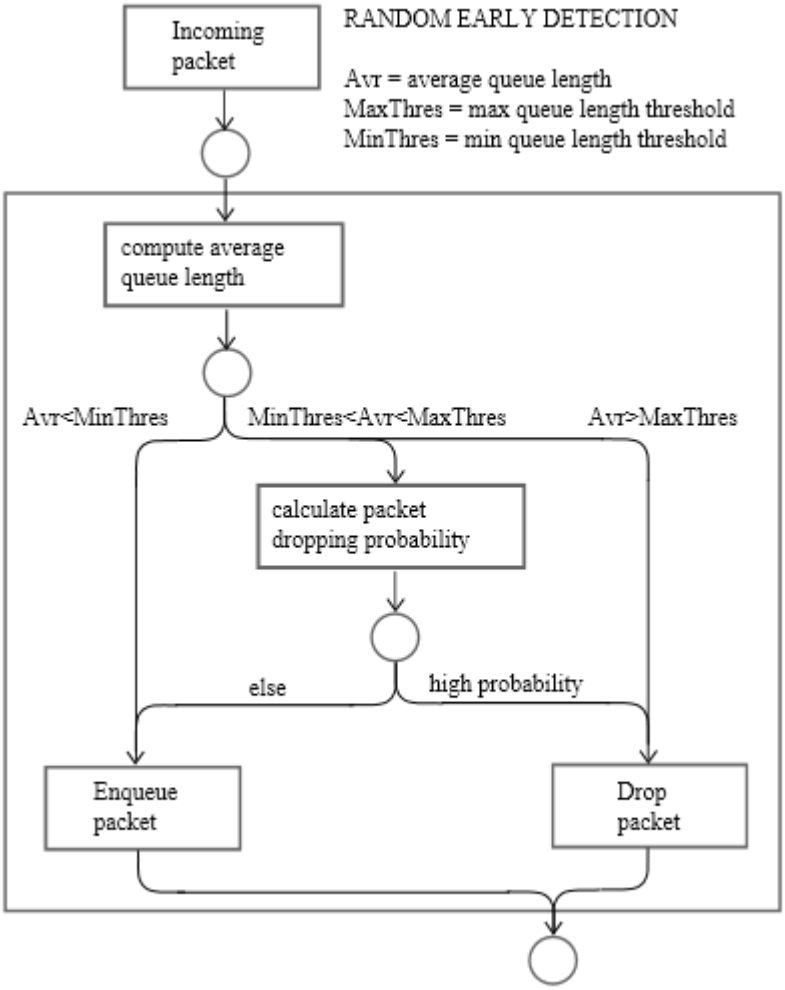

Figure 31: Random Early Detection [85].

Even if RED has only three configurable parameters, it did not become popular because these parameters have to be tuned to achieve good performance. Modern AQM algorithms as ARED, Blue or Pi are self-tuning, but still lack major rollout. Although, a variant of Blue - Stochastic Fair Blue has been included in the Linux kernel since version 2.6.39 [86]. Massive rollout of AQM was proposed as a solution for the Bufferbloat problem [76, 82].

An interesting approach to AQM is Remote AQM (RAQM) which introduces a method of imposing desired active queue management behavior on an upstream queue without even having access to the queue. RAQM achieves this by observing delay and based on that manipulates the upstream queue using congestion control mechanism as packet drops or ECN notifications [87].

1.5.4 Explicit Congestion Notification

Explicit Congestion Notification (ECN) is an extension to the Internet Protocol and Transmission Control Protocol defined in RFC 3168 [88]. ECN allows end-to-end notification of network congestion without dropping packets. ECN is optional and only functional when the underlying network supports it. Some old network HW even drops packets with ECN bits marked, or as we discovered during tests in live networks in Slovakia - ECN bits often get cleared in WWAN networks. On the other hand, we were not able to detect any problems with ECN marked packets on the tested wired networks, consisting of several different ISPs and the Slovak Academic Network (SANET). ECN marking is performed on the two least significant bits of the DiffServ field in IPv4 and IPv6 headers. Four different codepoints can be encoded:

- 00: Non ECN-Capable transport – non-ECT
- 10: ECN Capable Transport – ECT(0)
- 01: ECN Capable Transport – ECT(1)
- 11: Congestion Encountered – CE

When an ECN enabled packet, marked with ECT(0) or ECT(1), is detected in an AQM queue that is experiencing congestion and the router also supports and has ECN enabled, it marks the packet with CE instead of dropping it. The receiver then echoes this information to the transmitting node that reduces its transmission rate. It was shown that ECN in most cases benefits the overall network performance as it avoids costly RTO retransmissions. Modern AQM implementations usually apply ECN only until certain critical load is reached, after which they start dropping packets to avoid even higher congestion. Despite only a very limited number of network equipment is known to drop ECN marked packets, practically all operating systems have ECN disabled by default because of the fear that it might cause network problems [89, 90].

ECN capable transport protocols include TCP [88], DCCP [91], SCTP [92, 93] and recently also RTP [94, 95]. The most interesting protocol seems to be the Data Center TCP (DCTCP) which is part of MS Research - Cloud Faster research project. DCTCP uses the fact that most new datacenter switches support ECN. *DCTCP leverages Explicit Congestion Notification (ECN) in the network to provide multi-bit feedback to the end hosts. We evaluate DCTCP at 1 and 10Gbps speeds using commodity, shallow buffered switches. We find DCTCP delivers the same or better throughput than TCP, while using 90% less buffer space. Unlike TCP, DCTCP also*

provides high burst tolerance and low latency for short flows. In handling workloads derived from operational measurements, we found DCTCP enables the applications to handle 10X the current background traffic, without impacting foreground traffic. Further, a 10X increase in foreground traffic does not cause any timeouts, thus largely eliminating incast problems. [96]

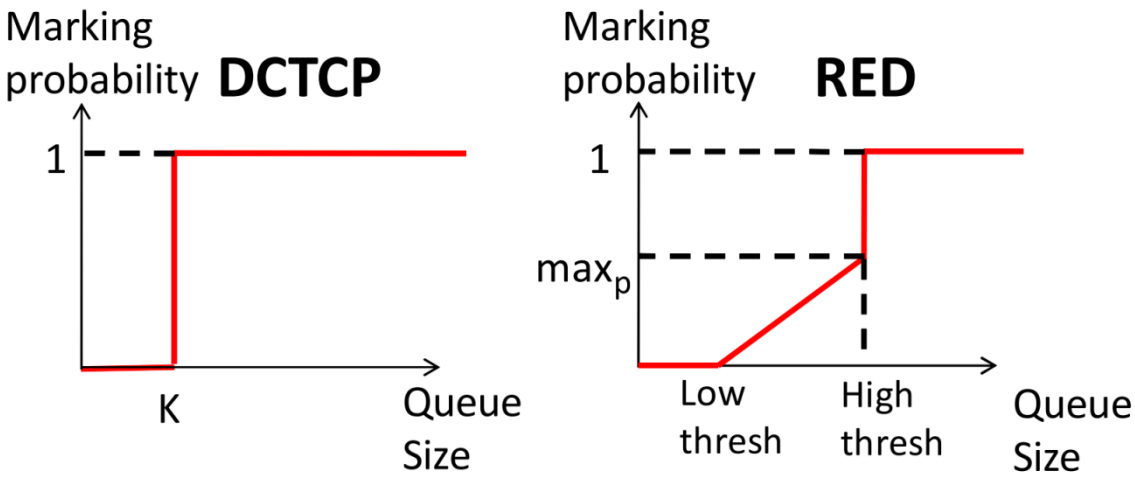

Figure 32: DCTCP vs. RED ECN marking probability, for DCTCP parameter K is number of packets defined as K > (C*RTT)/7, where C is in packets/second, RTT is in seconds [97].

1.6 Other Notable Transport Protocols

The Transmission Control Protocol is not universal, it is unable to provide transport for time constrained applications and its ability to utilize available bandwidth is often unsatisfactory. Several well-known protocols try to provide a solution in these cases. Datagram Congestion Control Protocol (DCCP) is an UDP based protocol, useful for real-time data transmissions, allowing for flow-based semantics like in TCP, but without reliable in-order delivery [98, 99]. DCCP is implemented in Linux kernel since version 2.6.14. Stream Control Transmission Protocol (SCTP) provides reliable, in-sequence transport of data chunks thus avoiding TCP head-of-line blocking. It strength lies in the support of multi-homing where both endpoints can consists of more than one IP and multi-streaming referring to the ability to transmit several independent streams of chunks in parallel. SCTP also solves the TCP SYN attack vulnerability by utilizing a 4-way handshake mechanism [100, 101]. Both DCCP and SCTP are TCP-Friendly[7]. Even if SCTP is able to reduce latency and improve throughput when compared to TCP [102], its benefits are not as much dramatic as other less-known protocols provide. In the following text we will look at the most interesting of them.

1.6.1 UDP-based Data Transfer Protocol

UDP-based Data Transfer Protocol (UDT) is a high performance data transfer protocol designed for transferring large datasets over high speed WANs [103, 104]. UDT has won the Supercomputing conference High Performance Bandwidth Challenge three times in 2006, 2008 and 2009 which was also the last Bandwidth Challenge held [105]. The challenge was designed to test the limits of network capabilities, and showcase multi-gigabit-per-second demonstrations never before thought possible [106]. UDT is TCP-Friendly and in most cases less aggressive than TCP. Instead of ACKs proportional to data packets it uses periodic ACKs to confirm packet delivery, while negative ACKs are used to report packet loss. UDT uses slightly modified AIMD style congestion control called AIMD with decreasing increases - DAIMD. The increase parameter is inversely proportional to the available bandwidth, thus it can probe high bandwidth rapidly and slow down for better stability when it approaches maximum link capacity. It also reduces the chance of loss synchronization by using a random number between 1/8 and 1/2 as the decrease factor.

[7] For example using TCP-Friendly Rate Control (TFRC), TFRC determines the throughput of a TCP connection under the detected conditions using the Padhye formula and limits the rate accordingly (1.4.3)

$$\alpha(x) = 10^{\lceil \log(L - C(x)) \rceil - \tau} \times \frac{1500}{S} \cdot \frac{1}{SYN}$$

Formula 6: Definition of DAIMD packet rate increase where SYN is the synchronization interval (0.01s), S is the UDT packet size, 1500 bytes is treated as the standard to compare to. C(x) is a function that converts the sending rate from packets/second to bits/second d (C(x) = x * S * 8), L is the measured link capacity. The protocol parameter τ is 9.

1.6.2 Structured Stream Protocol

Current Internet transport protocols offer applications a choice between two abstractions: reliable byte streams as in TCP or SCTP suited for long running conversations, or best-effort datagrams as in UDP or DCCP efficiently supporting small transactions and streams. Interestingly, neither abstraction adequately supports applications like HTTP that exhibit a mixture of transaction sizes, or applications like FTP, SIP or RTP that use multiple transport instances. Structured Stream Transport (SST) [107] tries to solve this issue by utilizing a hereditary stream structure, allowing applications to create lightweight child streams from any existing stream. Unlike TCP streams, these lightweight sub-streams do not incur 3-way handshaking delays on startup nor TIME-WAIT[8] periods on close that can cause TCP state overload. Each stream offers independent flow control, allowing different transactions to proceed in parallel with different priorities without head-of-line blocking, but all streams share one congestion control context. SST supports both reliable and best-effort delivery in a way that semantically unifies datagrams with streams and solves the classic "large datagram" problem, where a datagram's loss probability increases exponentially with fragment count. In addition to the new structured stream abstraction, SST utilizes several other novel design principles [108]:

- *SST builds its structured streams on top of an intermediate channel protocol: a connection oriented sequenced datagram service reminiscent of DCCP but semantically closer to IPsec's packet sequencing. The channel protocol's monotonic sequence numbers and replay protection logic, in particular, enable SST's lightweight streams to avoid 3-way handshakes or TIME-WAIT periods.*
- *The SST channel protocol selectively acknowledges packets via acknowledgment ranges, which provide more information than TCP's SACK and D-SACK extensions, facilitate forward acknowledgment and reordering*

[8] RFC 793 specifies the delay that a socket should not be reused to 4 minutes.

tolerance, and offer redundancy against lost acknowledgments, without the complexity or overhead of variable-length SACK headers.

