UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

TCP-in-UDP

Enabling gradually deployable TCP coupled congestion control using an efficient UDP encapsulation

Kristian A. Hiorth

Masteroppgave våren 2016

TCP-in-UDP

Kristian A. Hiorth

21st May 2016

Abstract

This thesis recalls the desirability of being able to apply congestion control coupling to concurrent TCP connections between the same end hosts. We identify challenges that must be overcome to provide practically deployable solutions to this end, chiefly the presence of multi-path routing mechanisms, such as Equal Cost Multi-Path Routing (ECMP) in the network. Additionally, we identify some inherent weaknesses in previous proposals for TCP congestion control coupling. We contribute a novel design for a TCP-in-UDP encapsulation scheme which is able to work around the problems created by multi-path routing, as well as delivering other advantages, which we implement in the FreeBSD kernel. In order to be able to test and evaluate this mechanism, we also present and implement a design for TCP congestion control coupling based on the Flow State Exchange architecture. Finally, we carry out an evaluation of the combinations of these two solutions, and find that they yield tangible performance benefits in terms of delay and packet loss reductions.

Contents

List of Figures

Preface

Acknowledgments

First of all, I would like to express my gratitude to my two excellent supervisors, Michael Welzl and Safiqul Islam. You both went far beyond the call of duty as master thesis supervisors and have taught me so much. Thanks to all the members of the Networks and Distributed Systems (ND) research group at the Department of Informatics for making the group such a great, stimulating and congenial environment.

In particular, I would like to thank Runa Barik and David Hayes for many encouraging, interesting, elucidating and just plain fun conversations. Simone Ferlin-Oliveira at Simula Research Laboratory / Bell Labs provided me with some helpful tips with respect to generating self-similar cross-traffic in another setting, which I was able to put to good use in the evaluation work for this thesis as well.

In general, I am grateful to all the members and contributors in the FreeBSD and wider Free Software communities who collaborate, and have collaborated for decades, to provide excellent software for the benefit of all.

I would like to thank my parents for encouraging and supporting me in this endeavour.

Finally, I dedicate this thesis to my beloved Eva, Vidar, Viktor and Morris. I couldn't have done this without your unending support!

Chapter 1

Introduction

1.1 Problem statement

The Transmission Control Protocol (TCP) provides a congestion control mechanism. This mechanism crucially enables TCP to dynamically adjust its sending rate to actual network conditions, the aim being to send as fast as possible without causing disruptions (i.e. congestion).

However, congestion control is applied independently to each and every connection, even if they are between the same host and destination pair. Intuitively, this seems sub-optimal, as such parallel connections logically ought to share the same network path, and thus be subject to identical network conditions.

In practice, complications arise. Due to network mechanisms like Equal Cost Multi-Path (ECMP) routing, one cannot be sure that such parallel connections actually do use the same paths through the network. This multi-path forwarding can break assumptions made by TCP congestion control, both about packets normally being delivered in order and packets experiencing similar round trip times, which has been shown to have adverse effect on TCP performance [55].

1.2 Contributions

Through this master thesis project we have devised the design of, implemented and evaluated the TCP-in-UDP encapsulation mechanism which can work around these difficulties in order to allow congestion control to be coupled for parallel connections, despite running over otherwise uncooperative networks. This work was inspired by previous work by Cheshire et al. [19], but the present encapsulation method and implementation is novel and intended to serve different goals.

To demonstrate the potential benefits afforded by TCP with coupled congestion control (CCC), we have also implemented and evaluated a proof of concept CCC mechanism for TCP, contributing to ongoing work springing out of the Flow State Exchange (FSE) concept pioneered by Islam et al. [48]. We have been active in the design effort involved in applying the

FSE to TCP, and present the first practical implementation of a TCP FSE, as well as the first published evaluation results concerning this mechanism.

1.3 Research questions

Overall: *Can TCP congestion control coupling be made deployable across the Internet, without changes to the network? Will doing so actually be beneficial?*

Multi-path forwarding

> Can multi-path forwarding be worked around by a judicious encapsulation method?

TCP-in-UDP

> Can such an encapsulation be designed so as not to waste either network bandwidth nor end host processing power?

TCP congestion control coupling

> How can TCP congestion control coupling be implemented in an actual production network stack?

Interface

> How can a TCP-in-UDP + TCP congestion control coupling scheme be activated on demand in the least disruptive way possible, without hurting latency and stability, while still affording flexibility and control to both application developers and administrators?

Performance

> Will such a scheme deliver the increase in performance predicted by simulation data in earlier works? Can it deliver other benefits?

1.4 Organization

The remainder of this thesis is organized as follows: chapter 2 will present some background information on the most relevant concepts, including a look at related works, providing the context for our work.

Then we will describe the design of the two solutions we have developed. Chapters 3 and 4 give a higher-level design point of view, discussing the architecture and mechanisms of the TCP-in-UDP encapsulation and Flow State Exchange coupled congestion control, respectively.

In chapter 5 we follow up with an account of the implementation itself and its particularities.

Chapter 6 presents our evaluation test setup and results, with analysis.

Finally, in chapter 7, we first summarize how we have answered the research questions, before presenting further possibilities for refinements and research avenues and wrapping up the discussion.

Chapter 2

Background

We will now give an overview over the most relevant concepts and related works. Section 2.1 presents an overview of TCP congestion control. In section 2.2 we discuss ECMP and multi-path issues in modern networks in general, with an emphasis on the challenges posed for coupled congestion control. Then, section 2.3 affords a look at some of the most common and relevant types of middleboxes we are likely to have to interact with. Finally, in section 2.4, we discuss coupled congestion control, including related work in section 2.4.2.

2.1 TCP Congestion Control

Originally, TCP only specified flow control, moderating sender behaviour so as not to overwhelm the receiver's buffer. Flow control does not, however, prevent a sender, or indeed *all senders* from overloading the network connecting the end hosts.

TCP Congestion Control was developed primarily by Van Jacobson at Berkeley, as a consequence of severe congestion events suffered by the growing Internet of the mid-to-late 1980's [50]. The aim of congestion control is to control sender behaviour so as not to overload the capacity of the transmission network. In IP networks, this capacity is bounded by the link with the lowest available capacity on the forwarding path from sender to receiver – the *bottleneck link*. In practice, the bottleneck might move dynamically between different links as their load varies.

Section 2.1.1 details the original algorithms that extended TCP with congestion control. Next, we discuss the many enhancements that have been made over the last two and a half decades, leading to an overview of the state of the art in section 2.1.2. Finally we look at some of the most important principles underpinning the design of TCP congestion control, which we must take into consideration in our effort to couple it, in section 2.1.3. Throughout these section, we will also comment on how TCP is implemented in the FreeBSD operating system, since we have carried out our implementation work in FreeBSD.[1]

[1]As a historical aside, FreeBSD's TCP implementation directly descends from the

2.1.1 Early Algorithms

TCP congestion control initially consisted of the Slow-Start and Congestion Avoidance algorithms, along with improved retransmission timeout computation. This revision of TCP is referred to as TCP "Tahoe", named after the 4.3BSD release it originally appeared in.

In addition to the flow control receiver window (`rwin`), a congestion window (`cwnd`) was added, which is used to limit the rate at which packets are sent when starting a new connection or recovering from packet loss. The amount of data that is allowed to be in flight at any time is limited to $min(cwnd, rwin)$.

The algorithm is self-clocking, relying upon the so-called *ACK clock*, i.e. the control loop is normally updated upon the arrival of ACKs.

Jacobson formulated an informal stability criterion for TCP, the "conservation of packets" principle inspired by flows in physics: a stable flow in steady-state, with a full window of data in flight, should exhibit a conservative behaviour, not injecting a new packet into the network until an old packet has left. [50] He identifies three manners in which this principle might be violated:

- The connection never reaches this equilibrium steady-state.
- Senders placing new packets into the network too soon, before an old one has left it.
- The equilibrium state is unreachable because of resource limitations in the network.

These issues are addressed by slow-start, better round-trip time estimation and congestion avoidance, respectively.

Slow-Start

As stated in [50], slow-start is intended to quickly allow the congestion controller to reach an equilibrium steady-state and establish the ACK clock.

Slow-start introduces the slow-start threshold (`ssthresh`), which dictates when to operate in this mode; if `cwnd` is below `ssthresh`, the connection is in slow-start mode and is updated according to the following rules:

1. Initially, set `cwnd` equal to Sender Maximum Segment Size (SMSS), i.e. allow one single segment to be sent.
2. If the connection is just starting up, initialize `ssthresh` to "infinity", in practice usually the maximum possible window size.
3. For each ACK received, increment `cwnd` by SMSS, until

original TCP/IP reference implementation by the University of California at Berkeley. Its lineage can be traced back all the way to 1982, and there are still many parts of the implementation which have changed little since FreeBSD imported the 4.4BSD-Lite source code in 1994. [85] BSD stands for Berkeley Standard Distribution.

- $cwnd > ssthresh$: switch to congestion avoidance mode.
- a loss is detected: set $ssthresh = cwnd/2$ and go back to step 1.[2]

This update regime is obviously not very slow at all, in fact `cwnd` grows exponentially during slow-start, doubling once per RTT. However, without slow-start, one could simply uncontrollably burst out however many segments permitted by the flow control window all at once when starting out a connection or resuming after an idle period, sending a quite possibly large traffic spike through the network, leaving no time for other senders to react before congestion would ensue. One could consider slow-start to be a simplistic mechanism for yielding a rapid ramp-up of the sending rate.

Round-Trip Time estimation

In the congestion control algorithm as originally presented in [50], loss was detected solely by relying on the pre-existing Retransmission Time-Out (RTO) timer. The RTO delay is calculated dynamically based on Round Trip Time (RTT) measurements, i.e. the delay between sending a segment and receiving an acknowledgment (ACK) of its reception from the receiver. The original TCP specification [RFC793] gives the following suggestions for an algorithm: [3]

$$sRTT = \alpha * sRTT + (1 - \alpha) * RTT \tag{2.1}$$
$$RTO = \beta * sRTT \tag{2.2}$$

where $sRTT$ is the smoothed RTT estimate, α a constant smoothing factor, RTT the latest RTT measurement sample, RTO the updated timeout value of the RTO timer and β a constant delay variance factor.

Jacobson contributes a better RTT-estimation algorithm, crucially extending it to also dynamically estimate the variation in RTT and taking that estimate into account when computing the RTO. Equation (2.2) used a constant factor to account for variation, which had a suggested value between 1.3 and 2.0, which is a poor choice because both the delay and its variance increase rapidly with load, leading to more and more unnecessary timeouts as the network load grows, further contributing to load it even more in a kind of positive feedback loop. The improved algorithm can be expressed as follows: [50, 81]

$$\sigma = (1 - \beta) * \sigma + \beta * (sRTT - RTT) \tag{2.3}$$
$$RTO = sRTT + 4 * \sigma \tag{2.4}$$

where σ is the (estimated) variation in RTT.

[2]The window update is technically part of the congestion avoidance algorithm, but it would be disingenuous not to describe it here due to the effect it has on slow-start.

[3]In [RFC793], the RTO formula in fact also specifies clamping the value within reasonable lower and upper bounds; we have left this out for clarity's sake.

Furthermore, on an RTO firing, its value is subjected to exponential back-off, applying exponential damping to sender behaviour which ensures stability of the network.

Congestion Avoidance

After the initial slow-start phase, the system enters Congestion Avoidance (CA), governed by Additive Increase, Multiplicative Decrease (AIMD) of the window size. This makes senders highly sensitive to congestion signals, while being more conservative in their probing of bandwidth (through sending packets faster and faster), allowing for the timely reception of congestion signals and stabilization of network load.

The following equations describe the behaviour of CA:

$$cwnd = cwnd + 1/cwnd \tag{2.5}$$

$$cwnd = cwnd/2 \tag{2.6}$$

Equation (2.5) is the additive increase part, whereby the congestion window is opened upon reception of an ACK. Equation (2.6) represents the sender's reaction to loss. The congestion window is halved – multiplicative decrease by a factor of 0.5. In practice, when combining the slow-start and CA algorithms TCP "Tahoe" actually sets `ssthresh` to this value and initiates a slow-(re)start by resetting the `cwnd` value to $1 * SMSS$. The rationale behind this is to restart the ACK clock after it has been disturbed by the loss event and associated pause in clocking out new segments.

Fast Retransmit / Fast Recovery

Fast Retransmit TCP "Tahoe" actually included a further refinement: Fast Retransmit. The absence of an ACK is in fact not the only way in which a sender can detect a loss; [RFC1122] specifies that a receiver may[4] respond with an ACK to incoming segments received out-of-order. This entails that if some segment with sequence number[5] x_i is lost in transit, but subsequent segments $x_{i+1}, x_{i+2}, ...$ which were in flight just "behind" the unlucky one do make it to the receiver, the receiver will respond by repeatedly acknowledging the highest sequence number it has received *in order* so far, i.e. x_{i-1}. Back at the sender these duplicate ACKs (dupACKs) are an implicit signal that either of two events has occurred: either segment x_i has been lost, or there has been some packet reordering in the network causing later segments to arrive first at the receiver. Significant reordering events were considered far less likely than packet loss in those early days of the Internet, so a simple threshold-based heuristic was adopted: if more

[4]At the time, fast retransmit was still an experimental algorithm, hence the weak recommendation. The current TCP congestion control overview specification in [RFC5681] states that receivers "SHOULD send an immediate duplicate ACK when an out- of-order segment arrives".

[5]TCP really counts sequence numbers in terms of byte positions within the byte-stream, but we will count in whole segments for the sake of clarity.

than a certain threshold (which was later fixed to three in [RFC2581]) consecutive dupACKs are received back to back, the sender considers it an indication of loss and immediately retransmits the first outstanding segment without waiting for the RTO timer to expire; this algorithm is called *fast retransmit*.

Fast Recovery Furthermore, since ACKs are indeed being generated, the sender can also infer that at least that many segments beyond x_i were in fact successfully transmitted. If the dupACK threshold is small in relation to the Bandwidth Delay Product (BDP) of the connection, which is generally the case, this means that there are still a large number of segments in transit, enough to keep the ACK clock ticking. Therefore, it is unnecessary to go into slow-start as part of the loss response, and `cwnd` can instead be kept at roughly half of its previous value before entering a fast recovery phase. This phase works as follows:

1. Set $ssthresh = max(cwnd/2, 2 * MSS)$ and perform a fast retransmit of the lost segment. Set $cwnd = ssthresh + 3 * MSS$. The constant part compensates for the fact that we know three segments have been received and cached by the receiver (or we wouldn't have received three dupACKs).

2. In the same vein, each additional dupACK inflates `cwnd` by one segment, since these indicate that yet another packet has left the network. If the window allows it, clock out a new segment.

3. On receiving the next ACK that acknowledges new data, leave fast recovery by setting $cwnd = ssthresh$, deflating the window to where congestion avoidance decreased it to upon loss. This ACK is expected to be an acknowledgment of the retransmitted segment from step 1, and should acknowledge not only the lost segment, but also all segments transmitted up until the reception of the third dupACK.

This is called the *fast recovery* algorithm, and even though it was proposed together [51] with fast retransmit, the implementation only came in the next 4.3BSD release, "Reno". TCP with the combination of all four of the congestion control algorithms we have discussed until now (slow-start, congestion avoidance, fast retransmit and recovery) is informally referred to as TCP "Reno" and was specified in [RFC2001]. The fast retransmit/fast recovery combination ensures that the ACK clock is kept ticking happily away even in the face of having to retransmit a segment, and additionally the fast recovery phase prevents the congestion controller from reacting to losses more than once per RTT. That said, it is far from certain that recovery will be successful - if the ACK in step 3 is not received in a timely manner, the RTO timer will expire and prompt a regular loss response in the form of slow-start.

2.1.2 Congestion Control Evolution

TCP Congestion Control has been constantly evolving since its inception to this day. We now summarize some of the most important milestones and provide an overview of the state of the art.

Fast Recovery Advances

"New Reno" As shown by Hoe in [43], the fast retransmit/fast recovery algorithms only properly cope with a single loss per window, yet multiple losses from a single window are in fact highly likely to occur, particularly during the start-up transient phase of the connection. The author's recommendations in that thesis and later in [42] provided the impetus for improvements to fast recovery which would culminate in TCP "New Reno". Originally specified in [RFC2582] and updated by [RFC6582], it further improves on this mechanism by differentiating between partial ACKs, which only acknowledge some of the segments sent before triggering Fast Recovery, and full ACKs which acknowledge all outstanding data. Fast Recovery will only end in the latter case, allowing for multiple fast retransmissions of distinct lost segments. The precise change to the original "Reno" algorithm is as follows:

- In the first step, the highest sequence number transmitted up until that point in time is stored in a variable called `recover`.
- In step 3, a partial or full ACK is distinguished by checking whether it acknowledges all outstanding data up to `recover`, inclusive.
  - In case it does not, this is a partial ACK, and we may assume that it actually points to another "hole" in the sequence of received segments. Therefore, the first unacknowledged segment is retransmitted, `cwnd` is partially deflated according to how much outstanding data was just acknowledged and the algorithm jumps to step 2, re-inflating the window by a segment and if possible clocking out another. Thus the fast recovery phase continues.
  - In case it did, this is a full ACK and we have recovered from the loss event. Return to congestion avoidance, in either of two manners:
    1. Deflate the congestion window by setting $cwnd = min(ssthresh, FlightSize + MSS)$ where $FlightSize$ is the amount of currently outstanding (sent, but not yet acknowledged) data.
    2. Deflate the congestion window like in "Reno", by setting $cwnd = ssthresh$. However, "New Reno" specifies that implementations opting for this variant should take measures to protect against suddenly sending a large burst of data into the network.[6]

[6]As we shall see, FreeBSD performs a kind of slow-start to accomplish this.

SACK The Selective ACKnowledgment (SACK) TCP option [RFC2018] enables receivers to explicitly inform the sender of which segments have actually been received, enabling it to perform multiple precisely targeted retransmissions per RTT in fast recovery instead of only being able to detect one hole each RTT.

By leveraging this information, the performance of the fast recovery and of TCP in general can greatly be improved.

"New Reno" with optional conservative SACK recovery [RFC3517, 31] is the TCP congestion control variant used by default in current versions of FreeBSD, and is the base we use to implement our modifications. [7]

Explicit Congestion Notification

An Explicit Congestion Notification (ECN) mechanism [RFC3168] has also been introduced, which enables routers to explicitly announce that they are experiencing unsustainable load, rather than rely on indirect observations such as packet loss and delay measurements.

ECN works by using the two least significant bits of what was the Type of Service field in the IP header to both signal ECN support to routers, and for letting routers mark packets as having experienced congestion. When an ECN-enabled router would otherwise have dropped a packet, it can instead mark it with the value Congestion Experienced (CE) if that packet carries a mark signifying ECN Capable Transport (ECT), which would have been set by an originating host that has ECN enabled.

While the actual ECN marking happens in the IP header, the receiver of a CE-marked packet must echo this signal back to the sender using features of the transport protocol. In TCP, two formerly reserved bits were assigned as ECN-related flag values: ECN-Echo (ECE) and Congestion Window Reduced (CWR). On receiving an ECE-marked packet, the receiver TCP responds by setting the ECE flag to 1 on all outgoing segments (including pure control messages) on the same connection. It keeps on doing so until the sender affirms that the ECN signal has been received and reacted upon by setting CWR to 1 on a new packet in the forward direction. An ECN negotiation protocol using TCP options was also specified, allowing hosts to dynamically enable ECN on a per-connection basis.

Active Queue Management

To be able to mark instead of dropping packets, it is clear that routers must detect and react to congestion before it becomes so pressing that the buffers overflow and experience "tail drops". Taking into account the fact that queuing packets inherently also means delaying them, it becomes obvious that there are significant benefits to such an early reaction, which would serve to curb the size of queues, not least due to TCP's RTT-limited performance [60, 68].

[7] There has since been a revision in [RFC6675], but FreeBSD has not implemented it at the time of writing according to [21].

Active Queue Management (AQM) [RFC2309] is the general term for queue management disciplines that are more advanced than simply setting a queue limit. One of the best known AQM schemes is Random Early Detection (RED) [33], which will randomly drop or mark incoming packets when the average queue size exceeds a certain target threshold. Compared to traditional drop-tail queueing, this approach confers several advantages:

- avoiding *global synchronization* where many independent flows experience losses at the same time, subsequently all reducing their sending rate simultaneously. This leads to under-utilization of the link, and may further lead to a phase effect when all those connections again exceed the available bandwidth at approximately the same time, assuming their RTTs are relatively similar.

- *fairness* between flows exhibiting different behaviours; drop-tail queues will tend to impact bursty flows more than flows which send at a steady rate, even if that rate is very high.

A number of more sophisticated AQMs have since been proposed, including Adaptive RED [32], PIE [69] and (FlowQueue-)CoDel [44, 64]. Modern AQMs mostly auto-tune their parameters, while the original RED required difficult configuration to achieve good performance, and have started to see some deployment [44].

Alternative algorithms

Many novel congestion control algorithms for TCP have surfaced over the last two decades, such as Westwood [59], CUBIC [37], Proportional Rate Reduction [26, RFC6937] and Data center TCP [2]. Many of these try to make better use of the information at hand (such as delay, ACK rate, etc.) to be able to avoid congestion even before a loss happens. In this work, we will only consider New Reno behaviour.

For a more in-depth explanation of TCP congestion control behaviour, see e.g. [81].

Initial Window increase

The Initial Window (IW) size, which is the value `cwnd` is initialized to at the start of a connection, has been raised by the IETF twice. As of 2002 [RFC3390] increased it $min[4 * MSS, max(2 * MSS, 4380)]$, in practice usually yielding 3 segments. 11 years later, it was experimentally further increased to $min[10 * MSS, max(2 * MSS, 14600)]$, i.e. 10 segments at MTU 1500, by [RFC6928]. Note that both of these documents specify an upper bound, implementations may choose to use a lower value, for example if the link is known to have a limited capacity. Higher IW matters especially much with respect to Web use cases, allowing a large proportion of requests to be completed within a single RTT, which was also the main rationale behind [RFC6928].

2.1.3 Principles

Reaction to loss

TCP detects loss in two ways: 1) by expiration of the Retransmission TimeOut (RTO) timer and 2) by reception of a certain number of duplicate ACKs (dupACKs), i.e. ACKs that acknowledge the same segment. The RTO duration is dynamically computed based on the RTT estimate, in order to achieve a sensible trade-off between reaction time in case of loss and low probability of false positives.

When congestion arises, overflowing a buffer, it is likely that several segments in flight at the same time will meet a full buffer and be dropped. However, it would be wrong to treat these as separate events, since a proper reaction to the first of these losses ought to alleviate the load on the bottleneck. Indeed, TCP's multiplicative decrease loss reaction is already rather conservative. Thus, in order to attain stability, TCP congestion control is designed to only react once to a particular *loss event*, as defined by loss during one single window (RTT duration).

The fast recovery (FR) algorithm ensures this behaviour, since the only further reaction to loss while already in FR consists of one of two things: 1) a new hole in the sequence number space is discovered, accompanied by a further retransmission, or 2) an RTO timer fires, causing a slow-start. FR normally lasts at least for a single RTT, since that is the time it takes for the initially retransmitted segment to arrive and be acknowledged.

Spurious timeouts

It is important to note that there is an underlying assumption that a timeout shall only happen when all outstanding packets have been lost, interrupting the ACK clock. If we receive an ACK for the original transmission of a segment after its RTO has expired, this means there was no real loss event (on the forward path at least), but that there was for some reason a delay in delivering the ACK. This is called a spurious timeout, and TCP's conservative loss reaction on RTO expiry – return to slow-start – can severely degrade performance upon such timeouts. Several algorithms have been devised to detect these so the CC state can be rectified.

FreeBSD currently has a simple heuristic implemented for detecting spurious timeouts, due to Allman and Paxson [3]: if an ACK for a retransmitted segment is received within $\frac{1}{2}RTT$ of the retransmission time, it is assumed that the timeout was in error, and the congestion state of the connection is reset to how it was before the RTO.

In response to the increasing prevalence of wireless link layer technologies, which tend to exhibit significant delay variations, several more sophisticated algorithms for detecting such spurious timeouts have been proposed, including Eifel [58, RFC3522], Forward RTO-Recovery (F-RTO) [74, RFC5682] and D-SACK [15, RFC3708]. In order to illustrate how these mechanisms work around the so-called *retransmission ambiguity problem* of not being able to distinguish between ACKs for initial transmissions and retransmissions of a certain segment [53], we will briefly describe the Eifel algorithm.