- *SST separates the multiplexing and rendezvous functions that port numbers serve in traditional transports, using small, temporary local stream identifiers for multiplexing and more friendly service and protocol names for rendezvous during connection initiation.*
- *SST can attach a stream to multiple underlying datagram channels successively or at once, to insulate application streams from temporary failures, IP address changes, and channel lifetime limits.*
- *SST jumpstarts a child stream's flow control state by borrowing from its parent's receive window, allowing the initiator to start sending data on the child stream without waiting for the receiver's initial window update.*
- *SST demonstrates how layering and functional reuse enables substantially more functionality than TCP with no additional per-packet overhead in comparable scenarios.*

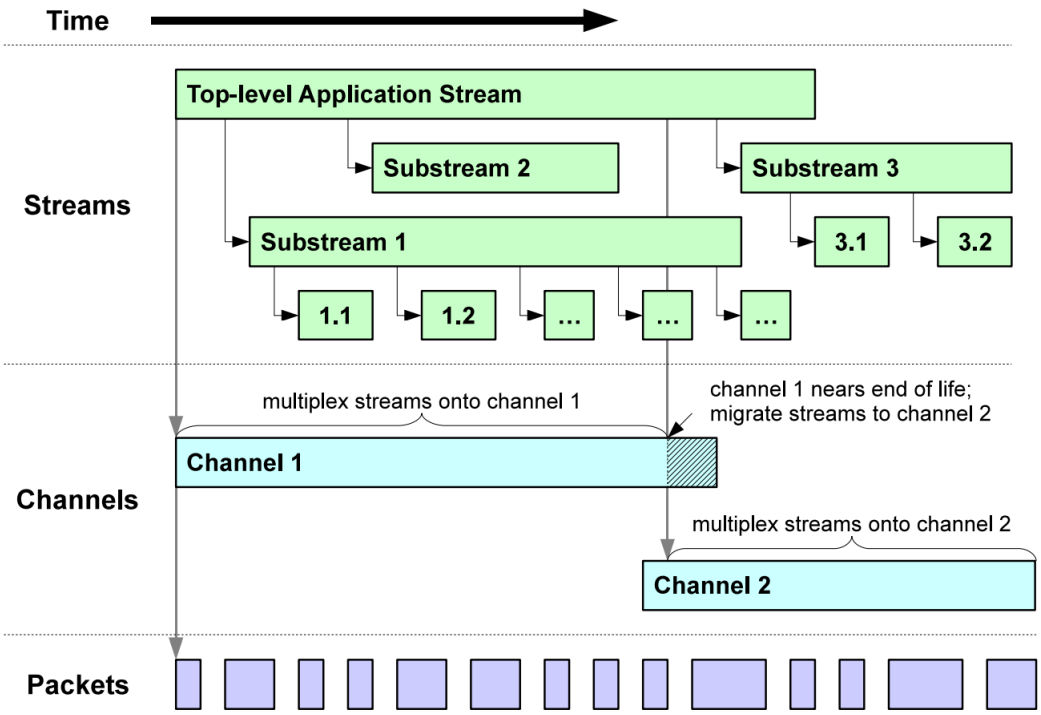

Figure 33: SST communication abstractions [108].

The performance benefit of SST can be demonstrated on an experiment where a simulated web browser workload, using one HTTP 1.0 transaction per stream over SST achieved the performance of HTTP 1.1 with pipelining [108].

SST also provides additional features as NAT traversal using UDP-hole punching when used over UDP and IPsec like authentication and encryption. SST implements congestion control at channel granularity, applications may use many concurrent streams without behaving "antisocially" as with redundant TCP connections. SST does not treat channel failure due to loss of connectivity as a "hard failure", like a TCP timeout. At the application's option, SST can retain stream state indefinitely until connectivity resumes and the negotiation protocol creates a new channel. At this point SST migrates the application's streams to the new channel and the application resumes where it left of [108].

1.6.3 Fast and Secure Protocol

Fast and Secure Protocol (FASP) is one of many commercial alternatives to TCP. It was chosen as a reference example for the whole category because it is currently being promoted by Amazon in association with its cloud computing services (Amazon EC2) as a *way to greatly accelerate the rate of bulk data transfer across the public Internet* and thus seems to have the widest customer base amongst all the commercial TCP alternatives [109]. Generally, the whole category lacks independent performance review from the research community, as the only available performance tests are provided by the developers themselves. FASP marketing material [110] states that:

- *Transfer speed is independent of network latency and is robust to packet loss, ensuring maximum speed over even the longest and most difficult WANs (1 sec RTT / 30% packet loss+)*
- *Does not require specialized hardware for high throughputs. Achieves 1 Gbps+ global WAN transfers on commodity computer hardware, and sufficiently lightweight for embedded systems including mobile and set top box platforms. Maximum throughput is obtained with a single transfer stream – multiple connections are not needed for high speeds.*

Because we were not able to find any independent performance review, we did our own measurements using the freely available client and the FASP test server. We were able

to confirm that FASP provides the promoted performance up to the demo limit of 45Mbps.

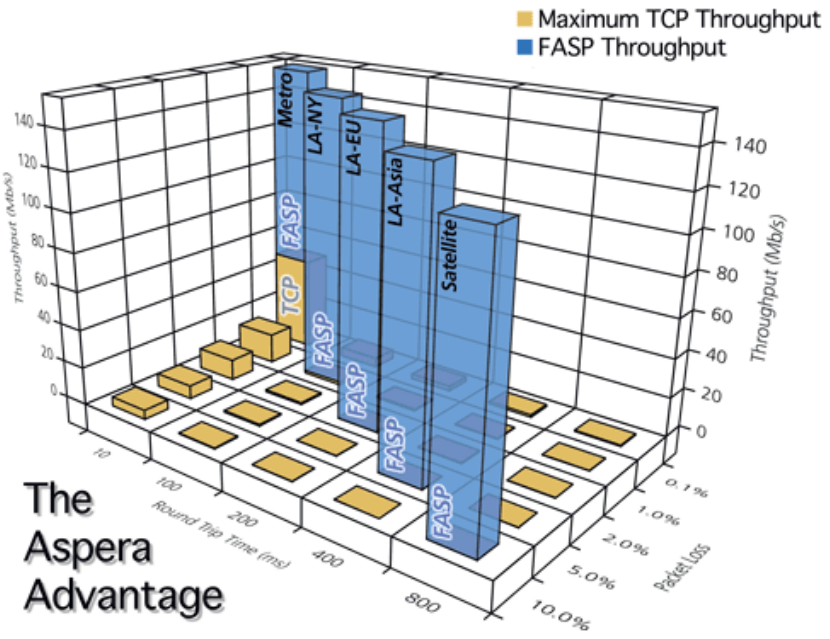

Figure 34: Aspera marketing material, FASP vs. TCP performance [111].

Internally, FASP uses a simple design: SSH for the control channel and UDP for transport. Reliability control is fully separated from the rate control; dropped data are retransmitted at the available bandwidth capacity. This design may be good at achieving maximal transmit rate but we have our doubts about its TCP-Friendliness and ability to share bandwidth with other applications as reported in [112].

1.6.4 Real-time Transport Protocol

Interestingly, even if the Real-time Transport Protocol (RTP) is primarily designed for delivery of audio and video, its design reassembles UDT or FASP. RTP is used in conjunction with the RTP Control Protocol (RTCP), while RTP transports the media streams, RTCP monitors the transmission statistics. To be able to provide timely delivery RTP is usually implemented on top of UDP. The recommended fraction of the session bandwidth added for RTCP is 5%. In usual unicast scenarios RTP transports streams with a pre-defined rate thus it will not expand to consume all available bandwidth, congestion is therefore not an issue [113]. RTP has already been specified in combinations with DCCP [114], SCTP [115], ECN [116] and even FEC [117].

2 Fountain Codes

In his 1948 seminal paper [118] Claude Shannon showed that communication channels are generally characterized by two factors: bandwidth and noise, where noise is anything that can disturb the transmission. For instance, in the case of the erasure channel, which is the standard model for transport networks, Shannon showed that the capacity of the channel with or without feedback is *(1 – f)*, where *f* is the channel erasure probability. This means that even if a fraction *f* of information is lost, information can still be transmitted reliably assuming that an appropriate Forward Error Correcting (FEC) code that operates at a rate strictly less than (ideally very close to) the channel capacity is used. This capacity is called the channel capacity and defines the limiting maximal rate at which reliable communication is still possible. Although Shannon's approach was very general and started the field of information theory, the proofs however used random coding arguments and did not explain how to construct such a code with reasonable decoding complexity (ideally linearly in length) [118, 119]. Only very recent advances in sparse graph codes or polar codes have proved that the channel capacity can in fact be achieved with linear complexity across a wide range of channels. This quest was initiated back in 1997 by Fountain codes. Foutain codes belong to the class of FEC codes called rateless codes, since they have the additional feature of automatically adjusting their rate to the channel characteristics forming [119].

ARQ based protocols, as for example TCP [120, 121], add delay on any retransmission event: this delay will always be more than 1 RTT which would be the case if the loss would be immediately detected. In real life it is more than that because of the method used to detect losses: RTO or duplicate ACKs. Notice that there is no RTT in the Shannon capacity equation, thus RTT should not have an impact on the achievable rate, and this is completely in contrast with any ARQ based scheme. We can always add more bandwidth, but RTT is a physical hard limit set by the speed of light, thus an efficient Internet needs to be a RTT independent Internet. As we already have shown, TCP congestion window scaling is directly dependent on RTT, spurious RTOs can trigger unnecessary retransmissions/congestion events which even if detected always will add delay, and of course any non-spurious loss degrades throughput. Forward Error Correction and its subgroup of erasure codes allow handling such events without the negative impact of RTT both on delay and on throughput. Traditionally,

FEC schemes as Reed Solomon (RS) were used in CDs, DVDs, DSL, WiMAX or RAID arrays have a fixed code rate which is the proportion of data-stream that is useful [122]. Thus the erasure probability has to be estimated forehand and the code rate chosen appropriately. But in a dynamic environment as Internet surely is, it is impossible to know the erasure probability before the transmission.

Fountain codes, because they are rateless erasure codes, solve this problem. Fountain codes are a class of erasure codes with the property of generating a potentially unlimited number of encoded symbols from a given set of source symbols so that the original source symbols can ideally be recovered from any subset of the encoded symbols of size equal or slightly larger than the number of source symbols. These codes thus do not exhibit a fixed code rate and as we will show can achieve linear complexity.

The Fountain codes represent an ideal for data transmission as they enable reliable communication without the need for receivers to send any feedback and for senders to resend any packets. Let's assume there are k packets of a file that we want to send, fountain codes can generate potentially limitless number of encoded packets on-the-fly, any $k(1+\alpha)$ packets, where α is the loss rate, regardless of the order are sufficient to reconstruct the original file. It does not matter what data is received or lost, the only factor is to receive enough data. Fountain codes therefore enable to design the flow and congestion control mechanisms independently of the reliability of the transmission links [119].

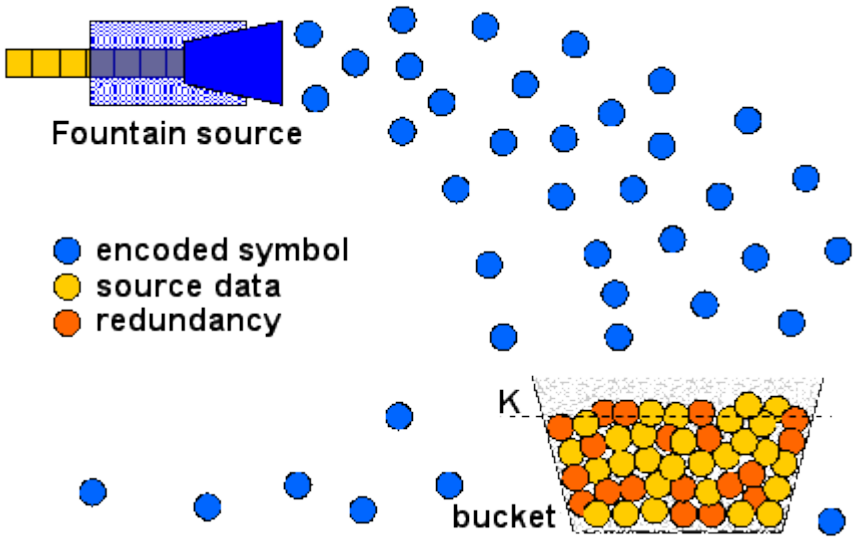

Figure 35: Digital Fountain Codes [123].

2.1 Random Linear Fountain Codes

Random linear codes represent the simplest form of fountain codes and therefore the core concepts will be described on them. Let's consider an encoder for a file of size K, with packets $s_1 s_2, \ldots, s_k$, where packets are the elementary transmission units. At each clock cycle, labeled n, the encoder generates K random bits $\{G_{kn}\}$ and the transmitted packet t_n is set to the bitwise sum, module 2, of the source packet for which G_{nk} is 1.

$$t_n = \sum_{k=1}^{K} s_k G_{kn}$$

Formula 7: Transmitted packet

This sum can be done by applying the XOR operation on the packets. Each set of K random bits is practically defining a new column in an ever growing binary generator matrix.