Eifel takes advantage of the TCP timestamp option [RFC7323], and requires them to function[8]. When a retransmission takes place, the corresponding timestamp is stored. When the first ACK covering the retransmitted segment is received, its timestamp is compared with the stored one; if the ACK timestamp is older, which by the TCP timestamp semantics indicates that the ACK was sent before receiving the retransmission, then the transmission is considered spurious. On detecting a spurious timeout, the sender either resets the CC state to how it was before or moderates the retransmission behaviour depending on how many segments were already retransmitted at that point.

Perhaps somewhat surprisingly, given the fact that Eifel was originally implemented on top of FreeBSD's TCP, FreeBSD does not support it; the code was unfortunately never merged into the upstream project.

Aggregate behaviour

Currently deployed standard TCP congestion control is entirely connection-oriented, and does not explicitly take into account the possibility of several parallel connections between the same two hosts. The dynamic behaviour of a pack of such independent congestion controllers can be quite disruptive and unfair amongst each other, especially until they reach steady-state. Section 2.4 will further explain the motivation and challenges for coupling such connections.

2.2 Multi-Path routing

ECMP [RFC2991] is a routing technique which allows packets to be forwarded along multiple paths as long as they have the same cost metric – allowing to share traffic load across "tied" routes, so to speak. It has gained widespread adoption since it allows for load balancing of networks and better utilization of link resources. Link Aggregation [46] is a similar link layer mechanism, which permits several point-to-point links between the same two machines – referred to as component links – to be aggregated into a single Link Aggregation Group (LAG), which the end hosts can treat as if it were a single link. This permits increased bandwidth, load balancing and automatic fail-over in the event of a component link going down.

Unfortunately, these mechanisms also make it difficult to infer path-specific attributes by observing single connections and extrapolating them to other connections that one would logically assume be routed along the same path, i.e. connections that share the same source and destination IP addresses.

[8]In [58], the authors note that it could be implemented by any other means of distinguishing an ACK for a duplicate received segment.

2.2.1 Flow identification

As we shall discuss further below, ECMP and LAG implementations will usually attempt to minimize the probability of inducing reordering within individual packet flows. How such flows are classified, however, is not entirely obvious.

Header five-tuple

While there is no standard that explicitly requires it, a number of sources provide strong indications that most ECMP and LAG equipment will identify and equally treat flows based on the five-tuple consisting of the following header fields:

$$\begin{aligned}(&\langle \text{IP destination address} \rangle,\\ &\langle \text{IP source address} \rangle,\\ &\langle \text{Transport destination port} \rangle,\\ &\langle \text{Transport source port} \rangle,\\ &\langle \text{IP protocol number} \rangle)\end{aligned}$$

This behaviour is described in several documents:

- [RFC2991] and [RFC2992] warn about the disruption potential of load balancing algorithms that induce packet reordering, such as round-robin, given the well-known performance issues caused by reordering on TCP [13, 55].
- [RFC6437] and [RFC6438] claim that use of the five-tuple is an operation reality: "In practice, many implementations [of ECMP/LAG] use the 5-tuple dest addr, source addr, protocol, dest port, source port as input keys to the hash function, to maximize the probability of evenly sharing traffic over the equal cost paths." [RFC6438, p. 2].
- Vendor documentation such as [79] generally supports this assumption and affirms that default behaviour is to forward flows over a single link.

IPv6 flow label

IPv6 specifies an explicit flow identification field, the flow label [RFC6437]. It is intended to allow flows to be identified only by comparing fields in the fixed part of the IPv6 header, which is more efficient for routers and other network layer devices, especially considering the fact that IPv6 allows chaining extension headers of variable size. That makes it more difficult for routers to look up transport protocol fields, since they may be located at a variable offset from the beginning of the IP datagram. Another goal is actually to enhance interactions with ECMP and LAG, as recommended in [RFC6438].

The flow label is a 20-bit field, and if its value is zero the datagram is considered unlabeled. If a value has been set, then it must be delivered intact end-to-end; middleboxes are not allowed to interfere with it. When in use, flows can be identified by the three-tuple consisting of:

$$\begin{aligned}(&\langle\text{IP destination address}\rangle,\\ &\langle\text{IP source address}\rangle,\\ &\langle\text{IP flow label}\rangle)\end{aligned}$$

In practice, it is highly uncertain to what degree routers actually respect this marking at the time of writing.

2.3 Middleboxes

There exist a great many sorts of mechanisms in the network that intercept and modify traffic. They are usually referred to collectively as Middleboxes, which [RFC3234] defines as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host." Examples of such contraptions are firewalls, traffic shapers, load balancers and so on.

2.3.1 Network Address Translation

One of the undoubtedly most common kind of middleboxes is the Network Address Translator (NAT) [RFC2663]. NAT is a traffic rewriting and multiplexing technique which is predominantly used to share a single or a few globally routable IP addresses between a larger set of hosts. Since appearing in the late 1990's, it has become almost ubiquitous in access networks that connect end-users (both consumers and enterprises) to the Internet due to the increasingly precarious shortage of IPv4 address space.

NAT functions by intercepting packets at a gateway between the local domain and the rest of the network, rewriting addresses and ports while maintaining a mapping of current connections. Thus, local hosts may be assigned IP addresses in private address space, and only the NAT gateway's externally facing network interface needs a globally routable address.

However, this approach poses a problem when establishing new connections. If the translator has not seen outbound packets in a flow, it does not have any mapping of which local host traffic is destined for. This affects both passive ends of TCP connections (listening servers) and UDP endpoints. To overcome this, one can configure static translation rules at the gateway, or attempt the connection in reverse.

The problem is compounded if both hosts attempting to connect to each other are behind NAT, since neither of them can in that case successfully receive a packet from the other end without access to configure the gateway. Mechanisms using a third party rendezvous server have been developed to work around this problem, such as Interactive Connectivity Establishment (ICE) [RFC5245], which leverages the Session Traversal Utilities for NAT (STUN) [RFC5389] and Traversal Using Relays around NAT (TURN) [RFC5766] protocols. Initially designed to facilitate Session Initial Protocol (SIP) [RFC3261] call establishment across NATs, ICE has been extended both to other real-time media use cases and to TCP connection establishment [RFC6544]. These work by having both endpoints coordinate their connection establishment via the rendezvous (RV) point, which is a server known to both endpoints in advance. The endpoints can learn of their public IP addresses, as well as establish and infer NAT port mappings when communicating with the RV server. The RV server distributes this information to the other endpoint, and the ICE system generates a list of candidate solutions for connection establishment according to defined rules and policies, which are attempted one after the other until the connection succeeds. In the worst case, it is possible to communicate using a relay server using TURN if none of the candidates lead to a successful connection establishment.

2.3.2 Network Security Appliances

Following the Internet's widespread adoption and commercialization, the lack of emphasis on security in its original architecture and design [11] has prompted the development of a diverse range of network security appliances. Perhaps the most widespread of these is the firewall, a policy enforcement tool located on a (sub-)network border. These perform packet filtering with variable depths of inspection, ranging from simple network-layer origin and destination checks to complete application-layer protocol inspection (Deep Packet Inspection).

Modern firewalls often have rules for dealing with TCP behaviour, and although measurements performed by Honda et al. [45] indicates that the large majority of middleboxes are compliant with the standard which states that "A TCP MUST ignore without error any TCP option it does not implement, assuming that the option has a length field" [RFC1122], there is still a significant number of them which either drop connections or remove the options in transit when they encounter an unknown (to the middlebox implementation) option. In fact, there is an entire class of network security appliance, scrubbers [80], that actively normalize protocol behaviour. While most of these appear not to hamper connectivity, they might very well prevent end hosts from negotiating and using TCP options that are supported by both end hosts, but not by the middlebox.

A common issue is that firewalls are configured with policies so conservative that they effectively block all peer-to-peer TCP connections. There exist measurements which indicate that UDP traffic is more likely than TCP to pass through firewalls in both directions [35]. Thus, by encapsulating TCP in UDP, we may be able to facilitate peer-to-peer connections even behind overly restrictive firewalls. While it may seem disingenuous to circumvent such a policy, it can also be argued that such policies often are unwarranted and restrict perfectly valid and safe use cases, for example a file transfer while connected to a hotel access network. When combined with the ability to multiplex STUN messages over the same UDP association, even NATs can be traversed in this manner. This is discussed further in section 3.2.2.

2.3.3 Performance Enhancing Proxies

[RFC3135] defines a Performance Enhancing Proxy (PEP) as "an entity in the network acting on behalf of an end system or user (with or without the knowledge of the end system or user) in order to enhance protocol performance." In the TCP context, these middleboxes compensate for troublesome characteristics of the underlying network by manipulating TCP connections. Such characteristics would be factors that interact badly with TCP's congestion control, such as very high RTT (satellite links), relatively high, fluctuating loss rates or jittery RTT (both common on wireless links).

There are several classes of TCP PEPs, some of which will actively terminate connections in a man-in-the-middle manner, while others just rewrite or suppress segments without actually terminating the end-to-end connection.

TCP Splitting (also known as TCP spoofing) [6] is a kind of active PEP, generally used to segment a network path consisting of some high-RTT sub-path, typically a satellite link, and a sub-network exhibiting shorter RTT, like a regular terrestrial wired network. The PEP terminates the end-system's TCP connection and opens a separate connection between itself and the other end-host, allowing separate congestion control instances to run on the two sub-paths. However, this behaviour is not what the end-hosts expect, nor is it known to them, and there can be repercussions to breaking the end-to-end principle, e.g. if the PEP implementation acknowledges data to a sender before receiving the corresponding ACK from the actual receiver. Application layer proxies, such as HTTP reverse proxies located near the web server [10] which are often used to enhance the performance of high-traffic web sites, implicitly act like TCP splitters.

An example of a non-terminating PEP is an ACK filter [8], which rate-limits ACKs before they hit a low-bandwidth return path by dropping redundant ACKs from the forwarding queue. ACK congestion can be a big problem over highly asymmetric connection where the return path has a lower capacity than the forward path, although one should be careful not to drop so many ACKs that the sender's ACK clock is disturbed.

Sometimes actual PEP implementations combine both terminating and non-terminating mechanisms.

Existing TCP PEPs will not recognize TCP-in-UDP encapsulated segments since these appear to be UDP datagrams, and as such be completely bypassed. One could of course modify them to detect TCP-in-UDP, but care would need to be taken to ensure that the TCP-in-UDP handshake-in-handshake remains coherent across terminating PEPs, cf. section 3.3.2.

2.4 Congestion Control Coupling

As we have discussed in section 2.1.3, TCP congestion control is traditionally applied strictly on a per-connection basis, even in the presence of multiple parallel connections between the same end-hosts. In section 2.4.1 we explain why it would be beneficial to take a more coordinated approach to this, dubbed *congestion control coupling* (CCC). We relate the lessons learned from previous forays into TCP congestion control coupling and contextualize TCP CCC within a wider collection of related congestion control works in section 2.4.2. Section 2.4.3 discusses the requirements for successfully applying congestion control coupling to TCP, before we conclude by identifying shortcomings of existing proposals and how we can address them in section 2.4.4.

2.4.1 Motivation

There would not be much point in adding complexity to an already adequately performing system, which TCP arguably already is, without some upside. We will now discuss the main benefits of coupling TCP congestion control, from a theoretical point of view.

Performance benefits

By coupling the congestion control loops of parallel connections, we expect several performance benefits.

Latency By only letting a single control loop probe the network capacity, which in TCP's case really means gradually increasing the sending rate past the capacity, we expect the queue length at the bottleneck link will be shorter on average compared to the current situation where each connection is probing independently in a more or less synchronized manner. This in turn means the latency will improve, as the delay from waiting in buffers will decrease.

Loss rate Similarly, we also expect to see fewer lost segments, due to the bottleneck buffer overflowing less frequently.

Prioritization Applying a coupled congestion control mechanism also paves the way for introducing new features; when congestion control is performed over an aggregate, the rate distribution function can be adapted in order to give certain flows higher (or lower) precedence compared to the others in the aggregate.

Fairness It becomes possible to achieve perfect fairness among flows, since the rate allocation is tightly controlled among them. Furthermore, by preventing flows from taking different paths and thus experiencing heterogeneous RTTs, unfairness due to differences in RTT is eliminated.

2.4.2 Related Works

Herein we enumerate the most prominent earlier works directly relating to TCP congestion control coupling. An overview of how the different proposals share congestion state information among connections is shown in table 2.1.