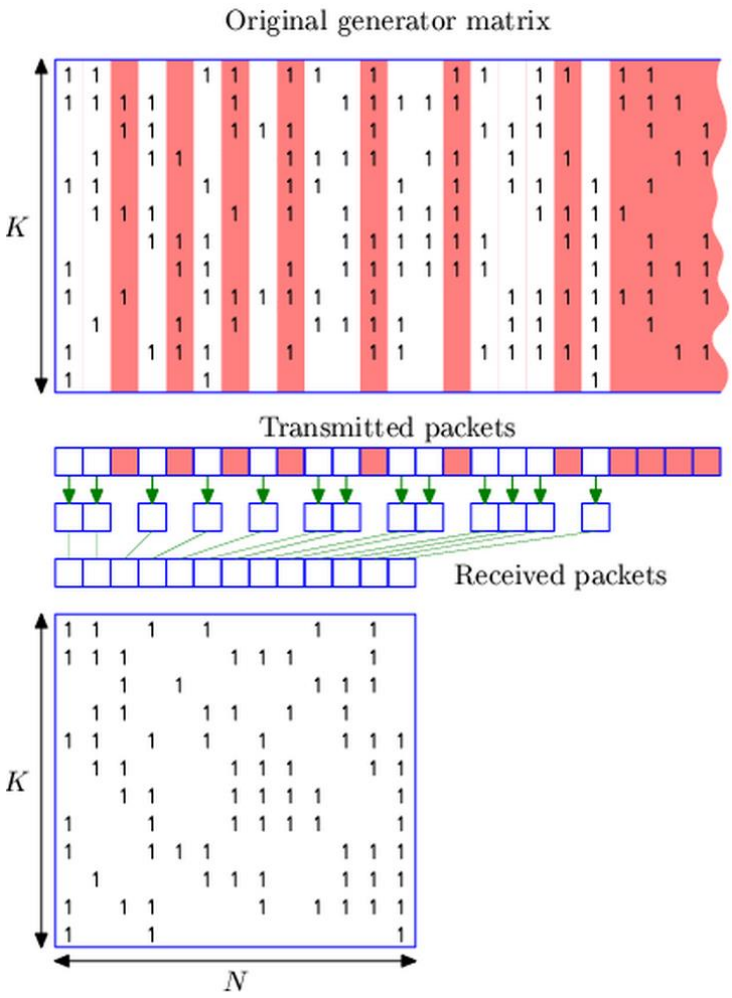

Figure 36: The generator matrix of a random linear code, red color represents the packets that were not received [124].

The receiver collects N packets. Let's assume that the receiver knows the generator matrix G_{kn} by some means. If $N < K$ then the receiver has not got enough information to recover the file. If $N = K$ then it is conceivable that the file can be recovered. If the *K-by-K* matrix G is invertible (modulo 2), the receiver can compute the inverse G^{-1} by Gaussian elimination, and recover:

$$s_k = \sum_{n=1}^{N} t_n G_{nk}^{-1}$$

Formula 8: Recovery sum

The probability of a random *K-by-K* binary matrix being invertible is a product of K probabilities, where each is the probability that a new column of the matrix is linearly independent of the preceding columns. Thus, the probability of invertability is $(1 - 2^{-K})(1 - 2^{-(K-1)})* \ldots *(1 - 1/8)(1 - 1/4)(1 - 1/2)$ which is 0.289, for any K larger than 10. If $N > K$, let $N = K + E$, where E is the number of excess packets. Let δ be the probability that the receiver will not be able to decode the file when E excess packets have arrived.

$$\delta(E) \leq 2^{-E}$$

Formula 9: For any K, the probability of failure is bounded and only dependent on the number of excess packets

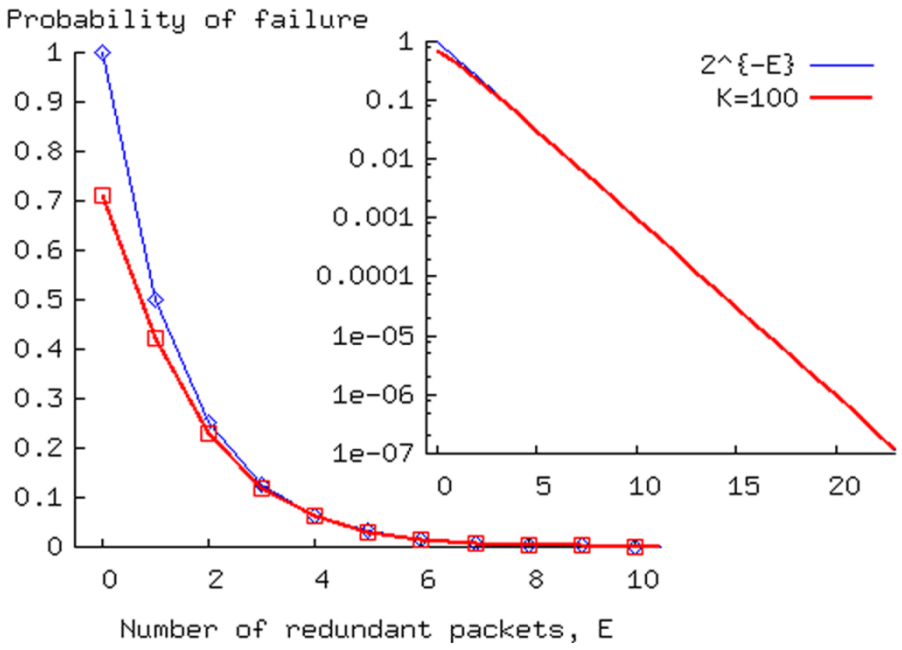

Figure 37: Performance of the random linear fountain, the red line shows the probability that the complete decoding is not possible as a function of the number of excess packets - E. The blue line shows the upper bound on the probability of error [124].

The number of packets required to have probability $1 - \delta$ of success is then $K + log_2 1/\delta$. Encoding complexity is $O(K^2)$ with decoding complexity of $O(K^3) + O(K^2)$ [119].

2.2 LT Codes

Even if the random linear fountain enables to approach arbitrarily close to the Shannon limit, its quadratic encoding and cubic decoding computational costs greatly limit its usage possibilities. The Luby Transform (LT) code retains the good performance of the random linear fountain but drastically reduces the encoding/decoding complexities to $K \log_e K$. The source data can be decoded from any set of K' encoded packets, for K' in practice, about 5% larger than K. The overhead $K' - K$ decreases (relative to K) as K increases: $K' - K \sim \sqrt{K}(ln(K/\delta))^2$ [124].

Every time an encoded packet is generated in a LT code, a weight distribution is sampled and an integer d between 1 and k is returned, where k is the number of source packets. Then d random distinct source packets are chosen, their value is added to yield the value of that encoded packet. The decoding failure probability depends only on the weight distribution. It was shown that if an LT code has a decoding algorithm with probability of error that is at most inversely polynomial in the number of source packets, and if the algorithm needs n encoded packets to operate, then the average weight of an encoded packet needs to be at least $ck*log(k)/n$ for some constant c. Thus, in the desirable case where n is close to k, the encoded packets of the LT code need to have an average weight of $\Omega(log(k))$. It is possible to construct a weight distribution that matches this lower bound via a faster decoder. Such distributions were exhibited by Luby and are the core innovation behind LT codes [125, 126].

2.3 Raptor Codes

Raptor (Rapid Tornado) codes are the first known class of fountain codes with linear time encoding and decoding and therefore represent a significant improvement over LT codes. Raptor codes use a combination of two codes: the outer code (pre-code) and the inner code. The pre-code, is a fixed rate erasure code and can itself be a combination of multiple codes. For example the 3GPP standardized code uses a combination of high density parity check code and a regular low density parity check code. The inner code is a form of LT code; it takes the output of the pre code and generates the encoded packets [119, 124, 126].

For the latest generation, the RaptorQ codes (RQ), which were standardized in RFC6330 [127] in August 2011, the probability of failure for K received packets e.g. without excess packets is less than 10^{-2} percent, with one excess packet less than 10^{-4} and with only 2 excess packets is less than 10^{-6}. Practically meaning that when a stream should be protected against 10% packet loss, with K=200 packets and with the failure probability of 10^{-6} the RQ code would add an overhead of 2 packets. The bandwidth overhead could then be calculated as: 202/(1-0.1) ~ 225, then 225/200 = 1,125 thus the overhead is 12.5%. We could compare this result to what could be achieved using a random linear fountain resp. using an older Raptor code (R10) which for packet loss rate of 10 percent performs similarly to an random linear fountain. Using the equation in Formula 9 resp. Figure 37 we determine that 24 excess packets would be required translating to 24.5% bandwidth overhead. RQ failure-overhead curve is clearly an improvement over that of the random linear fountain or R10 [128]. In fact, as we will show in the following text, this failure-overhead curve and specifically the performance with 0 excess packets is critical for the use of RQ in a general purpose unicast transport protocol.

The RQ is a systematic code, meaning that all the source packets are amongst the encoded packets, the encoded data can be considered to be a combination of the original source data and the repair data. RQ support from 1 to 56,403 source symbols (packets[9]) per source block (file/data stream). Every source block can be up to 3.5TB. RQ can generate up to 2^{24} encoded symbols per source block. Symbols can be from 1 to 65,536 bytes long, enough even to support super jumbo frames [129, 128]. Typically, the required buffer size is at minimum the size of the largest source block size. However, sub-blocking is also supported and therefore as low as 256KB of RAM can be enough for much larger source blocks e.g. 10MB [130].

[9] Per RFC6330 it is recommended that each packet contains exactly one symbol.

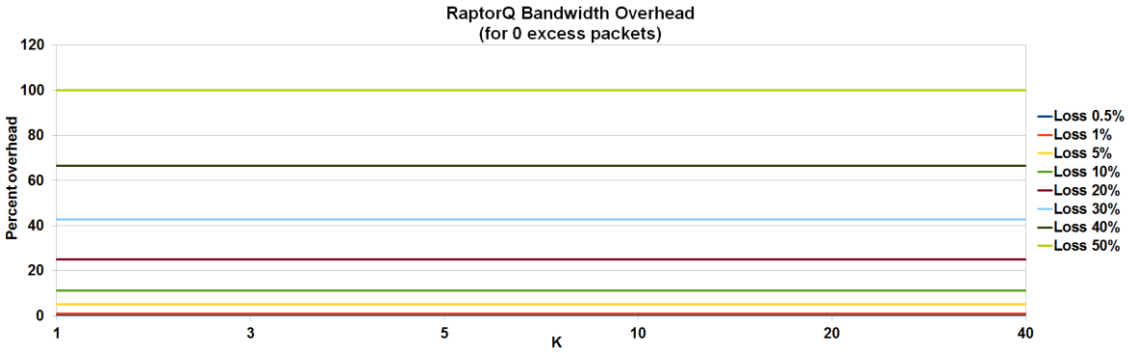

Figure 38: RaptorQ bandwidth overhead for 0 excess packets

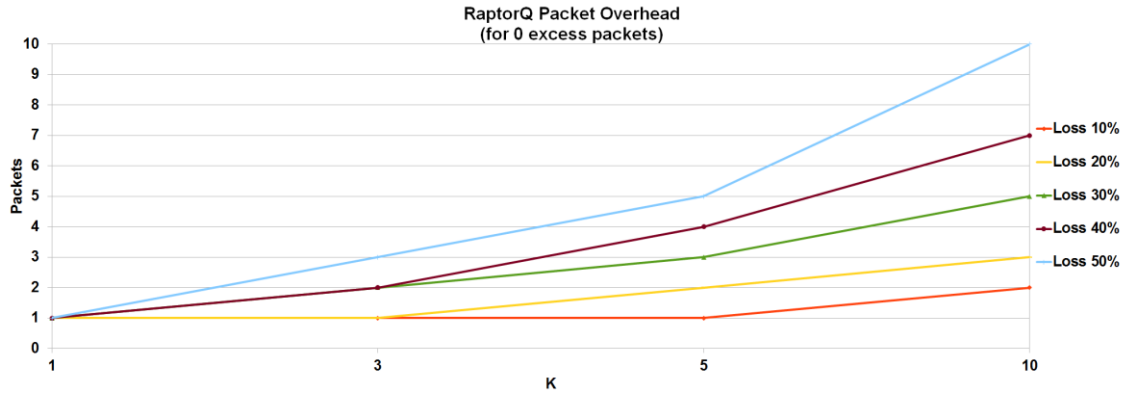

Figure 39: RaptorQ packet overhead for 0 excess packets

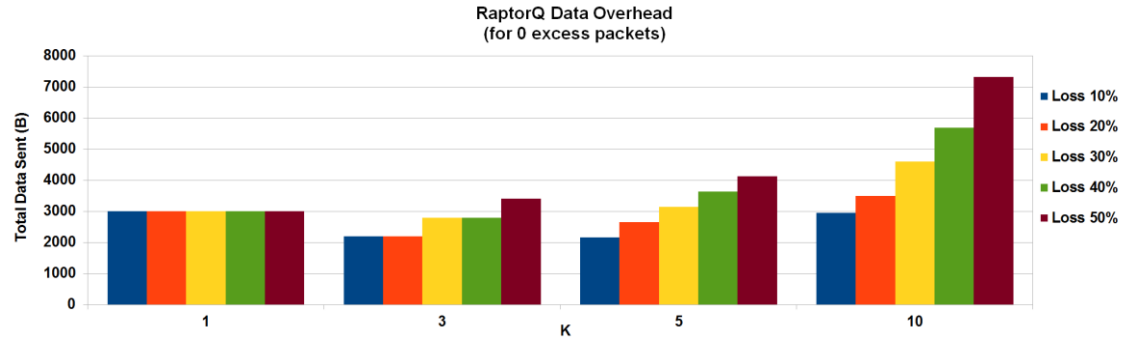

Figure 40: Data overhead example for small source block of 1460B and 40B as the header overhead.