Reference	Information usage
RFC2140, "temporal sharing"	Flow 1 has *ended*. Flow 2 joins, and uses the information only initially
RFC2140, "ensemble sharing"	Flow 1 is *ongoing*. Flow 2 joins, and uses the information *only initially*.
E-TCP, EFCM, CM, TiU	Flow 1 is ongoing. Flow 2 joins, and both flows share the information for their lifetime

Table 2.1: Congestion state sharing

After discussing the relevant coupled congestion control works, we also briefly present some multi-streaming protocols, a related class of congestion control mechanisms.

TCB interdependence

[RFC2140] advocates sharing state in an interdependent fashion among "similar concurrent connections" and "similar connection instances"; referring to concurrently *ongoing* connections and *subsequent* connections, respectively. While not definitely specifying what constitutes a "similar" connection, it is suggested that one might apply this to connections between the same two end hosts, or perhaps even between hosts in the same sub-networks.

The authors classify those parts of TCP state, as defined by the contents of the TCP Control Block (TCB), that they deem to be characteristics of the association between a certain host pair, rather than of the individual connections, and thus are candidates for being shared, into two groups:

- Clearly host-pair dependent state
  - Maximum Segment Size
  - Round Trip Time and its variance
- Host-pair dependent state in aggregate
  - Congestion window size (`cwnd`)
  - Slow-start threshold (`ssthresh`)
  - High-water mark (largest observed window size)

It is worth emphasizing that this document advocates sharing state at TCB initialization time *only*.

The authors of [RFC2140] outline two mechanisms for sharing such state: *temporal* sharing and *ensemble* sharing.

Temporal sharing Parameters can be shared by caching them for reuse when opening subsequent connections, e.g. to overcome slow-start, although this could lead to problems if network behavior has changed in the mean time. In this mode, state is normally only updated in the cache upon a connection ending.

Ensemble sharing In ensemble sharing mode, state is cached even as connections are ongoing, and may be used to initialize the TCB of connections starting up concurrently with existing connections to the same destination host.

Adoption Parts of RFC2140 are currently or have been implemented by mainstream operating systems such as FreeBSD (the *TCP host cache* mechanism) and Linux. However, this information is only used in a limited manner in recent versions. Inspection of the source code confirms that, at the time of writing, FreeBSD definitely uses cached MTU/MSS, RTT and RTT variance values. In addition it caches and uses `ssthresh`, although this is not explicitly recommended for sharing in [RFC2140].

Ensemble-TCP

Eggert et al. built on [RFC2140] and proposed the Ensemble-TCP (E-TCP) [30] extension to the TCP stack. E-TCP strives to coordinate bundles of concurrent connections so that their aggregate behaviour is akin to that of a single TCP "Reno" connection. In difference to [RFC2140], in E-TCP state is shared in real-time between flows, not only on connection start-up. The authors make essentially the same recommendations about grouping choices as those in [RFC2140].

Shared state E-TCP's state sharing mechanism consists of replacing some per-connection state in TCBs with pointers to variables located in per-ensemble (connection group) Ensemble Control Blocks (ECBs). ECB's contain the common variables identified by [RFC2140] (see above), as well as state specific to the E-TCP ensemble sharing algorithm:

- List of member TCBs (connections)
- Chronologically sorted list of unacknowledged segments transmitted by member connections
- Rate-based pacing flag

Operation As previously mentioned, some of this state only makes sense to share in aggregate. The strategy chosen in E-TCP to address this is twofold: 1) a scheduler is introduced, which determines which flow should be allowed to transmit when, and 2) the ensembles as a whole runs on a common ACK clock. ACKs and loss signals then have their usual effect on the congestion window and `ssthresh`, but how flows may consume the window is moderated by the scheduler.

The E-TCP scheduler supports giving different weights to each member of an ensemble, allowing some flows to receive precedence over others. In the paper [30], several weighting strategies are outlined, tailored to different use cases.

When new connections join an ensemble, the transmission of the first packets is paced to avoid sending sudden, large bursts into network. Such bursts would likely induce congestion, since the congestion window represents the amount of data it is safe to transmit *over the whole course of a round-trip time*.

Appropriate detection of duplicate ACKs is handled using the previously mentioned list of outstanding segments. Each segment is associated with a counter that is increased each time a preceding segment is acknowledged. The sum of increments to each connection's segments is then added to that connection's duplicate ACK counter, ensuring proper operation of the fast recovery mechanism.

Evaluation Evaluations by simulation related in [30] demonstrate the potential benefits of coupling together the congestion control of connection ensembles, particularly with regard to Web use cases. Web pages usually consist of a number of objects which can be requested in parallel to lower the page load time, which is the essential performance metric in this application. Allowing previous and parallel connections to open the congestion window for such parallel requests is then clearly beneficial, especially given the fact that the file sizes are most commonly small.

To the best of our knowledge, E-TCP has not been implemented outside of the simulator.

EFCM

Another solution was presented by Savorić et al. in [75]: Ensemble Flow Congestion Management (EFCM). They argue that it is appropriate for a connection bundle consisting of *n* flows to be as aggressive as *n* independent TCP "Reno" connections, seeing as how such aggressive behaviour would be the alternative to performing congestion control coupling.

Under EFCM, the shared TCP state is the same as in E-TCP and [RFC2140]. As described in [75], EFCM does not support temporal sharing, state may only be shared among concurrently ongoing connections.

Operation The congestion control state sharing algorithm of EFCM is slightly simpler than that of E-TCP. Each connection runs its own ACK clock and maintains the congestion window as normal. When the window is updated, it is aggregated by summation and redistributed according to fair share between all coupled flows. EFCM does not incorporate any prioritization system.

EFCM incorporates a pacing mechanism to reduce bursting behaviour. It works by limiting each connection to sending at most two segments per burst, and spacing out such bursts according to the formula:

$$\Delta_t = \alpha * sRTT_{agg} / cwnd_{agg}$$

where Δ_t is the time between bursts, α is a pacing factor and $sRTT_{agg}$, $cwnd_{agg}$ are the ensemble's aggregated smoothed RTT estimate and congestion window, respectively.

Evaluation A simulation-based evaluation of EFCM is outlined in [75]. The authors show performance improvements compared to standard TCP "Reno" both in scenarios involving a reliable last hop (e.g. wire-line networks) and an unreliable last hop (e.g. wireless networks), with particularly positive results in the latter case.

We have not been able to find any non-simulator implementations of EFCM.

Congestion Manager

The Congestion Manager (CM) [9, RFC3124] was an extensive proposal by Balakrishnan et al. which went farther than merely extending TCP congestion control. Indeed, the CM is a suite of protocols and APIs intended to revamp Internet congestion control as a whole and throughout the network stack.

Architecture The CM consists of several components:

- Algorithms and protocol
  - Additive Increase, Multiplicative Decrease window-based congestion control algorithm with traffic shaping capabilities
  - Receiver-feedback protocol
  - State storage with information aging support
  - Flow scheduler
  - Prober that can perform active network measurements
- CM adaption API that relays congestion control information between the CM and the application and transport layers.

These components replace some portions of existing protocols, such as TCP. In [9], the authors outline the design of TCP-over-CM (TCP/CM), wherein TCP/CM no longer keeps track of the congestion window, but still does flow control. Flow control information, e.g. loss information and RTT estimates, are fed into the CM, which maintains an aggregate congestion window. The CM flow scheduler then doles the window out to any takers, such as TCP/CM, which entails an asynchronous notification of an opportunity to send. TCP/CM is comparable to E-TCP, in that a bundle of parallel TCP/CM flows is coordinated to behave as a single TCP "New Reno" flow.

Generalized congestion control One of the goals of the CM is to be able to manage congestion in a holistic way even across different transport protocols. In order to achieve this, and due to the fact that many unreliable protocols do not signal back much, if any, feedback from the receiver, the CM can: 1) receive explicit feedback from its client transport or application protocols, 2) actively measure congestion using probe messages or 3) passively monitor for congestion by inserting a CM protocol header between the IP and transport layer headers. Options 2) and 3) require the CM framework to be running on the receiver side as well, whilst option 1) can function in a sender-only manner (e.g. TCP). The CM-based feedback system also requires additional CM protocol handshaking to set up.

Evaluation A simulation evaluation presented in [9] validates the ability of the CM to fairly schedule flows.

Adoption Although the CM has been both standardised by the IETF with proposed standard status in [RFC3124], and implemented within the Linux kernel network stack [4] (although not merged upstream), it has not seen any widespread use.

We submit that the reasons for this probably encompass the sheer complexity of the system and the fact that it requires large changes to the network stack. Indeed, the CM entails changes both at the network and transport layers, and introduces a transport protocol number of its own. Experience with the deployment of IPv6 and novel transport protocols such as SCTP or DCCP have exposed an overwhelming inertia working against larger changes to the network protocols. Even a relatively simple IPv4 extension like ECN is only just becoming generally available for use, over a decade and a half after being conceived. Furthermore, the threats to Internet stability the proponents of CM warned of have not materialized, or at least not resulted in destabilization of the network. TCP is still the dominant protocol in the Internet, and despite the congestion control landscape having become more diverse over the last decades, there are arguably no signs of a new congestion collapse looming on the horizon.

Multi-streaming transport protocols

A number of proposed and standardised transport protocols include some form of multi-streaming support. By multi-streaming we mean that these protocols permit the embedding of several more or less independent subflows of messages within a single connection or association between the same two end hosts. Many of these also perform some kind of congestion control coupling. We now briefly present some of the most prominent such multi-streaming protocols and relate them to our project.

SCTP The Stream Control Transmission Protocol (SCTP) [RFC4960] supports this kind of scenario right out of the box because it supports multi-streaming using only one congestion control instance for all streams.

However, SCTP traffic may not actually pass through all middleboxes because it is a relatively new, at least in Internet terms.

There has been some work [14, 65, 83] toward enabling SCTP adoption without having to modify application code through the use of a TCP-to-SCTP shim layer. The authors of [83] found that there were tangible benefits to doing this in several cases, even when accounting for the overhead incurred by their user-space shim layer's connection management. Since the SCTP network transparency problem remains and is likely to persist for some time still, we find these results encouraging with respect to the mid-term usefulness of our own UDP-based solution, which does not suffer from the former.

HTTP/2 (SPDY) Hypertext Transfer Protocol version 2 (HTTP/2) [RFC7540] is a recent revision of the HTTP protocol, which powers the World Wide Web. It evolved from Google's experimental SPDY protocol [12, 76]. While still running on top of TCP, HTTP/2 specifies multiplexing several HTTP virtual connections, streams, over a TCP single connection. This alleviates HTTP application layer HOL blocking issues; earlier revisions of the protocol required that requests be responded to strictly in order of arrival, but now requests can be made in separate streams and thus served as fast as possible. Obviously, transport layer TCP HOL blocking remains a concern.

QUIC QUIC [17, 38] is Google's proposed solution to the above mentioned HOL issues in HTTP/2, in addition to addressing issues related to connection setup latency, congestion control and endpoint identification. By transporting HTTP/2 multiplexed streams over UDP instead of TCP, delivery of messages from different streams become truly independent of one another, eliminating inter-stream HOL blocking.

QUIC brings several other advantages for common Web use cases: In the (common) case of short Web flows, connection setup latency often makes up a substantial portion of the total request completion time, thus doing away with the TCP three-way handshake can yield improved page load times. Since UDP does not provide any congestion control, QUIC provides an opportunity to define CC mechanisms that are not restricted over compliance with TCP CC standards. Finally, QUIC specifies an application layer endpoint identification mechanism for catering to mobile roaming use cases, a space in which existing standard efforts like Mobile IP [RFC5944, 70] have in practice all but stranded.

In some ways, QUIC is similar to our approach, in that it proposes to multiplex what would previously (in HTTP/1.1) have been transported over separate, but parallel TCP connections, over a single UDP connection. However it goes further, completely doing away with TCP and is aimed squarely at serving Web use cases and their requirements, while our proposal is more general and retains almost all standard TCP semantics.

RTMFP The Secure Real-Time Media Flow Protocol (RTMFP) [RFC7016] is a protocol developed by Adobe Systems primarily for the transport of real-time media. It multiplexes several logical flows within a single holistically congestion controlled UDP connection. The protocol is message-oriented and supports any configuration of full, partial or no reliability combined with both ordered and unordered delivery. NAT traversal and rendezvous-point assisted peer-to-peer connection setup is an integral part of the protocol, along with mobile roaming capabilities.

Probably owing to its origin as a proprietary and closed protocol [9], there are few if any published independent evaluations of this protocol. However, the transport services described by [RFC7016] do cover what we are aiming to achieve, namely coupled congestion control via flow multiplexing, and the protocol is designed to be able to transport any application layer stream.

Minion The Minion suite [67], is a proposal that aims to provide new transport services delivered in a form that looks and superficially behaves like commonly accepted and deployed legacy transport protocols (TCP, UDP and SSL). By ensuring the wire behaviour is not easily distinguishable from how those protocols usually work, it becomes possible to pass through middleboxes in the network, and even benefit from them in the case of PEPs. By using message boundary encodings and with some modifications to the end point TCP implementations, the TCP (and SSL) minions are able to provide unordered datagram delivery on top of the TCP ordered byte stream, easing the HOL problem. However, only the UDP minion, which simply encapsulates newer transports as payload while mirroring the port numbers of the encapsulated datagram, is able to provide unreliable or partially reliable unordered delivery; the TCP and by extension SSL minions will still need to retransmit any datagrams that are lost, potentially hurting performance.

The main emphasis of Minion seems to be on providing (some) newer transport services along with increased ability to make it through the network by mimicking the very most common protocols currently deployed in the Internet.

MultiPath TCP MultiPath TCP (MPTCP) [RFC6824, 84] is a protocol which orchestrates a number of TCP connections in concert as sub-flows in a MPTCP super-connection. Congestion control is coupled across these sub-flows, although the base assumption is that they forward segments along *different* paths through the network, usually by taking advantage of multi-homing, i.e. connecting hosts to several distinct access networks. MPTCP has also been shown [72] to be able to provide benefits even between single-homed hosts, provided that network contains some multi-path mechanism such as ECMP that splits the paths taken by sub-flows within the network itself.

[9]While Adobe Systems later published an open protocol description in [RFC7016], this document does not fully specify the necessary authentication mechanisms to inter-operate with their own implementation in the Adobe Flash Player and Adobe AIR products. At the time of writing, there is at least one actively maintained open source implementation: https://github.com/MonaSolutions/MonaServer.

The coupled congestion control mechanism in MPTCP is different from what we are aiming to create, however, in that it coordinates the transmission of data onto several TCP sub-flows that in turn apply their own end-to-end congestion control. One could consider sender-side MPTCP as an inverse multiplexing mechanism [27]. [10]

In contrast, our goal is to join together a number of related, but separate TCP connections and apply CC over the aggregate as if it was a single flow. Our CCC mechanism can be considered the inverse of MPTCP's, i.e. a sender-side multiplexer.

Substrate Protocol for User Datagrams (SPUD)

The Substrate Protocol for User Datagrams (SPUD) [28, 78] is a proposed UDP encapsulation protocol with built-in end-host to middlebox signaling and cooperation. The concept is that by explicitly tagging sub-flows while informing middleboxes of its properties and intended behaviour, they will both be able and amenable to giving those flows the treatment of their choice with respect to priority, ECN support and so on. Furthermore, the actual payload is hidden using cryptography out of privacy concerns.

SPUD does not explicitly deal with congestion control, but it does multiplex several flows over a single UDP port pair, which would usually prevent path splitting by ECMP, although one could imagine that the flow itself could export a preference in this respect explicitly, too.

It would be possible to imagine a combination of our scheme with SPUD.

2.4.3 Requirements

Based on our analysis of the previous works, we submit that successfully applying CCC to TCP ultimately has requirements that extend beyond a good algorithm for amalgamating the congestion control state of related connections.

Single-path delivery

In order to make TCP CCC useful, it is imperative to somehow ensure that all the parallel connections use the same network path. One way of doing this is to bundle together the connections so that they all look like a single flow to an external observer along the route. In other words, ensuring that they will receive the same per-flow treatment at routers implementing multi-path routing such as ECMP or LAG as described in section 2.2.

That said, there are actually means of marrying multi-path routing with CCC techniques in order to achieve better load balancing and performance. In [72] the authors combine ECMP with MPTCP in a data center "fat tree" topology, yielding far better performance even for communication between single-homed hosts. Without ECMP load balancing within the network,

[10]It is not a demultiplexer, since the sub-flows remain tightly interrelated.

this would not be achievable. Of course, this is thanks to the fact that MPTCP's CCC is designed around handling multi-path networks.

Pacing

TCP's bursty sending behaviour is a well known consequence of ACK clocking and how the congestion window is updated. This is an undesirable dynamic in any case, and especially in high Bandwidth-Delay Product (BDP) situations. Bursts can lead to rapid queue growth and thus higher latency, not to mention inducing packet losses and introducing oscillations in the congestion controller resulting in lower link utilization than optimal.

In coupling congestion control across connections, one risks exacerbating this problem, since connections might gain `cwnd` in large chunks in a manner that would not occur in regular TCP congestion control. Additionally, assuming each connection maintains its own independent ACK clock, one might experience odd effects due to desynchronization.

In general, a number of approaches for moderating TCP's burstiness have been suggested. On the sender path, [RFC3742] e.g. limits congestion window growth in Slow-Start in high-bandwidth environments, while proposals such as [7] modulate the feedback stream of ACKs to induce pacing from within the network.

An alternative is to simply delay packets at the sender to spread them out more evenly in the time domain, i.e. (rate-based) *pacing*. In relation to coupled congestion control, EFCM [75] implements such a pacer, whilst both E-TCP and the CM introduce full-blown schedulers.

In our case, we are unfortunate as the FreeBSD TCP stack does not support pacing at this time, and it is outside the scope of this thesis to attempt to implement it. [11]

Deployability

Practical deployability concerns must be taken into consideration. If a proposed solution is not incrementally deployable, it seems extremely unlikely that it might be adopted in the public Internet today. Preferably it should be possible to deploy changes on the sender-side only, although that may prove hard to reconcile with the single-path requirement stated above. In any case, sweeping changes to system architecture appear to hinder the uptake of otherwise well founded solutions; if applications can benefit from changes without being modified, this is clearly an advantage.

2.4.4 Weaknesses in previous proposals

Both E-TCP and EFCM break the TCP loss event model described in section 2.1.3, but in different ways. E-TCP lets all flows react to their

[11] TCP pacing was experimentally implemented for FreeBSD by a team of researchers at BBN Technologies, but was never adopted by the main project. See http://web.archive.org/web/20090907003159/http://www.ir.bbn.com/projects/pace/.

individual losses independently, which leads to behaviour that is too conservative, e.g. when a flow experiences a timeout, this forces the aggregate into slow-start even if other flows are still receiving ACKs. EFCM, on the other hand, behaves aggressively by only applying loss reactions to the individual congestion windows; in a member of a ten flow ensemble experiences a loss, it will halve it's own rate and sum it with the other nine flows, leading to a very moderate decrease from $10x$ to $9.5x$ aggregate congestion window size. We attempt to settle on a more appropriate, middle ground behaviour.

The algorithm for sharing the slow-start threshold in EFCM can also lead to issues, as it allows a joining flow to share it's "infinite" initial `ssthresh` value with the ensemble, switching the whole aggregate into slow-start and triggering a severe bursting event due to the fact that the other flows already have running ACK clocks. We will instead let newly joined flows inherit the `ssthresh` of the ensemble, since it is an estimate of a safe level to slow-start to based on actual observations of the network.

Chapter 3

Design: TCP-in-UDP encapsulation

In this chapter, we will present the design of the chief contribution of this thesis, our proposed solution to the multi-path routing issues facing TCP CCC: *TCP-in-UDP encapsulation*.

TCP segments are encapsulated as UDP datagrams, while compressing the TCP header in order to maintain the same Sender Maximum Segment Size (SMSS). All encapsulated segments are multiplexed onto the same UDP port-pair, so they appear as one UDP flow on the wire, i.e. have the same flow-identifying five-tuple. This means it is very likely that they will take the same path through the network, as previously discussed in section 2.2.1.

While our approach does require changes to the network stack at both ends of the connection, it avoids Head of Line (HOL) blocking between flows since each segment is sent and delivered as separate and independent datagrams. By omitting fields that are redundantly present in the UDP header or little used, we are able to preserve the same SMSS as unencapsulated TCP over the same path and avoid unnecessarily repeating computationally expensive checksum calculations.

TCP-in-UDP encapsulation can be made transparent to the application layer, all transformation happens at the transport layer and behaviour is like that of a regular TCP connection from the perspective of the upper network layers. The removal of certain header fields, which is necessary in order to preserve SMSS, does slightly affect what services TCP is able to provide to the application layer.

In addition to ensuring single-path passage through the network, this encapsulation method delivers even more desirable features:

- Inside the tunnel, we make the rules; we gain the freedom to experiment with TCP options without having to fear the interference of middleboxes. Notably, we can also enable options that extend the TCP option space, which has proved quite restrictive as the number of common options has grown.

- With an astute reordering of header fields, we may even multiplex other protocols over the same UDP port-pair. One possible application of this is to enable automatic NAT traversal using ICE and STUN.

The next sections describe TCP-in-UDP in depth.

3.1 Requirements

In designing the encapsulation system, we drafted the following requirements:

Transparent to the application Applications should not need to be modified in any way to be able to take advantage of the encapsulation (and CCC) scheme.

Configurability However, it must be possible to control it both as a system-wide policy, but also per-socket if that is desired.

SMSS conservation Sender Maximum Segment Size (SMSS) must not be decreased by the encapsulation.

Opportunistic use It should be possible to opportunistically detect the availability of the mechanism and enable it on the fly, with a fallback to regular TCP.

No connection delay We do not want to delay the already relatively long TCP connection setup. We must not impose further delays beyond the normal three-way handshake.

Low overhead The mechanism must lend itself to an efficient implementation so as not to hurt performance.

Simplicity The encapsulation system should be implemented with as little disruption of the existing network stack as possible.

ICE multiplexing As evidenced by related works, it is trivial to lay the groundwork for multiplexing of Interactive Connectivity Establishment (ICE) messages over the same UDP port pair.

We will now detail the rationale for these requirements.

3.1.1 Application transparency

Many existing TCP-based applications could benefit from the use of CCC. One common example are Web browsers and servers, which typically need to complete a series of requests in order to load a web page. First, the HTML markup defining the page must be transmitted. Then, other resources such as images, Cascading Style Sheets, Javascript source files and so on, embedded in the page must also be transmitted – these requests can and do in fact gradually happen even before the entire HTML content is received. HyperText Transfer Protocol (HTTP) version 1.1 [RFC7230] compliant browsers usually use a combination of two methods to speed up this process: they make multiple concurrent persistent connections to the server, which allow several requests to be made sequentially over each TCP connection, and one concurrent request per connection opened. [1]

Clearly, it would be an advantage to couple the congestion control of these concurrent connections. With respect to the time needed to complete all requests, essentially the same as the page load time which is important for user experience, it would be very advantageous to share the `cwnd` of the initial HTML-fetching connection with other connections opened to request other objects, bypassing the need for slow-start on those connections.

Therefore, we require the CCC and encapsulation mechanisms to be able to be turned on and used completely transparently to existing TCP applications. In this case, it is the system administrator who decides whether or not to use them. Note that this does not preclude affording TCP-in-UDP aware applications more control.

3.1.2 Configurability

As we have stated above, we want our mechanism to be able to be employed without application-level configuration. However, it is also highly desirable that new applications should be able to be made aware of the mechanism and allowed to configure it. There are also some other means by which this could be achieved on a per-application basis without modifying the application itself. One way would be to use a wrapper-library and linker preloading to override socket calls, in the vein of [65]. We will not pursue this within the scope of the present project.

System administrators also need to be able to set system-wide policies governing the use (or not) of TCP-in-UDP.

3.1.3 Maximum Segment Size conservation

In order to achieve the highest possible performance, we must take care not to reduce the Sender Maximum Segment Size (SMSS) parameter of TCP. It is already small compared to the transmission capacity of modern network links, being constrained by the link-layer Ethernet Maximum Transmission Unit (MTU) of 1500 bytes. Although modern link layer technology often does support higher MTUs, this size is still considered the safest choice for ensuring your packets will pass through the Internet.

While it would be possible to let Path MTU Discovery (PMTUD) or IP fragmentation take care of this problem, these solutions incur great costs. In the case of PMTUD, you are likely to lose a number of packets before adapting to an acceptable MTU (and by extension SMSS), causing unacceptable delays and perhaps interfering with CC. IP fragmentation is well known to hurt TCP performance, e.g. by causing reordering, and adds to the computational load of the end hosts.