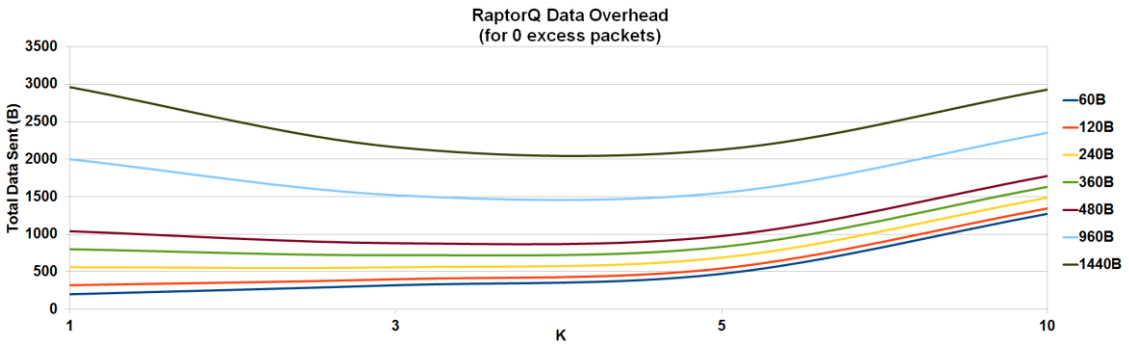

Figure 41: RaptorQ total transferred data for various source block sizes and K with 10 percent loss.

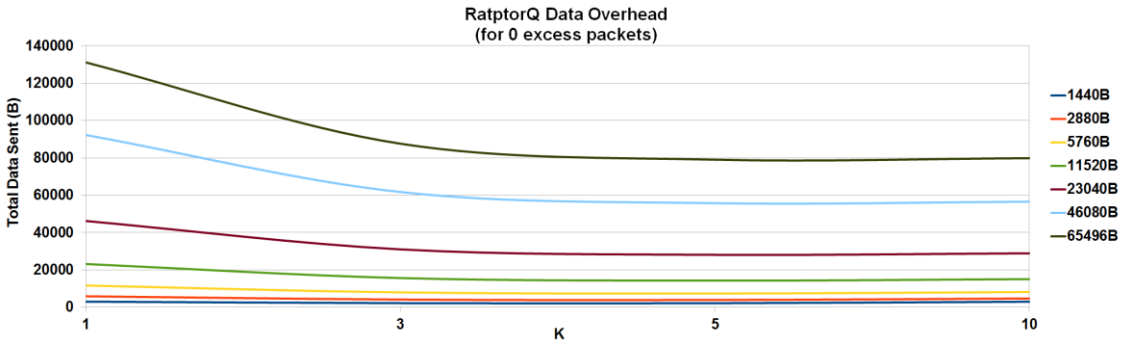

Figure 42: RaptorQ total transferred data for various source block sizes and K with 10 percent loss.

We explore the bandwidth overhead of RQ represented in percent and packets in Figures 38 and 39. Notice that there is no added overhead by the code itself, all overhead is due to the loss as we assume 0 excess packets. Figure 40 puts these numbers into network context in a strongly simplified model, suppose we want to send a 1460 bytes long request. To simplify we assume the headers are 40 bytes long as in TCP/IP. If we would send the request using a single packet and this packet would get lost we would need to resend it so the total data transmitted would be 3000 bytes, degrading RQ to a repetition code. For the one packet scenario it does not matter what is the packet loss rate, at any rate if the packet gets lost it needs to be resent whole. If we would divide this request into three separate smaller packets, the total data sent would be less with the same loss rate. For 10 and 20 percent loss the bandwidth saving would be 37 percent. Only at 50 percent packet loss it becomes less optimal than the single packet. Clearly the most optimal number of packets depends on the source block size; the situation can be seen in Figures 41 and 42. Similar overhead/source block size behavior has been studied for Online codes [131], which are a variant of non-systematic Raptor codes in [132].

It is critical to understand that the systematic fountain nature of the code would in a unicast scenario with backchannel create only as much waste redundancy as it would take for the receiver to acknowledge the successful decoding. The sender would be generating and sending encoded packets until an ACK would be received, this property assures that the best performance for both delay-sensitive and throughput-oriented applications can always be achieved. To simulate a delay-sensitive application, let’s suppose a Web request over a RQ coded stream. As mentioned earlier, Web requests are extremely sensitive to loss events; the loss of SYN-ACK packets has drastic consequences for TCP as its RTO is set to the default value of 3s. Some variants

as WATCP try to solve this problem by sending critical packets 4 times just to make sure they get through. A RQ stream would send only as much redundant packets as really required – determined by the acknowledgment of successful decoding from the receiver, thus dynamically adjusting the redundancy level. The amount of waste redundancy could be limited for example by adjusting the packet sending rate to the expected/measured loss rate, and if after RTT still no ACK was received add more redundancy. Also, waste redundancy does not add any delay to the request/respond; the receiver can respond immediately after enough data has been received disregarding all the extra data that are still in-flight at that moment. A throughput-oriented application could perform similarly to UDT/FASP, as the throughput achievable by using RQ is not necessarily dependent on the RTT or the packet loss with the right rate/congestion control. All issues with packet reordering or head-of-line blocking [15, 16] would not exist anymore as the order of received packets is not important, only the received amount is. It may be even possible to design the protocol so that does not need an initial handshake [10, 120, 121] and still provides reliable stream oriented delivery where stream can be anything ≥ 1 packet.

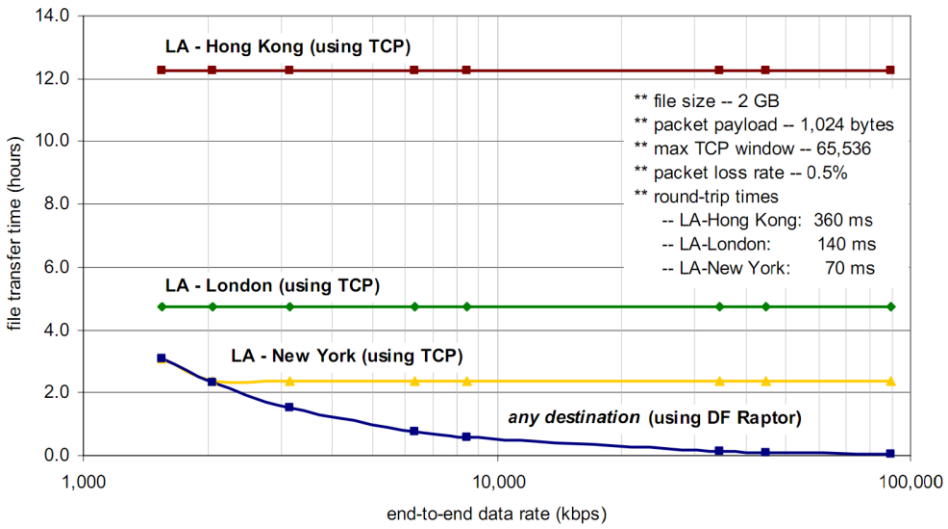

Figure 43: Raptor with large source block vs. TCP [135].

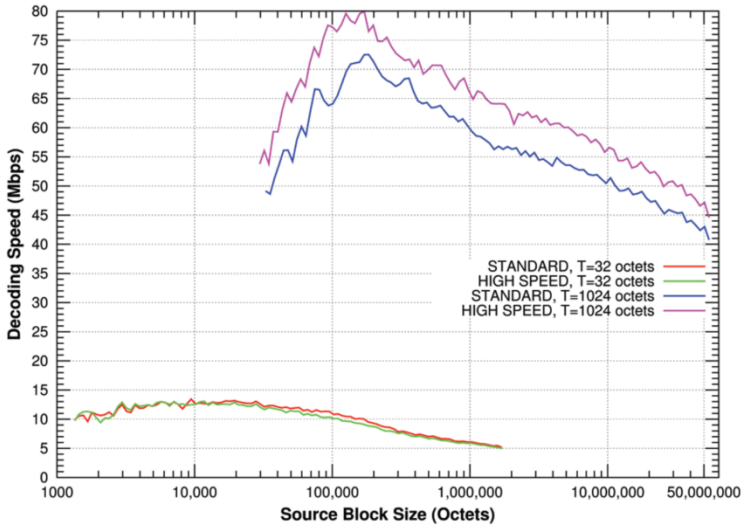

Figure 44: RaptorQ decoder performance on 1GHz first-generation Snapdragon 8250 (HTC Nexus One, Android 2.2, 680 DMIPS) with 2 excess packets, 50% random loss, T is the size of a packet in Bytes[10] [136]. For comparison, current third generation Snapdragon cores as the APQ8060 have 3750 DMIPS (used for example in Samsung Galaxy S2), upcoming fourth-generation APQ8064 achieve 8750 DMIPS.

2.4 Raptor Codes in Standards

The R10 code has already been adopted into a number of different standards including [128]:

- 3GPP Multimedia Broadcast Multicast Service
- IP Datacast (DVB-IPDC) for DVB-H and DVB-SH
- IPTV for DVB-IPTV and ITU-T H.701

All the existing standards are multicast/broadcast related as ARQ does not scale well, fountain codes provide a solution. Only the IETF FECFRAME working group states that: *The group will develop a protocol framework for application of FEC codes to arbitrary packet flows over unreliable transport protocols over both IP multicast and unicast.* But their primary focus is still on multimedia applications: *A primary objective of this framework is to support FEC for real-time media applications using RTP over UDP, such as on demand streaming and audio/video broadcast* [137]. One of the results of the working group is RFC 6363: Forward Error Correction (FEC) Framework. Raptor codes are one of the options that can be used with FECFRAME [138].

[10] Octet == Byte

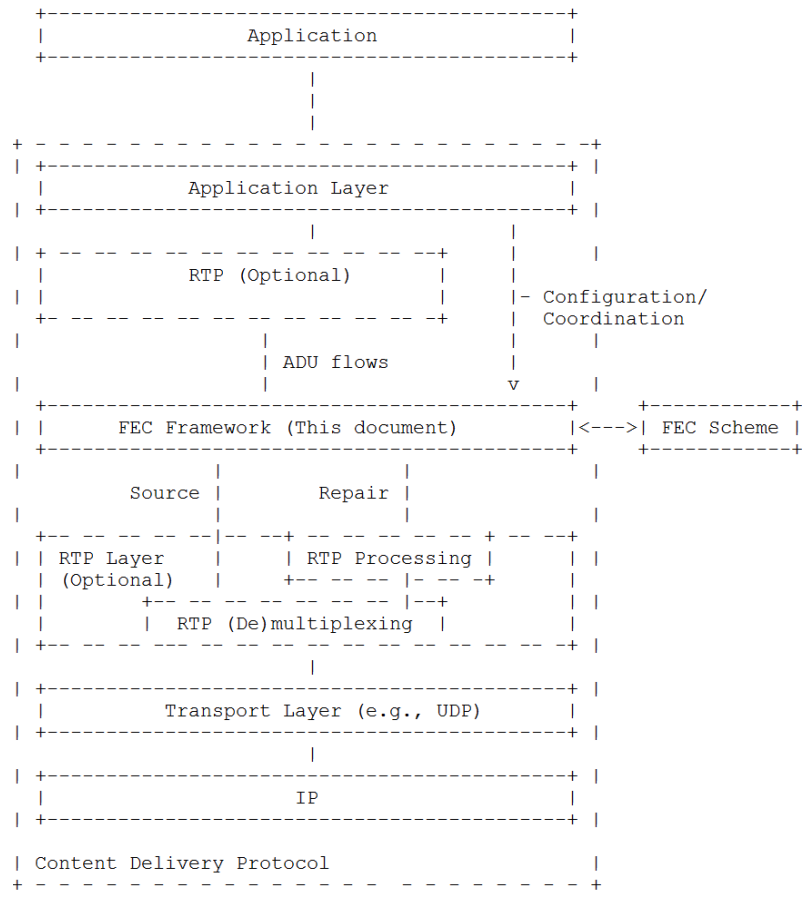

Figure 45: FEC Framework Architecture, from RFC 6363 [138].