[1]HTTP/1.1 does in fact also sport a pipelining feature, which allows the client to make many requests without waiting for them to be serviced first, however popular browsers have historically not turned it on by default.

The solution then is to avoid inflating the size of generated IP datagrams. In turn, this means we must reclaim the additional 8 bytes taken up by the UDP header somewhere in the TCP header.

3.1.4 Opportunistic and Timely Connection Establishment

The delay in setting up a TCP connection is already lamented by some as too leisurely. Owing to the three-way handshake, and to the fact that due to the threat of denial-of-service attack, TCP implementations delay processing, and in fact usually discard, any payload data sent before the connection is confirmed, it usually takes at least a whole RTT before a TCP client can successfully transmit any data. In most client-server model systems, like HTTP, this puts a lower bound of 2 RTTs on the total waiting time for a request to complete, so it is highly desirable to minimize this delay.

Any failures, e.g. from packets dropped by middleboxes, during the handshake will significantly increase the delay; not only are these crucial packets lost, but it usually takes a very long time to react to the loss because the initial RTO value is set very conservatively and many middleboxes, or indeed end hosts that do not understand how to process a packet, will not signal an error in any way. It is then crucial that our connection establishment phase does not rely on anything middleboxes may consider out of the ordinary, or on explicit error signaling by way of ICMP messages.

This is the same dilemma facing those seeking to gradually deploy IPv6. Many networks would drop IPv6 packets silently, and these networks could be anywhere along the path between the sender and receiver. The Happy Eyeballs algorithm [RFC6555] is a widely deployed solution that permits dual-stack IPv4/IPv6 networking, where the system can seamlessly pick either IPv4 or IPv6 on the fly.

Efforts such as QUIC and TCP Fast Open (TFO) [RFC7413, 71] permit zero RTT connection establishment delay in repeat connections to the same host. QUIC can be considered a completely alternative transport protocol solution, and we will not consider it in our design. TFO is a TCP extension, and it may be possible to accommodate our design to work with it, but we will not investigate this within the scope of this thesis. [2]

3.1.5 Minimal Overhead

Any encapsulation obviously requires more work to be done during packet processing, and consequently introduces some incremental delay as well. Both in order to keep latency to a minimum, and to economise on valuable computing resources in the end hosts, we must strive to design our scheme so it lends itself to a highly efficient implementation.

[2]Incidentally, FreeBSD support for TFO was added after we forked off from the main source tree. Early attempts to keep up with upstream changes to the tree proved very costly in terms of time and effort, so we have refrained from merging upstream code into our development tree for the duration of this project.

3.1.6 Implementation Simplicity

As we noted earlier in section 2.4.2, earlier attempts at providing CCC, notably the Congestion Manager [9, RFC3124], likely stranded at least in part due to the sheer complexity of implementing them. The more radical a departure from the established functioning of the network stack, the less likely a solution is to be adopted. The likelihood of introducing new problems also increases with complexity, and it is easier to make mistakes during implementation.

We strive to keep our solution as simple as we can. It should interfere minimally with the way the existing network stack works, and we try to preserve well defined interfaces where we can. This makes for a solution that is relatively simple to implement correctly, and which can be modularized easily.

3.1.7 ICE multiplexing

NATs are ubiquitous, and as we discussed in section 2.3.1, often interfere heavily with attempts to establish peer-to-peer connections. In response, the Internet community has developed sophisticated machinery for working around this problem, chief among them is ICE (see section 2.3.1), which will often succeed at piercing through NAT when combined with the STUN protocol.

We borrow the ingenious mechanism for a combined STUN and TCP encapsulation put forward by Denis-Courmont in [25] and Cheshire et al. in [19]. It is based on the observation that the TCP offset field has a minimum value of 5, whereas STUN messages all start with the two most significant bits of the first octet set to zero. Therefore, if we place the TCP offset in first position, we can know that any datagram with a value lower than 5 encoded in the first nibble cannot be a TCP segment, and can be handled differently by the tunnel endpoint.

Note that we will not attempt to implement any STUN handling in the context of this project, as that is most likely a relatively large undertaking in of itself.

3.2 Header format

To avoid issues related to the Maximum Transmission Unit size when encapsulating TCP segments in UDP datagrams, entire segments will not be placed as payloads in the datagrams as is. Instead, the TCP header, depicted in figure 3.1, is rewritten, removing redundant and, infrequently used fields. This frees up space that can be used to encode enough information to allow a certain number of concurrent connections to be demultiplexed.

3.2.1 Modified fields

The checksum field is entirely redundant, since UDP provides an equivalent facility. We further elide the infrequently used Urgent Pointer field, along with the associated urgent flag. Although this will break compatibility with the, admittedly few, applications that make use of this feature, it is a reasonable trade-off. As of 2011, the Internet Engineering Task Force (IETF) has reaffirmed that TCP implementations must support the urgent pointer feature, although strongly discouraging (deprecating) its use [RFC6093].

Earlier efforts to design such a header format, as described in [19, 25], also remove the source and destination port pairs and rely only on the port fields in the UDP header. Unfortunately, the TCP-in-UDP design can not replicate this since, unlike the aforementioned authors, the primary goal is to be able to multiplex several TCP connections over the same outside-observable transport flow, as defined by the (source address, destination address, source port, destination port, IP protocol number) five-tuple (see section 2.2.1). Instead, we compress the full source and destination port fields into a narrower flow identification field at the cost of additional complexity – the introduction of flow identifier to pair-pair negotiation, as well as another demultiplexing step on reception – and fewer possible parallel (encapsulated) streams.

This header format satisfies the requirement of SMSS conservation. It is also conducive to an implementation incurring low overhead.

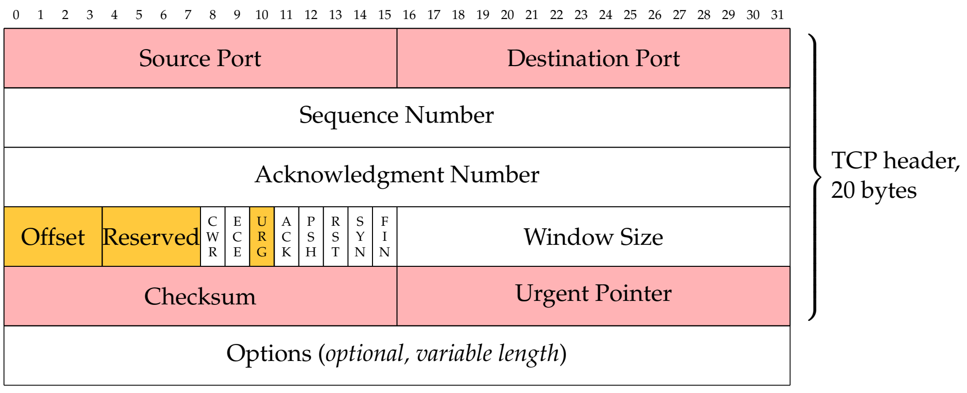


Figure 3.1: Standard TCP header. Fields on red background are removed by TCP-in-UDP, those on orange background have their semantics modified.

3.2.2 Field reordering

As explained in section 3.1.7, we reorder the fields in the header so the offset field is placed at offset zero. The most sensible way to accomplish this with respect to alignment is to move all the other fields within the same 32-bit word along with it.

This reordering satisfies the requirement of ICE multiplexing.

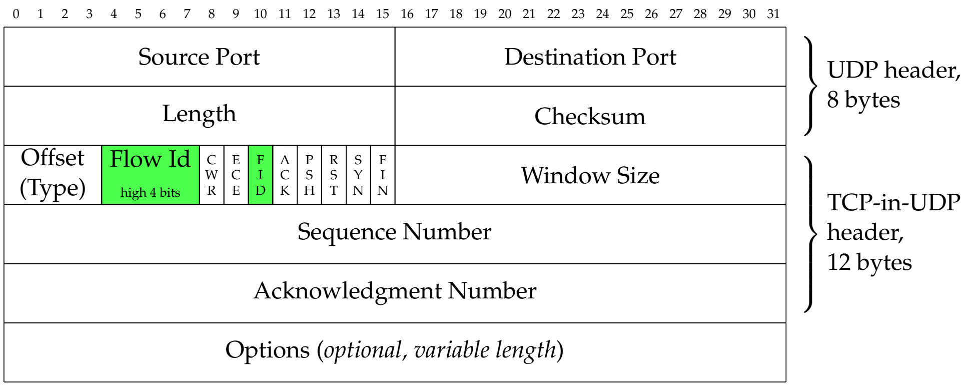


Figure 3.2: Compressed TCP-in-UDP header. The Flow Id split-field is highlighted in green. Notice that the port numbers in the UDP header are those of the tunnel, *not* the TCP connection.

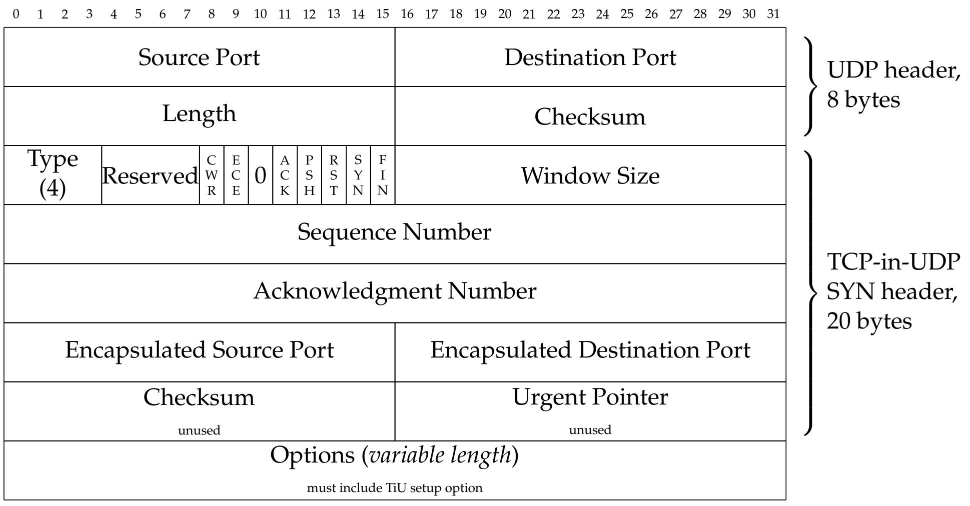


Figure 3.3: TCP-in-UDP SYN/SYN-ACK header.

3.3 Protocol

The TCP-in-UDP encapsulation needs a mechanism for establishing the necessary state to correctly encapsulate connections. We will now specify this protocol.

3.3.1 Setup negotiation

Since we are going to encode the application endpoints represented by the source and destination port-pair in fewer bits, we must provide a means for the two hosts to negotiate a mapping between the compressed code-points and the full port pair as understood by TCP.

Wire format

We take advantage of the fact that we need not take into account middlebox support when constructing tunneled segments, and convey this

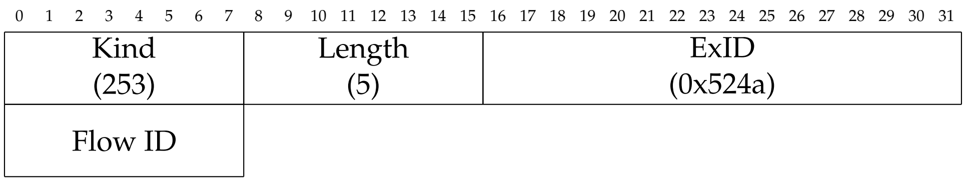


Figure 3.4: TCP-in-UDP setup option

information using a combination of a full-length header, see figure 3.3, with our own experimental TCP-in-UDP setup option, as depicted in figure 3.4. This special header format is only used when encapsulating SYN and SYN-ACK segments, i.e. the first two packets in the TCP three-way-handshake.

We leave the original port pair unchanged, but reordered – swapped with the offset, flags and window fields – and encode a five-bit flow identifier into the setup option. The setup option follows the TCP experimental option format defined by [RFC6994]. The experimental identifier (ExId) has been arbitrarily chosen and checked for collisions against the relevant Internet Assigned Numbers Authority (IANA) registry. [3] The range of valid flow identifiers is 0-31. A value of 255 in this field is used to indicate an error condition, such as a collision with an already in-use flow identifier.