The RFC 6363: *describes a framework for using Forward Error Correction (FEC) codes with applications in public and private IP networks to provide protection against packet loss.* It broadly covers:

- Sender operation – how to create a FEC encoded packet
- Receiver operation – how to receive and recover it
- Packet format
- FEC framework configuration information and scheme requirements
- Security considerations connected with FEC
- Operations and management considerations – middleboxes etc.

The RFC specifies normative requirements for congestion control as:

- The bandwidth of FEC repair data must not exceed the bandwidth of the original source data being protected
- FEC repair data must react to congestion similarly as source data

Except of these normative requirements the RFC does not specify any other details about what congestion control should be used. Also it does not go into any other transport protocol details. As illustrated in Figure 45.

For the use of FEC in multicast scenarios several rate/congestion schemes have been proposed such as: Wave and Equation Based Rate Control (WEBRC)[139], Fair Layered Increase/Decrease with Dynamic Layering (FLID-DL)[140] or TCP-Friendly Multicast Congestion Control (TFMCC)[141]. All these schemes try to model the performance of TCP and mimic its behavior to achieve TCP-Friendliness. As they have been specifically designed for the use in multicast scenarios the author is not aware of any clear benefit that they could provide for FEC protected unicast traffic.

3 Forward Error Correction in Unicast Transport Protocols

Many researchers proposed various solutions for the problems of TCP using FEC; some of them try to fix it by adding FEC somewhere in the inner-working of TCP and some present completely new protocols. FEC did not attract attention in the long evolution of TCP primarily due to its computational complexity [142]. Only in 1997 Luigi Rizzo suggested the possibilities for use of FEC in TCP using Reed-Solomon codes [143]. Following in Rizzo's footsteps, Anker et.al. [144] proposed integrating proactive FEC with TCP, but their method was not adaptive, did not consider variable MSS size or reactive FEC and its performance gain was limited to lower erasure rates (< 10%).

On the contrary, Loss Tolerant TCP (LT-TCP) [145] proposes the use of RS codes in a combination of proactive FEC as a function of the actual packet erasure rate and reactive FEC that aims to minimize the effect of erasures during the retransmissions. The scheme also includes adaptive MSS sizing. Its congestion control reacts only to ECN. Compared to TCP-SACK it was showed that LT-TCP achieves 5 times better goodput for 10% packet loss rate, while introducing about 30% FEC overhead. Another approach demonstrated in TCP-aware FEC [146] is to integrate FEC scaling with the congestion window scaling, again it uses RS codes.

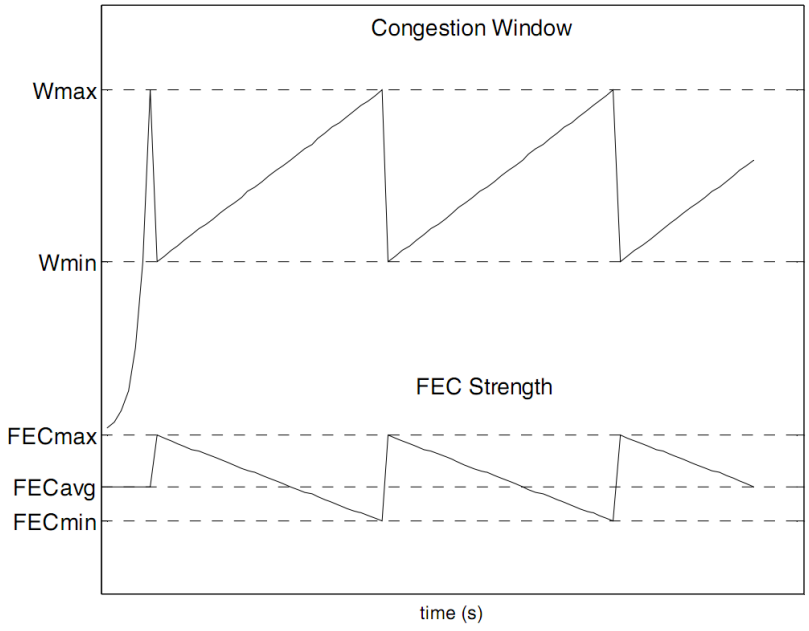

Figure 46: TCP-aware FEC strength evolution [146].

Driven by the TCP congestion and flow control mechanics, the proposed solution varies the strength of the RS code protecting TCP from incorrect decisions for window reduction. Simulation show clear benefits of this scheme when compared to fixed strength FEC schemes. Similar dynamic congestion-state based FEC scheme but applied to TFRC was shown to outperform fixed-rate FEC substantially in [147].

Until now all the schemes used RS codes, but in [148] the authors propose to use fountain codes to create FEC Transport Protocol (FECTCP) a TCP-like protocol where reliability is achieved using erasure correcting codes while the congestion control uses the distribution of losses to discriminate between errors on the channel and congestion in the network. The authors propose to use the Neyman Type A distribution for this purpose, as they state that most transmissions experience errors in burst or clusters for which this distribution was showed to fit.

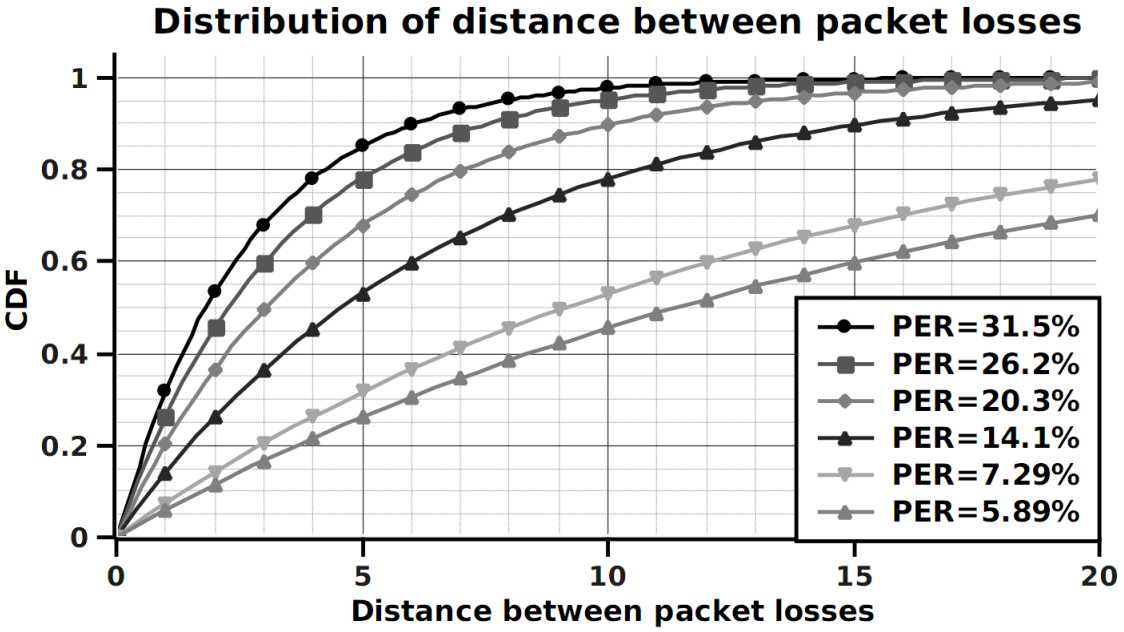

Figure 47: Distribution of distance between packet losses, CDF shows that the consecutive packet losses occur with a probability that is roughly equal to the average packet loss rate of the link [148].

The achieved results show better performance for high packet loss rates. Unfortunately, the authors did not specify what kind of fountain codes they used, as they state that the coding overhead makes their protocol inefficient at low loss rates. Their congestion window scaling is RTT dependent exactly as in classic TCP or TFRC, also FECTCP does not use ECN.

In [149] the authors propose to retain all existing TCP mechanisms for congestion control and retransmission but use random linear codes across all data in the window to mask loss at the network and improve responsiveness.

Another approach is proposed in the FEC-based protocol (FBP) [150], its control structure is based on RTP with separate data channel using RS protected UDP and control channel using reliable TCP. FBP FEC strength is dynamically adjusted as a function of the estimated loss which is calculated by indexing ACKs and comparing them to k. For example, if k=8 and first ACK is 10, the packet loss will be estimated as (10-8)/10 = 20%. Bandwidth is estimated by monitoring jitter, if the packets are arriving at close to the same interval as they are being transmitted, the network is probably capable of higher speeds. To avoid sending many excess packets before an ACK is received, the sender increases its sending interval by half the RTT until an ACK is received after $k(1+p_{loss})$ packets have been transmitted. Tests showed that FBP performs better than TCP if the loss rate is higher than 10%, and is able to perform even on loss rates above 25% at which TCP is not capable of functioning.

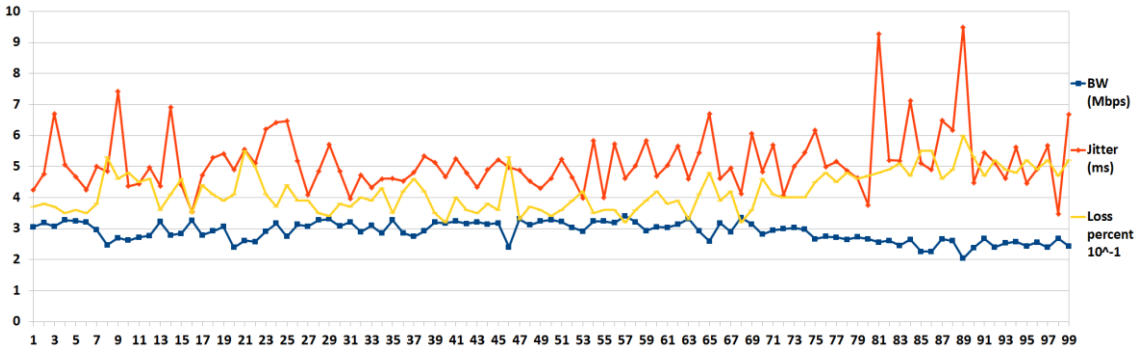

Figure 48: Jitter vs. bandwidth vs. loss rate. To check the proposed method of using jitter to estimate bandwidth we performed an UDP iperf over a WWAN connection, the results show some connection between jitter and bandwidth. In our tests we used a fixed rate UDP stream, thus the achieved throughput is inversely proportional to the loss rate. The possibility of using jitter to estimate available bandwidth has been also studied in [151, 152].

All the mentioned proposals have one fact in common, they use FEC to achieve better throughput. But FEC can be also used to minimize delay as shown in the research of Mehrotra et al – Reliable Protocol for Improving Delay (RAPID) [153, 154, 155, 156]. *Since delay is the most important factor in determining the perceived user performance of interactive applications, the overarching goal for the transport module is to minimize the expected delay incurred by each packet while ensuring reliable in-order delivery. The delay incurred by the packets has several components – e.g.,*

waiting time in the sender's queue, propagation delay, network queuing delay, retransmission delay, and decoding delay if a coding scheme is used. The requirement of in-order delivery can also cause additional delay as a packet may need to wait for prior missing packets to be delivered or decoded [156].

The authors argue that traditional media streaming can afford large buffers, for example to hold a few seconds of content. But because of the interactivity requirement, web-apps, games or remote-desktop applications cannot afford such significant buffering. Their proposed solution therefore tries to improve the performance by adding FEC packets but in the same time limit the overhead by using ARQ where it seems to be a better option. By using a hybrid random linear code FEC-ARQ scheme, RAPID is able to provide better delay performance than traditional ARQ protocols as TCP, FEC-ARQ schemes based on block codes (RS) and also opportunistic FEC.