The offset field is always set to the special value 4, which makes it easy to decapsulate these messages differently from regular segments. [4]

Using this format, we cannot preserve MTU, however, in this special case we can live with that. Although the TCP standard allows payload data to be included with these segments, modern TCP implementations do not do so due to the danger of denial-of-service attacks. In so-called "SYN flood" attacks, an attacker overwhelms a listening TCP server with fake SYN segments, usually having spoofing the source address, meaning the server will never get any response to its SYN-ACK and be forced to wait until a timeout expires before being able to drop the state associated with the (half-open) connection. If an attacker is able to generate these fake SYNs at a high enough rate, the server will run out of resources and be unable to do any useful work. If TCP were to accept and buffer data appended to the SYN, which it would have to do since the connection has not been synchronized yet, it would make the attacker's job of starving server resources that much easier.

As a response to this kind of attack, FreeBSD does not even instantiate a full TCP Control Block upon receiving a SYN; rather the minimally required state to be able to recognize when a connection is confirmed (by the third, ACK segment from the initiator) is stored in an overwrite-

[3]We have not at the time of writing made a request to register this ExId with IANA, although that may change pursuant to ongoing IETF discussion of this work, see [82].

[4]This choice really stems from an earlier iteration of the design, which allowed for such negotiations to happen outside of the three-way-handshake. That proved not to be very robust, so we have restricted it. The implementation still relies on this fact, so we have left the old format intact in this description. Going forward, these setup headers could instead be identified by checking the state of the SYN flag, which is always in the same, fixed position.

when-full SYN cache. Any data present in the SYN segment is simply not acknowledged, a practice allowed by the semantics of the TCP standard.

Aside from these facts, we are in a position to specify such a rule in any case, seeing as how we have full specification control of the behaviour of a tunneled connection.

A peculiarity of this format is that we are leaving the checksum and urgent pointer fields intact. This is purely done in the interest of minimizing copying operations, which would be necessary in order to remove them, which is unnecessary since we have no need to save space when using this particular format. The fields are not filled, but the main TCP segment construction code reserves space for them.

Negotiation protocol

The protocol used to negotiate mappings is very simple:

- On connection initialization, the initiator reserves a free flow identifier and associates it with the connection's port-pair.
- When constructing the SYN segment, the TiU setup option is appended with the previously reserved flow id.
- When the listening TCP receives that segment, it checks if it already has an active mapping for that flow id.
  - If no, it creates one and responds with a SYN-ACK segment appended with the TiU setup option, echoing the flow id. back to the initiator.
  - If there is a collision, the connection is marked as not encapsulated, but the SYN-ACK is still sent encapsulated in order to signal back the error. This is done by appending the TiU setup option and setting the flow id. field to 255.
- Based on the response from the listening side, the initiator either confirms the encapsulated connection or falls back to regular, unencapsulated TCP, freeing the previously reserved flow id.

Figures 3.5a and 3.5b on page 44 illustrate the messages exchanged in a successful and failed negotiation, respectively.

Port-pair to flow identifier mappings are keyed against the source and destination address pair, i.e. there is a unique map for each host (interface) pairing. We do not specify any mechanism to renegotiate another flow id. if the first one does not work, since that would necessitate a longer handshake and introduce a connection setup delay. Collisions are very unlikely, and should only practically happen for a short while following some kind of state reset at the initiator side, e.g. a reboot.

In case the initiator is unable to reserve any flow id. because the map is full, it will not attempt setup negotiation and directly fall back to regular TCP.

By providing this mechanism, we satisfy requirements for application transparency and zero additional connection delay.

3.3.2 Happy eyeballs protocol

As we have set a requirement that applications should not have to deal with controlling the scheme, we need to provide automatic activation logic. At the same time, we satisfy the requirements for configurability and opportunistic use.

Our solution is an implementation of the Happy Eyeballs scheme [RFC6555] for automatic IPv4/IPv6 selection that we have adapted to fit our needs.

On connection setup, we run the following protocol:

1. Check if we have cached any TiU capability information for the destination host.
    - If yes, we can shortcut the decision – if we know the host supports TiU, we use it without further probing. TiU state is set to **wanted** $\wedge$ **offering**.
    - If TiU probing has failed with some kind of error earlier, we will not attempt probing and fall back to TCP. TiU state is set to **disabled**.
2. If we could not make a decision based on cached information, we send both an encapsulated and a regular SYN segment in rapid sequence. TiU state is set to **probing** $\wedge$ **offering**.
3. The listener's response depends on which arrives and is processed first:
    - If TiU arrives first, proceed with the flow id. negotiation. Unless that fails, we will go ahead and use encapsulation on this connection. TiU state is set to **offered**. The TCP SYN will be silently ignored when it arrives.
    - In the converse case, if the TCP SYN arrives first, the connection proceeds as normal over regular TCP. When the TiU SYN arrives, however, we will respond to it, but indicate an error. The corresponding host is cached as being TiU capable at the receiver side.
4. The initiator receives either an encapsulated or regular SYN-ACK segment back:
    - If we got a TiU SYN-ACK that echoes the chosen flow id., the encapsulated connection is confirmed. Encapsulation will be used for the rest of the connection lifetime. TiU state is set to **enabled**. The correspondent's TiU capability is cached.
    - If we got a TiU SYN-ACK that indicates an error, the connection falls back to regular TCP. TiU state is set to **disabled**, but the correspondent's TiU capability is also cached since this response indicates it can perform TiU.

- If we got a regular TCP SYN-ACK, the connection falls back to regular TCP. Nothing is cached. TiU state is set to **disabled**.
- Should the initiator get back an ICMP connection refused error, we know there is no TiU tunnel endpoint active on the server. Fall back to TCP and cache that the correspondent is unable to handle TiU. TiU state is set to **disabled**.

5. Depending on the outcome of the previous step, the initiator completes the handshake by transmitting either a TiU or TCP ACK.

6. On receipt of the ACK, the server determines if a connection in the TiU **offered** state should continue to be encapsulated in TiU or not. If it is a TiU ACK, state changes to **enabled**, otherwise it changes to **disabled** and the connection falls back to TCP.

7. For the *entire* of the lifetime of the connection, segments will be encapsulated or not based on the TiU state being either **active = enabled** $\vee$ **offering** $\vee$ **offered**, or **disabled**. This is valid on both the initiator and listener side of a connection.

Step 6 of this handshake-within-the-handshake is required to avoid problems in situations where the server has seen the TiU SYN and starts using TiU encapsulation, but the (TiU) SYN-ACK is lost in transit. If this happens in just the wrong circumstances, the initiator will reach a limit for the number of TiU SYN segments it will attempt and definitively fall back to TCP. Due to the simplified test (i.e. step 7) used to determine whether a segment should be encapsulated or not, the server would then continue encapsulating even though the initiator had fallen back, which also implicates that it will have released the state allowing it to correctly decapsulate segments on this connection.

The (mostly) layered approach outline here simplifies integration of our encapsulation implementation into the existing network stack, which is one of the requirements we have set.

Figures 3.6a and 3.6b on page 45 show the message sequences in case the TiU SYN or TCP SYN arrives first, respectively.

3.4 Interoptability

In this section we will briefly discuss how TCP-in-UDP affects interoptability with other TCP implementations and standard network mechanisms.

3.4.1 Urgent Data

As previously discussed in section 3.2.1, since both the urgent pointer field and the URG flag are removed or re-purposed by TCP-in-UDP, this feature of standard TCP is not usable when enabling the encapsulation. In practice, this is not very problematic because very few applications still in widespread use actually rely on this feature. The IETF has furthermore deprecated the entire feature for new applications in [RFC6093], noting that almost all current TCP implementations implement it in inconsistent and standard-incompliant manners. `telnet` is one well-known example of an existing application which depends on this functionality.

We outline a workaround for this issue in section 3.2.1.

3.4.2 Explicit Congestion Notification

TCP-in-UDP should not conflict with ECN per se, but routers will obviously see UDP datagrams. As McQuistin et al. [62] have shown, about 98.97% of the servers they surveyed were reachable by UDP datagrams marked as being ECN-capable, but it is uncertain whether ECN marking will actually be applied by AQMs in favor of just dropping the packets.

It would be possible to combine TCP-in-UDP with a protocol such as SPUD [78] to explicitly inform routers and middleboxes that the TiU UDP flow really is ECN-aware.

3.4.3 IPv6

TCP-in-UDP does not embed any addresses within the protocol and is as such not dependent in any way on the IP protocol version used.

IPv6 support has not been in the scope of this work implementation-wise, however.

3.4.4 Stateful middleboxes

In the absence of explicit state-defining messages, NATs and stateful firewalls tend to maintain soft state using timeouts for UDP flows. Idle timeouts are typically short, on the order of 2 minutes, compared to hours for TCP [66]. Thus one risks that a TiU encapsulated connection could get interrupted by such a middlebox flushing it's state if the connection goes silent for a while.

Likewise, these middleboxes are likely to interfere with the happy eyeballs fallback mechanism. If they don't observe a TCP SYN coming from inside their network, they will not prepare any TCP state for passing in segments on the return path. It is not unlikely that fallback will fail, even if an unencapsulated ACK segment goes through the middlebox (or at least attempts to) from inside the network in that event. One possible solution to this problem would be to *always* transmit duplicate TCP segments, even if TiU is expected to work based on cached information. [5]

3.4.5 Performance Enhancing Proxies

As we discussed in section 2.3.3, Performance Enhancing Proxies (PEPs) will not know how to deal with TiU encapsulated traffic. This might be a disadvantage under some circumstances, since PEPs are usually deployed

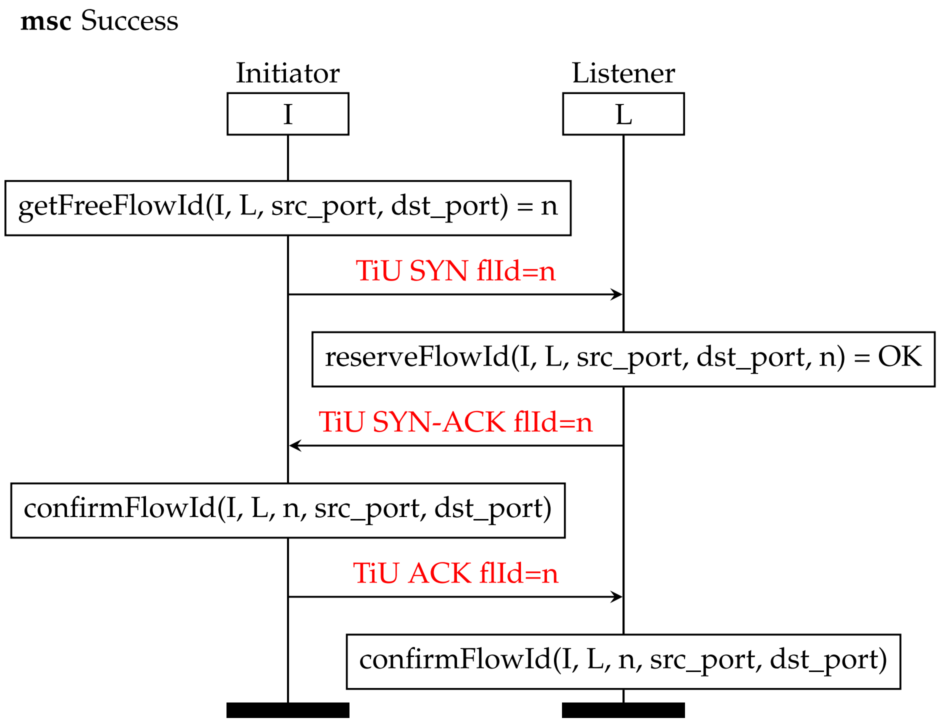
[5]In the IETF draft concerning the present work, we do make this recommendation. [82]

to alleviate performance issues in environments where TCP struggles, e.g. high-latency satellite links and so on.