Media streaming	Interactive App	File delivery
Strict deadline	Delay sensitive	No deadline
Best effort	Reliable delivery	Reliable delivery
No ordering	In-order	In-order
Low delay	Delay-aware	Delay agnostic

Table 6: Media streaming vs. interactive app vs. file delivery [156].

Opportunistic meaning that a FEC packet of all unacknowledged source packets is sent whenever the sender queue is empty, one such protocol was proposed in the early stages of RAPID development [153]. RAPID dynamically chooses between the following transmission policies:

- Sending a new source packet without coding.
- Sending a FEC packet of only the first certain number of unacknowledged packets. This is the FEC part of the protocol.
- Resending an already sent packet which has timed out or been negatively unacknowledged. This is the ARQ part.

The choice is made based on a cost calculation which takes into consideration the indexes of last acknowledged and first unacknowledged packets, propagation delay, queuing delay, time the original packet enters the sender queue and a few other factors.

In their latest publication the authors propose to use RAPID together with a delay based congestion estimator to avoid unnecessary adding delay by filling buffers.

In [157] the authors using Game Theory analyzed the behavior of TCP vs. their own idealized Fountain Based Protocol (FBP) in the presence of congestion. FBP was idealized in a way that it guaranteed that all received packets were useful. It was shown that any given host using TCP has an incentive to switch to FBP to obtain higher throughput, meaning that the Nash equilibrium will be reached when all hosts use FBP. Also, it was shown that when this equilibrium is reached, the overall performance of the network is similar to the performance when all hosts use TCP. The authors stated concern about the network performance when small units would be continually transferred because all their tests focused on simulated large file transfers. Small data units could be, according to them, problematic as it could be possible that useless packets not containing additional information could flood the network before the appropriate ACK from the receiver arrives at the sender.

As a closing remark to the overview of existing FEC-enabled unicast transport protocols we find it important to mention that even Raptor codes have been already used for special purpose unicast file delivery. Although, their usage has been limited to maximizing the throughput under difficult circumstances as high-RTT, high-loss networks etc. In these scenarios, Raptor has been directly implemented on top of UDP. Congestion control is not always required for example in private networks, but when it is, existing congestion control schemes as TFRC could be used [158, 159].

4 Conclusion

The Transmission Control Protocol transports majority of Internet's traffic, but it has many known problems, in fact so many that we believe that the only way to fix it, is to replace it. Trying to suddenly replace a protocol used by the whole Internet is of course neither realistic nor trivial. The aim of this work was to prove that it is necessary, incrementally possible, and beneficial to replace TCP in order to make Internet more efficient and scalable. For this goal to materialize, the incentive to switch has to be non-negligible. Not only for the corner cases, but consistently across all situations that a general purpose unicast transport protocol is supposed to handle.

Incentive is a measure of difference in value; to maximize it, it is necessary to prove that the current state is not optimal. The first chapter therefore summarized the current Internet architecture, explored its core problems and identified repetitive performance degrading design patterns. Starting from HTTP, moving down to TCP we presented how the Internet status quo is built upon design decisions that make the current Internet performance a function of RTT and packet loss, design decisions that waste RTTs. Internet is not unbreakable, Internet already collapsed once back in 1986. Mechanisms were put in place so that such situation would never happen again. Over time, network equipment manufacturers forgot about these congestion control mechanisms and Bufferbloat was born. We explored the problem, the proposed solutions and demonstrated that TCP is defenseless. Bufferbloat effectively defeats TCP congestion control mechanisms. Without them, the Internet stability, manageability and of course performance suffer. A congestion collapse is once again a real threat.

Lots has changed since 1970s when TCP's core logic was designed, amongst other things, high-speed Wireless WANs emerged. Forward Error Correction codes where one of the crucial technologies behind the mobile communication revolution. The value for the proposed Universal Transport Protocol comes from utilizing the revolutionary newest generation of FEC/Fountain codes - RaptorQ. We consider RaptorQ to be revolutionary because its properties as failure/overhead curves and linear computational complexity enable the construction of a truly universal Fountain based, channel capacity approaching, transport protocol. The second chapter described the basic principles behind Fountain codes and illustrated their unique properties in the networking context.

Special focus was given to unicast-specific scenarios, where until RaptorQ, the usage of Fountain codes was not always the optimal choice.

Prior research in the field of FEC enabled unicast protocols was surveyed in the third chapter. We explored both throughput and delay optimization schemes built upon FEC. Because prior to RaptorQ all FEC/Fountain coding schemes added too much overhead for small source blocks, the protocols were beneficial only under high-loss situations or combined FEC with ARQ to achieve better performance. This prior research showed that FEC enabled unicast transport protocol performance is superior to that of TCP, either for throughput or for delay. Using RaptorQ we believe it may be possible to create a scheme optimized for both throughput and delay, performing better than TCP under all conditions. Such protocol would enable to experiment with novel congestion/rate control mechanisms that could be robust enough to approach channel capacity even under difficult dynamic conditions such as on WWANs but still be able to detect and avoid congestion issues such as Bufferbloat.

5 Theses for dissertation

Based on the analysis of state of the art following theses are proposed:

1) Design and prototype a Fountain (RaptorQ) code based unicast transport protocol whose performance would approach channel capacity even under difficult conditions (packet loss, long RTT, jitter) and at the same time be optimized for smallest possible delay and minimal waste redundancy.

2) Design and prototype rate/congestion control mechanism optimized for the prototyped protocol with the ability to coexist with legacy TCP streams.

3) Analyze the proposed approaches using simulations.

4) Experimentally verify the simulation model.

Bibliography

1. **G. Huston, Telstra.** *TCP Performance.* The Internet Protocol Journal, Vol. 3, No. 2, pp. 2-25. 2000.

2. **T. Berners-Lee, R. Fieldingand, H. Frystyk.** *Hypertext Transfer Protocol -- HTTP/1.0.* RFC 1945. May 1996.

3. **Goland, Y.** *Multicast and Unicast UDP HTTP Messages.* IETF draft. 1999.

4. **Y. Goland, T. Cai, P Leach, Y. Gu, S. Albright.** *Simple Service Discovery Protocol/1.0 Operating without an Arbiter.* IETF draft. 1999.

5. **Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee.** *Hypertext Transfer Protocol -- HTTP/1.1.* RFC 2616. June 1999.

6. **B. Krishnamurthy, J. C. Mogul, D. M. Kristol.** *Key differences between HTTP/1.0 and HTTP/1.1.* Issues 11–16, 17. May 1999, Computer Networks, Volume 31, pp. 1737-1751. ISSN 1389-1286, 10.1016/S1389-1286(99)00008-0.

7. **E. Cohen, H. Kaplan, J. Oldham.** *Managing TCP connections under persistent HTTP.* Computer Networks, Volume 31, Issues 11–16, pp. 1709-1723, May 1999. ISSN 1389-1286, 10.1016/S1389-1286(99)00018-3.

8. **SPDY**: An experimental protocol for a faster web. Google, 2011. [Online: 30.12.2011] http://www.chromium.org/spdy/spdy-whitepaper.

9. **V. N. Padmanabhan, J. C. Mogul.** *Improving HTTP latency* Computer Networks and ISDN Systems, Volume 28, pp. 25-35, December 1995,. ISSN 0169-7552, 10.1016/0169-7552(95)00106-1.

10. **S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, B. Raghavan.** *TCP Fast Open.* ACM CoNEXT 2011, Tokyo. Japan. December 6–9 2011. ACM 978-1-4503-1041-3/11/0012.

11. **N. Dukkipati, M. Mathis, Y. Cheng, M. Ghobadi.** *Proportional Rate Reduction for TCP.* IMC'11. Berlin, Germany. November 2–4, 2011. ACM 978-1-4503-1013-0/11/11.

12. **N. Willis.** Reducing HTTP latency with SPDY. [Online: 30.12.2011] http://lwn.net/Articles/362473/.

13. **M. Nottingham.** The State of Proxy Caching. [Online: 30.12.2011] http://www.mnot.net/blog/2007/06/20/proxy_caching.

14. **Squid development team.** Squid configuration directive pipeline_prefetch. [Online: 30.12.2011] http://www.squid-cache.org/Doc/config/pipeline_prefetch/.

15. **M. Scharf, S. Kiesel.** *Head-of-line Blocking in TCP and SCTP: Analysis and Measurements.* IEEE GLOBECOM 2006. San Francisco, CA, US. Nov. 2006. 1930-529X, 1-4244-0357-X.

16. **R. Rajamani, S. Kumar, N. Gupta.** *SCTP versus TCP: Comparing the Performance of Transport Protocols for Web Traffic.* July 22, 2002.

17. **A. Hopkins.** Optimizing Page Load Time. [Online: 30.12.2011] http://www.die.net/musings/page_load_time/.

18. **S. Ramachandran..** Web metrics: Size and number of resources. 2010. [Online: 30.12.2011] http://code.google.com/intl/sk-SK/speed/articles/web-metrics.html.

19. Average Web Page Size Septuples Since 2003. [Online: 30.12.2011] http://www.websiteoptimization.com/speed/tweak/average-web-page/.

20. **I. Klimek.** *Instant Page Load (IPL).* TERENA Networking Conference - TNC 2011, 16. - 19. May, 2011. Prague, Czech Republic.

21. **I. Fette, A. Melnikov.** *The WebSocket Protocol.* RFC 6455. December 2011.

22. **J. Postel.** *Transmission Control Protocol.* STD 7, RFC 793. September 1981.

23. **M. Mathis, J. Mahdavi, S. Floyd and A. Romanow.** *TCP Selective Acknowledgment Options.* RFC 2018. October 1996.

24. **V. Jacobson, R. Braden and D. Borman.** *TCP Extensions for High Performance.* RFC 1323. May 1992.

25. **S. Floyd, T. Henderson and A. Gurtov.** *The NewReno Modification to TCP's Fast Recovery Algorithm.* RFC 3782. April 2004.

26. **V. Jacobson.** *Congestion Avoidance and Control.* Computer Communication Review, Vol. 18, pp. 314-329, August 1988.

27. **V. Paxson, M. Allman, J. Chu and M. Sargent.** *Computing TCP's Retransmission Timer.* RFC 6298. June 2011.

28. **M. Allman, V. Paxson and E. Blanton.** *TCP Congestion Control.* RFC 5681. September 2009.

29. **S. Rewaskar.** *Real world evaluation of techniques for mitigating the impact of packet losses on TCP performance.* THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HIL,. PhD thesis. 2008.

30. **N. Moller, K. H. Johansson, H. Hjalmarsson.** *Making retransmission delays in wireless links friendlier to TCP.* 43rd IEEE Conference on Decision and Control. Vol.5 , pp. 5134 - 5139. 14-17 Dec. 2004. 0191-2216, 10.1109/CDC.2004.1429622 .

31. **W. Gang, B. Yong, L. Jie, A. Ogielski.** *Interactions between TCP and RLP in wireless Internet.* Global Telecommunications Conference, GLOBECOM '99. Rio de Janeireo , Brazil . 1999. 0-7803-5796-5, 10.1109/GLOCOM.1999.830139.

32. **A. Muhammad, A. A. Iqbal.** *TCP Congestion Window Optimization for CDMA2000 Packet Data Networks.* International Conference on Information Technology (ITNG'07). pp.31-35. Las Vegas, NV, US. 2007.

33. **J. Schiller.** Mobile Communications. *Chapter 9. Mobile Transport Layer.* [Online: 30.12.2011] http://www.iith.ac.in/~tbr/teaching/docs/transport_protocols.pdf.

34. **A. Das, F. Khan, A. Sampath, S. Hsuan-Jung.** *Performance of hybrid ARQ for high speed downlink packet access in UMTS.* Vehicular Technology Conference, 2001. VTC 2001 Fall. IEEE VTS 54th. Vol.4, pp. 2133 - 2137. Atlantic City, NJ, US. 2001. 0-7803-7005-8, 10.1109/VTC.2001.957121.

35. **M. Meyer, H. Wiemann, M. Sagfors, J. Torsner, C. Jung-Fu.** *ARQ Concept for the UMTS Long-Term Evolution.* Vehicular Technology Conference, 2006. VTC-2006 Fall. 2006 IEEE 64th. Montreal, Que. 2006. 1-4244-0063-5, 10.1109/VTCF.2006.442 .

36. **WiMAX Forum.** Mobile WiMAX – Part I: A Technical Overview and Performance Evaulation. 2006. [Online: 30.12.2011] http://www.wimaxforum.org/technology/downloads/Mobile_WiMAX_Part1_Overview_and_Performance.pdf.

37. **P. Sarolahti, M. Kojo, K. Raatikainen.** *F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts.* ACM SIGCOMM Computer Communication Review, Volume 33. Issue 2, 2003. 10.1145/956981.956987.

38. **A. F. Canton, T. Chahed.** *End-to-end reliability in UMTS: TCP over ARQ.* Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE.. Vol.6, pp. 3473 - 3477. 2001. 0-7803-7206-9, 10.1109/GLOCOM.2001.966327.

39. **R. Ludwig, M. Meyer.** *The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions.* ACM Computer Communications Review, Vol. 30. No. 1, 2000.

40. **R. Ludwig, M. Meyer.** *The Eifel Detection Algorithm for TCP*. RFC 3522. April 2003.

41. **S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky.** *An Extension to the Selective Acknowledgement (SACK) Option for TCP.* RFC 2883. July 2000.

42. **Z. Ming, B. Karp, S. Floyd, L. Peterson.** *RR-TCP: a reordering-robust TCP with DSACK*. 11th IEEE International Conference on Network Protocols, pp. 95 - 106. 2003. 1092-1648, 10.1109/ICNP.2003.1249760.

43. **P. Sarolahti, M. Kojo, K. Yamamoto, M. Hata.** *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP*. RFC 5682. September 2009.

44. **P. Sarolahti.** *TCP Performance in Heterogeneous Wireless Networks.* University of Helsinki, Department of Computer Science, PhD thesis. 2007.

45. **Wikipedia.** TCP window scale option. [Online: 30.12.2011] http://en.wikipedia.org/wiki/TCP_window_scale_option.

46. **D.M. Chiu, R. Jain.** *Analysis of the increase and decrease algorithms for congestion avoidance in computer networks* Computer Networks and ISDN Systems, Volume 17, pp. 1-14. Issue 1, June 1989. 0169-7552, 10.1016/0169-7552(89)90019-6.

47. **R. Braden.** *Requirements for Internet Hosts - Communication Layers.* STD 3, RFC 1122. October 1989.

48. **L. Peterson, B. Davie.** *Computer Networks: A Systems Approach.* Fifth Edition. Morgan Kaufmann, March 25, 2011. 0123850592, 978-0123850591.

49. **A. Greenberg, C. Huang, D. Maltz, J. Padhye, P. Patel, M. Sridhara.** Cloud Faster: milliseconds matter. Microsoft Research, 03. 03 2010. [Online: 30.12.2011] http://research.microsoft.com/en-us/projects/cloudfaster/2010-03-03-cloudfaster-techfest10.pdf.

50. **Microsoft Research.** Cloud Faster. [Online: 30.12.2011] http://research.microsoft.com/en-us/projects/cloudfaster/.

51. **J. Gettys.** Bufferbloat, Dark Buffers in the Internet. Alcatel-Lucent, Bell Labs, 21. February 2011. [Online: 30.12.2011] http://staff.science.uva.nl/~delaat/netbuf/110126140926_Bufferbloat12.pdf.

52. **X. D. Wei, J. Cheng, H. S. Low, S. Hegde.** *FAST TCP: motivation, architecture, algorithms, performance.* IEEE/ACM Trans. on Networking, Vol. 14, no. 6, pp. 1246–1259. 2006. 10.1109/TNET.2006.886335.

53. **S. Floyd.** *HighSpeed TCP for Large Congestion Windows.* RFC 3649. December 2003.

54. **T. Kelly.** *Scalable TCP: improving performance in highspeed wide area networks.* ACM SIGCOMM Computer Communication Review, Volume 33. Issue 2, April 2003. 10.1145/956981.956989.

55. **D.J. Leith, R.N. Shorten, Y. Lee.** *H-TCP: A framework for congestion control in high-speed and long-distance networks.* PFLDnet Workshop. 2005.

56. **T. Hatano, M. Fukuhara, H. Shigeno, and K. Okada.** *TCP-friendly SQRT TCP for High Speed Networks.* Proceedings of APSITT 2003. pp. 455-460. November 2003.

57. **M. Gerla, M.Y. Sanadidi, W. Ren, A. Zanella, C. Casetti, S. Mascolo.** *TCP Westwood: congestion window control using bandwidth estimation.* Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE. Vol.3, pp. 1698 - 1702. San Antonio, TX , US . 2001. 0-7803-7206-9, 10.1109/GLOCOM.2001.965869.

58. **X. Lisong, K. Harfoush, R. Injong.** *Binary increase congestion control (BIC) for fast long-distance networks.* INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies. Vol. 4, pp. 2514 - 2524. 2004. 0743-166X, 10.1109/INFCOM.2004.1354672.

59. **H. Sangtae, R. Injong, X. Lisong.** *CUBIC: a new TCP-friendly high-speed TCP variant.* ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel, Volume 42. Issue 5, July 2008. 10.1145/1400097.1400105.

60. **D. Miras, M. Bateman, S. Bhatti.** *Fairness of High-Speed TCP Stacks.* Advanced Information Networking and Applications, AINA 2008. Okinawa, JP. 2008. 1550-445X, 10.1109/AINA.2008.143 .

61. **E. Blanton, M. Allman, K. Fall, L. Wang.** *A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP.* RFC 3517. April 2003.

62. **M. Mathis, J. Semke, J. Mahdavi, T. Ott.** *The macroscopic behavior of the TCP congestion avoidance algorithm.* ACM SIGCOMM Computer Communication Review, Volume 27, Issue 3, July 1997. 10.1145/263932.264023.

63. **J. Padhye, V. Firoiu, D. Towsley, J. Kurose.** *Modeling TCP throughput: a simple model and its empirical validation.* Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication. 1998. 10.1145/285237.285291.

64. **Wikipedia.** TCP tuning. [Online: 30.12.2011] http://en.wikipedia.org/wiki/TCP_tuning.

65. **SWITCH - Swiss University Network.** TCP throughput calculator. [Online: 30.12.2011] http://www.switch.ch/network/tools/tcp_throughput/index.html.

66. **L. Cottrell,** Throughput versus loss. Standford. [Online: 30.12.2011] http://www.slac.stanford.edu/comp/net/wan-mon/thru-vs-loss.html.

67. **W. Bao, V.W.S. Wong, V.C.M. Leung.** *A Model for Steady State Throughput of TCP CUBIC.* Global Telecommunications Conference (GLOBECOM 2010), 2010. 1930-529X, 10.1109/GLOCOM.2010.5684172 .

68. **S. Poojary, V. Sharma.** *Analytical Model for Congestion Control and Throughput with TCP CUBIC connections*. IEEE GLOBECOM 2011. 978-1-4244-9268-8/11.

69. **S. Floyd.** *HighSpeed TCP for Large Congestion Windows.* RFC 3649. December 2003.

70. **J. Gettys, K. Nichols.** *Bufferbloat: Dark Buffers in the Internet.* ACM QUEUE, Zv. Vol. 9, No. 11, November 2011. 1542-7730/11/110.

71. **D. Comer.** *Internetworking with TCP/IP: Principles, protocols, and architecture.* 3rd Edition. Prentice Hall, 688 pages. 2006. 0131876716, 9780131876712.

72. **J. Gettys.** Whose house is of glasse, must not throw stones at another. [Online: 30.12.2011] http://gettys.wordpress.com/2010/12/06/whose-house-is-of-glasse-must-not-throw-stones-at-another/.

73. **End-to-End interest mailing list.** September 2009 Archives. [Online: 30.12.2011] http://mailman.postel.org/pipermail/end2end-interest/2009-September/thread.html.

74. **J. Gettys.** Mitigations and Solutions of Bufferbloat in Home Routers and Operating Systems. [Online: 30. 12 2011.] http://gettys.wordpress.com/2010/12/13/mitigations-and-solutions-of-Bufferbloat-in-home-routers-and-operating-systems/.

75. **J**. **Gettys.** Bufferbloat in 802.11 and 3G Networks. [Online: 30.12.2011] http://gettys.wordpress.com/2011/01/03/aggregate-Bufferbloat-802-11-and-3g-networks/.

76. **J**. **Gettys.** RED in a Different Light. [Online: 30. 12 2011.] http://gettys.wordpress.com/2010/12/17/red-in-a-different-light/.

77. **R. S. Prasad, M. Jain, C. Dovrolis.** *On the Effectiveness of Delay-Based Congestion.* Proceedings of Second International Workshop on Protocols for Fast Long-Distance Networks. 2004.

78. **D. J. Leith, L. L. H. Andrew, T. Quetchenbach, R. N. Shorten.** *Experimental evaluation of delay/loss-based TCP congestion.* Proc. PFLDnet. 2008.

79. **J. Martin, A. Nilsson, I. Rhee.** *Delay-based congestion avoidance for TCP.* IEEE/ACM Transactions on Networking, Volume 11, pp. 356-369. Issue 3, June 2003. 10.1109/TNET.2003.813038.

80. **BitTorrent Inc.** µTP Documentation. [Online: 30.12.2011] http://www.utorrent.com/help/documentation/utp.

81. **A. Norberg.** uTorrent transport protocol. [Online: 30.12.2011] http://bittorrent.org/beps/bep_0029.html. BEP 29.

82. **Wikipedia.** Active queue management. [Online: 30.12.2011] http://en.wikipedia.org/wiki/Active_queue_management.

83. **S. Floyd, V.Jacobson.** *Random early detection gateways for congestion avoidance.* IEEE/ACM Transactions on Networking, Vol. 1, pp. 397 - 413. 1993. 1063-6692, 10.1109/90.251892.

84. **H. Sawashima, Y. Sunahara.** *Characteristics of UDP packet loss: Effect of tcp traffic.* Proceedings of INET ’97: The Seventh Annual Conference of the Internet. Kuala Lumpur, Malaysia. June 1997.

85. **Wikipedia.** Random early detection. [Online: 30. 12 2011.] http://en.wikipedia.org/wiki/Random_early_detection.

86. **J. Chroboczek.** Stochastic Fair Blue for the Linux kernel. [Online: 30.12.2011] http://www.pps.jussieu.fr/~jch/software/sfb/.

87. **D. Ardelean, E. Blanton, M. Martynov.** *Remote active queue management.* Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video NOSSDAV '08. 2008. 10.1145/1496046.1496052.

88. **K. Ramakrishnan, S. Floyd, D. Black.** *The Addition of Explicit Congestion Notification (ECN) to IP.* RFC 3168. September 2001.

89. **Wikipedia.** Explicit Congestion Notification. [Online: 30.12.2011] http://en.wikipedia.org/wiki/Explicit_Congestion_Notification.

90. **A. Medina, M. Allman, S. Floyd.** *Measuring interactions between transport protocols and middleboxes.* Proceedings of the 4th ACM SIGCOMM conference on Internet measurement IMC '04. 2004. 10.1145/1028788.1028835.

91. **S. Floyd, E. Kohler, J. Padhye.** *Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC).* RFC 4342. March 2006.

92. **R. Stewart, M. Tuexen, X. Dong.** *ECN for Stream Control Transmission Protocol (SCTP).* IETF draft-stewart-tsvwg-sctpecn-01. July 2011.

93. **G. Ye, T.N. Saadawi, M. Lee.** *On Explicit Congestion Notification for Stream Control Transmission Protocol in Lossy Networks.* Cluster Computing, Vol. 8. Issue 2-3, July 2005. 10.1007/s10586-005-6180-x.

94. **M. Westerlund, I. Johansson, C. Perkins, P. O'Hanlon, K. Carlberg.** *Explicit Congestion Notification (ECN) for RTP over UDP*. IETF draft-ietf-avtcore-ecn-for-rtp. October 2011.

95. **M. Westerlund, I. Johansso, C. Perkins.** ECN for RTP over UDP/IP. Ericsson, University of Glasgow. [Online: 30.12.2011] http://www.ietf.org/proceedings/75/slides/tsvarea-3.pdf.

96. **M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan.** *Data center TCP (DCTCP).* ACM SIGCOMM Computer Communication Review - SIGCOMM '11, Volume 41. Issue 4, August 2011. 10.1145/2043164.1851192.

97. **M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan.** *DCTCP: Efficient Packet Transport for the Commoditized Data Center*. Microsoft Research, 2010.

98. **S. Floyd, M. Handley, E. Kohler.** *Problem Statement for the Datagram Congestion Control Protocol (DCCP).* RFC 4336. March 2006.

99. **E. Kohler, M. Handley, S. Floyd.** *Datagram Congestion Control Protocol (DCCP).* RFC 4340. March 2006.

100. **L. Ong, J. Yoakum.** *An Introduction to the Stream Control Transmission Protocol (SCTP).* RFC 3286. May 2002.

101. **R. Stewart.** *Stream Control Transmission Protocol.* RFC 4960. September 2007.

102. **R. Rajamani, S. Kumar, N. Gupta.** *SCTP vs. TCP: Comparing the performance of transport protocols for web traffic.* University of Wisconsin-Madison, May 2002.

103. **Y. Gu, R. L. Grossman.** *UDT: UDP-based data transfer for high-speed wide area networks.* Computer Networks, Volume 51, Issue 7, pp. 1777-1799. 16 May 2007. 1389-1286, 10.1016/j.comnet.2006.11.009.

104. **Y. Gu.** *UDT: UDP-based Data Transfer Protocol.* IETF draft-gg-udt. April 2010.

105. **Wikipedia.** UDP-based Data Transfer Protocol. [Online: 30.12.2011] http://en.wikipedia.org/wiki/UDP-based_Data_Transfer_Protocol.

106. **Conference, Supercomputing.** SC07 Challenges. [Online: 30.12.2011] http://sc07.supercomputing.org/?pg=challenges.html.

107. **B. A. Ford.** *Structured streams: a new transport abstraction.* ACM SIGCOMM Computer Communication Review, Volume 37. Issue 4, October 2007. 10.1145/1282427.1282421.

108. **B. A. Ford.** *UIA: A Global Connectivity Architecture for Mobile Personal Devices.* s.l. : Massachusetts Institute of Technology, PhD thesis. September 2008.

109. **Wikipedia.** Fast And Secure Protocol. [Online: 30.12.2011] http://en.wikipedia.org/wiki/Fast_And_Secure_Protocol.

110. **Aspera, Inc.** ASPERA FASP SOFTWARE ENVIRONMENT: TECHNOLOGY CAPABILITIES. [Online: 30. 12 2011]

http://www.asperasoft.com/images/file_pdf/Aspera_Technology_Capabilities_2010.pdf.

111. **Aspera, Inc.** the fasp solution. [Online: 30.12.2011] http://www.asperasoft.com/en/technology/fasp_solution_3/the_fasp_solution_3.

112. **ciscoforce.blogspot.com.** Aspera FASP (Fast and Secure Protocol). [Online: 30.12.2011] http://ciscoforce.blogspot.com/2011/02/aspera-fasp-fast-and-secure-protocol.html.

113. **H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson.** *RTP: A Transport Protocol for Real-Time Applications.* STD 64, RFC 3550. July 2003.

114. **C. Perkins.** *RTP and the Datagram Congestion Control Protocol (DCCP).* RFC 5762. April 2010.

115. **M. Molteni, M. Villari.** *Using SCTP with Partial Reliability for MPEG4 Multimedia Streaming.* Proc. of BSDCon Europe 2002. October 2002.

116. **M. Westerlund, I. Johansson, C. Perkins, P. O'Hanlon, K. Carlberg.** *Explicit Congestion Notification (ECN) for RTP over UDP.* IETF draft-ietf-avtcore-ecn-for-rtp. October 2011.

117. **A. Li.** *RTP Payload Format for Generic Forward Error Correction.* RFC5109. December 2007.

118. **C. E. Shannon.** *A mathematical theory of communication.* Bell System Technical Journal, Vol. 27, pp. 379-423 and 623-656. July and October 1948,

119. **D.J.C. MacKay.** *Information Theory, Inference and Learning Algorithms.* 1st edition. Cambridge University Press. October 2003. 978-0521642989.

120. **Computer Laboratory, University of Cambridge.** Continuous ARQ (TCP) adapting to congestion. [Online: 30. 12 2011.]

http://www.cl.cam.ac.uk/teaching/1011/CompNet/TCP-window.html.

121. **Virginia Tech.** TCP - Error Control. [Online: 30.12.2011] http://www.cs.virginia.edu/~cs458/slides/module15-tcp3V2.pdf.

122. **Wikipedia.** Reed–Solomon error correction. [Online: 30. 12 2011.] http://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

123. **Image Processing Lab, Electronics Department at Politecnico di Torino.** Research @ IPL - Digital fountains. [Online: 30.12.2011] http://www1.tlc.polito.it/sas-ipl/research_digital_fountains.php.

124. **D.J.C. MacKay.** Fountain codes. 2005. [Online: 30. 12 2011.] http://www.inference.phy.cam.ac.uk/mackay/presentations/htmlfountain/.

125. **M. Luby.** *LT codes.* The 43rd Annual IEEE Symposium on Foundations of Computer Science, Proceedings. pp. 271 - 280. 2002. 0272-5428, 10.1109/SFCS.2002.1181950.

126. **A. Shokrollahi,** *Raptor codes.* IEEE Transactions on Information Theory. Vol. 52, Issue 6, pp. 2551 - 2567. June 2006, 0018-9448, 10.1109/TIT.2006.874390 .

127. **M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, L. Minder.** *RaptorQ Forward Error Correction Scheme for Object Delivery.* RFC 6330. August 2011.

128. **A. Shokrollahi, M. Luby.** *Raptor Codes (Foundations and Trends(R) in Communications and Information Theory).* Now Publishers Inc.,. May 2011. 978-1601984463, 10.1561/010000006.

129. **W. Rutherford, L. Jorgenson, M. Siegert, P. V. Epp, L. Liu.** *16 000–64 000 B pMTU experiments with simulation: The case for super jumbo frames at Supercomputing '05.* Optical Switching and Networking, Volume 4, Issue 2, pages 121-130. June 2007. 1573-4277, 10.1016/j.osn.2006.10.001.

130. **Peltotalo, J.** *Solutions for Large-Scale Content Delivery over the Internet Protocol.* Tampere University of Technology, PhD thesis. 2010.

131. **P. Maymounkov, D. Mazières.** Rateless Codes and Big Downloads. Springer Berlin / Heidelberg, pp. 247-255. 2003.

132. **M. Knezevic, V. Velichkov, B. Preneel, I. Verbauwhed.** *On the Practical Performance of Rateless Codes.* International Conference on Wireless Information Networks and Systems. 2008.

133. **A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. Andersen, J. Byers, S. Seshan and P. Steenkiste.** *XIA: An Architecture for an Evolvable and Trustworthy Internet.* HotNets-X. Cambridge, MA, US. November 14-15, 2011. ACM 978-1-4503-1059-8/11/11.

134. **D. G. Andersen, H. Balakrishna, N. Feamster, T. Koponen, D. Moon and S. Shenker.** *Accountable Internet Protocol (AIP).* SIGCOMM'08. Seattle, Washington, US. August 17–22, 2008. ACM 978-1-60558-175-0/08/08.

135. **Qualcomm Inc.** Why Raptor codes are better than TCP/IP for File Transfer. 30. 11 2011. [Online: 30.12.2011] http://www.qualcomm.com/media/documents/why-raptor-codes-are-better-tcpip-file-transfer.

136. **Qualcomm Inc.** RaptorQ Technical Overview. 1. 10 2011. [Online: 30.12.2011] http://www.qualcomm.com/media/documents/raptorq-technical-overview.

137. **IETF.** FEC Framework (fecframe): Description of Working Group. [Online: 30.12.2011] https://datatracker.ietf.org/wg/fecframe/charter/.

138. **M. Watson, A. Begen, V. Roca.** *Forward Error Correction (FEC) Framework.* RFC 6363. October 2011.

139. **M. Luby, V. K. Goyal, S. Skaria, G. B. Horn.** *Wave and equation based rate control using multicast round trip time.* ACM SIGCOMM Computer Communication

Review - Proceedings of the 2002 SIGCOMM conference, Volume 32. Issue 4, October 2002. 10.1145/964725.633044.

140. **J. W. Byers, G. Horn, M. Luby, M. Mitzenmacher.** *FLID-DL: congestion control for layered multicast.* IEEE Journal on Selected Areas in Communications. Vol. 20. Issue 8, October 2002. 0733-8716, 10.1109/JSAC.2002.803998.

141. **J. Widmer, M. Handley.** *TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification.* RFC 4654. August 2006.

142. **V. Subramanian.** *TRANSPORT AND LINK-LEVEL PROTOCOLS FOR WIRELESS NETWORKS AND EXTREME ENVIRONMENTS.* Rensseler Polytechnic Institute, PhD thesis. 2008.

143. **L. Rizzo,** *On the feasibility of software FEC.* DEIT Technical Report, LR-970131. [Online: 30.12.2011] http://citeseer.ist.psu.edu/rizzo97feasibility.html.

144. **T. Anker, R. Cohen, D. Dolev.** *Transport Layer End-to-End Correcting.* The School of Computer Science and Engineering , Hebrew University. Technical report. 2004.

145. **O. Tickoo, V. Subrumanian, S. Kalaynaraman, K.K. Ramakrishnan.** LT-TCP: End-to-End Framework to Improve TCP Performance over Networks with Lossy Channels. *Quality of Service – IWQoS 2005.* Springer Berlin / Heidelberg, 2005.

146. **D. Kliazovich, M. Bendazzoli, F. Granelli.** *TCP-Aware Forward Error Correction for Wireless Networks.* In Proceedings of MOBILIGHT'2010. pp.68-77. 2010.

147. **H. Seferoglu, A. Markopoulou, U. C. Kozat, M. R. Civanlar, J. Kempf.** *Dynamic FEC Algorithms for TFRC Flows.* IEEE Transactions on Multimedia, Vol. 12. Issue 8, December 2010. 1520-9210, 10.1109/TMM.2010.2053840.

148. **A. Botoș, Z. A. Polgar, Z. I. Kiss.** *FECTCP for high packet error rate wireless channels.* 8th International Conference on Communications (COMM), 2010. pp. 327 - 330. Bucharest, RO. June 2010. 10.1109/ICCOMM.2010.5509118.

149. **J. K. Sundararajan, D. Shah, M. Medard, M. Mitzenmacher, J. Barros.** *Network Coding Meets TCP.* IEEE INFOCOM 2009. pp. 280 - 288. Rio de Janeiro, Brazil. April 2009. 0743-166X, 10.1109/INFCOM.2009.5061931.

150. **D. Churms, B.A.A. Olsen, D.J.J. Versfeld.** *Reliable File Transfer Using Forward Error Correction.* SAIEE Africa Research Journal, Vol. 101. No. 4, 2010.

151. **E.H.-K. Wu, M-Z Chen.** *JTCP: jitter-based TCP for heterogeneous wireless networks.* IEEE Journal on Selected Areas in Communications. Vol. 22, pp. 757 - 766. No. 4, May 2004. 0733-8716, 10.1109/JSAC.2004.825999.

152. **C. Huang, C-S Lu, H-K Wu.** *JitterPath: Probing Noise Resilient One-Way Delay Jitter-Based Available Bandwidth Estimation.* IEEE Transactions on Multimedia, pp. 798 - 812. June 2007. 1520-9210, 10.1109/TMM.2007.893343.

153. **C Huang, S. Mehrotra, J. Li.** *A hybrid FEC-ARQ protocol for low-delay lossless sequential data streaming.* IEEE International Conference on Multimedia and Expo, ICME 2009. pp. 718-725. New York, NY, US. 2009. 1945-7871, 10.1109/ICME.2009.5202596.

154. **S. Mehrotra, J. Li, C. Huang.** *RAPID: a reliable protocol for improving delay.* Proceedings of the international conference on Multimedia, MM '10. 2010. 10.1145/1873951.1874259.

155. **S. Mehrotra, J. Li, Y-Z Huang.** *Minimizing delay in lossless sequential data streaming.* IEEE International Conference on Multimedia and Expo (ICME), 2010. pp. 1-6. July 2010. 1945-7871, 10.1109/ICME.2010.5582531.

156. **S. Mehrotra, J. Li, Y-Z. Huang.** *Optimizing FEC Transmission Strategy for Minimizing Delay in Lossless Sequential Streaming.* IEEE Transactions on Multimedia. pp. 1066 - 1076. October 2011. 1520-9210, 10.1109/TMM.2011.2153193.

157. **L. Lopez, A. Fernandez, V. Cholvi.** *A game theoretic analysis of protocols based on fountain codes.* 10th IEEE Symposium on Computers and Communications, 2005. ISCC 2005. Proceedings.. pp. 625-630. June 2005. 1530-1346, 10.1109/ISCC.2005.11.

158. **Qualcomm Inc.** DF Raptor in Military Communications. 27. 09 2010. [Online: 30.12.2011] http://www.qualcomm.com/media/documents/df-raptor-military-communications.

159. **Qualcomm Inc.** The world's most advanced Forward Error Correction (FEC) for data networks. [Online: 30.12.2011] http://www.qualcomm.com/solutions/broadcast-streaming/media-delivery.