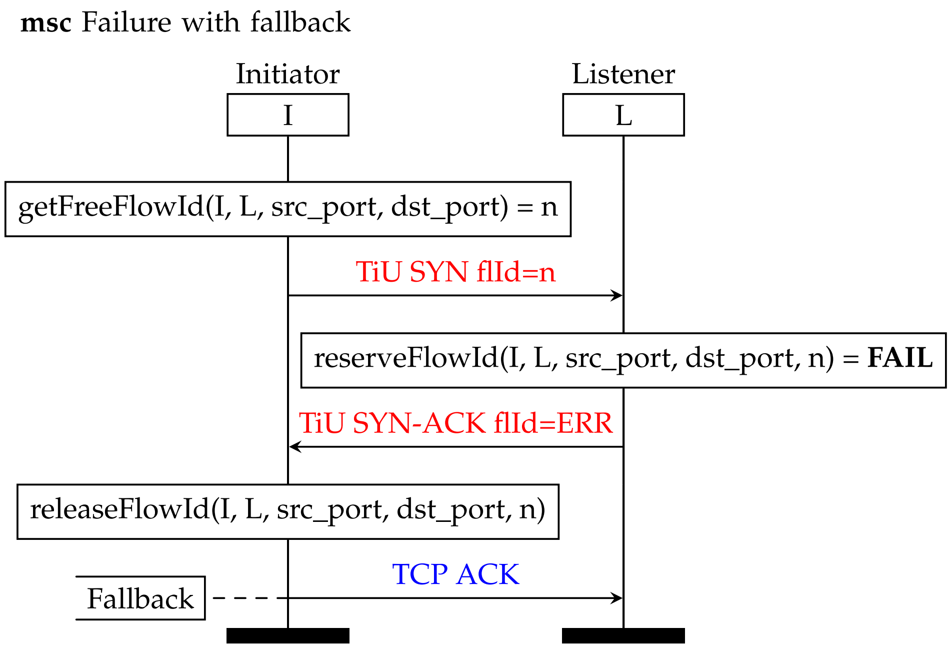
3.5 Security

TCP-in-UDP is generally as secure as TCP, since its protocol behaviour truly is that of TCP. The exception is the connection setup phase, during which TCP-in-UDP is somewhat vulnerable to Denial of Service attacks mostly due to the narrow flow-Id field. An attacker could easily flood a TCP-in-UDP aware receiver with bogus TCP-in-UDP encapsulated segments on all possible flow-Ids, with the range only being 0-31. Another attack would consist of flooding with SYN/ACK packets with the flow-id in the TCP-in-UDP setup option set to an invalid value, normally used to indicate a flow-id collision. We do not, however, compromise TCP security since sequence numbers and ports are always checked as normal after TCP-in-UDP decapsulation, and in the event of TCP-in-UDP flow-id collisions, fallback to regular TCP is instant.

The use of a three-way handshake-in-handshake also prevents a man-in-the-middle attacker from inducing a desynchronized TCP-in-UDP state, wherein the receiver would believe it to be enabled whilst the sender believes it has failed, by dropping the SYN-ACK segment. See section 3.3.2.

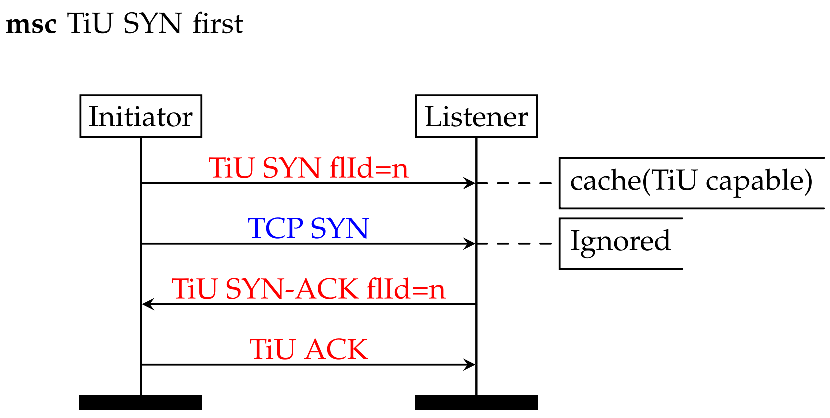

msc Success
Initiator
Listener
I
L
getFreeFlowId(I, L, src_port, dst_port) = n
TiU SYN flId=n
reserveFlowId(I, L, src_port, dst_port, n) = OK
TiU SYN-ACK flId=n
confirmFlowId(I, L, n, src_port, dst_port)
TiU ACK flId=n
confirmFlowId(I, L, n, src_port, dst_port)

(a) Successful negotiation

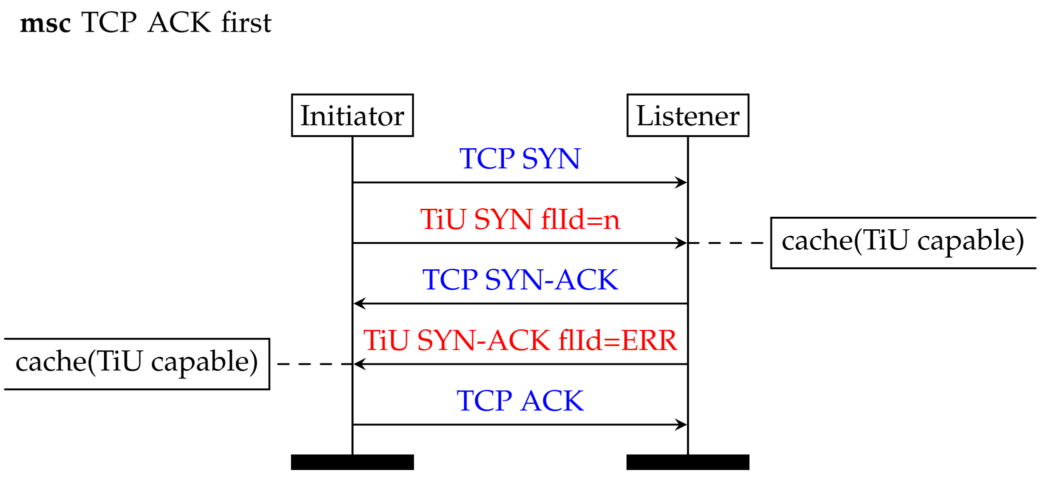

msc Failure with fallback
Initiator
Listener
I
L
getFreeFlowId(I, L, src_port, dst_port) = n
TiU SYN flId=n
reserveFlowId(I, L, src_port, dst_port, n) = FAIL
TiU SYN-ACK flId=ERR
releaseFlowId(I, L, src_port, dst_port, n)
Fallback
TCP ACK

(b) Flow id collision on listener, fallback

Figure 3.5: TCP-in-UDP flow identifier negotiation



(a) TCP-in-UDP arrives first



(b) TCP arrives first

Figure 3.6: TCP-in-UDP Happy Eyeballs

Chapter 4

Design: Coupled Congestion Control

While the TCP-in-UDP (TiU) encapsulation mechanism can be useful in its own right, it is mainly a means to an end. Our target has been to overcome the challenges imposed by multi-path routing on the feasibility of deploying coupled congestion control (CCC). In order to be able to test and validate the TiU mechanism, we contributed to the design of and implemented a CCC mechanism for TCP.

The mechanism we have developed is based on the Flow State Exchange (FSE) concept originally envisaged by Islam et al. for coupled congestion control in real-time media applications [47–49]. The original FSE system is part of the research and standardisation effort surrounding the WebRTC [57] family of standards and protocols for browser-based real-time communication, and is actively being specified within the IETF RMCAT[1] working group at the time of writing.

4.1 The Flow State Exchange concept

Figure 4.1 illustrates the high-level architecture of the FSE system. S1 and S2, and D1 and D2 are the sender-side, and receiver-side congestion controllers, respectively, for two concurrent flows running between the two end hosts depicted. On the sender side, the congestion controllers coordinate through the FSE.

As the name indicates, the purpose of the FSE is to allow congestion controllers to exchange state information. The FSE contains both a storage element and logic to appropriately share information between related flows. In other words, it is the central component of a coupled congestion controller.

An FSE can be either *passive* or *active*. The difference lies in whether an update coming from one connection will actively be pushed to any other connections coupled to it. In the situation depicted in figure 4.1, an active FSE might trigger S2 to send some data if S1 received an ACK

[1]RTP Media Congestion Avoidance Techniques

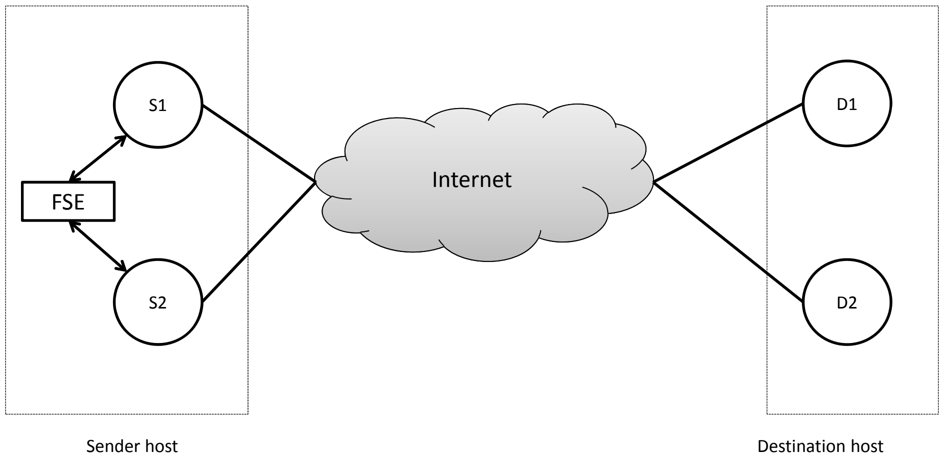


Figure 4.1: FSE abstract architecture [49]

and increased the congestion window sufficiently. In the passive case, the window increase would just be stored and not acted upon by S2 until the next time it made an update of its own.

4.2 Coupling Decision

The FSE must determine how to group flows together into flow groups. This grouping should reflect which flows it is appropriate to treat as an ensemble. A naïve approach would be to do what previous CCC solutions did, and base this choice on the source and destination address pair. While this may work in many instances, this approach is insufficient for general Internet use, as we have explained in section 2.2.

4.2.1 Dynamic bottleneck detection

Research into dynamically identifying flows that share a common bottleneck link using measurements has shown some promise. Rubenstein et al. [73] demonstrated the ability to identify TCP connections experiencing congestion at the same point in the network based on delay and loss observations. Recent work in the context of real-time media transport for WebRTC by Hayes et al. [29, 39] achieves the same for such flows and that mechanism is in fact intended to be integrated with the RMCAT FSE. However, to get the most out of CCC, we need to be able to accurately predict what flows will share bottlenecks *before* they even start transmitting. The aforementioned detection schemes are also most accurate only at times when the flows are actually suffering congestion, and they really only detect whether the flows share the same bottleneck, not whole network paths. Since TCP congestion control is highly sensitive to RTT differences and reordering, we must be reasonably certain that grouped flows really take the same paths through the network, or we risk degrading performance severely.

4.2.2 Bottleneck prediction

In light of this, we will instead actively attempt to force parallel connections down the same paths using the TCP-in-UDP encapsulation described in chapter 3, and group together those flows which are multiplexed within the same TiU tunnel.

4.3 Passive TCP Flow State Exchange

We originally investigated an FSE design for TCP coupled congestion control purposes based on the active FSE concept. We were able to identify several reasons why such a design is *unsuitable*:

- It interferes heavily with ACK clocking.
- The state of coupled connections becomes tightly intertwined, giving rise to complicated concurrency issues at implementation time.
- Unlike in the real-time communication application, TCP applications rarely adapt to the congestion situation in the network; When the sender is not application-limited, TCP flows are called “greedy” and the only adaptation they make is to send more if they think they can get away with it. RTC flows on the other hand generally have some upper bound on the data rate they will transmit at, and are far more sensitive to latency – they will actively reduce the rate at which they push data into the transport layer to respect latency bounds. The active FSE is more beneficial in those circumstances.

Instead, we settled on a variation of a passive FSE design. The principal advantages of the passive architecture are that:

- It does not get in the way of ACK-clocking as much; although, without pacing, large increases in `cwnd` could lead to (micro-)bursts.
- It is far easier to implement; that state which needs to be shared among connections can be deposited in the FSE. It does not matter if it is slightly out of date.

4.3.1 Features

Mildly conservative aggregate behaviour

The goal we have aimed for is an algorithm that makes an ensemble of coupled TCP flows behave roughly like a single ordinary TCP “New Reno” flow. In emulating that behaviour, it should try to do the right thing from the perspective of the controller of such a single “New Reno” connection. For instance, this means that if one flow experiences a timeout while others are still getting ACKs, we should be heavily prejudiced against treating the timeout as a true indication that all packets have been lost.

Straightforward implementation

The data structures and data flow of the FSE are designed to permit a straightforward implementation that does not require making radical changes to the existing network stack.

Weighted distribution of available bandwidth

When designing a CCC solution, one can often quite easily enhance the rate allocation algorithm so as to provide more flexible distribution functions that allocate bandwidth according to more flexible criterion than simple fair share. In keeping with the RMCAT FSE, we incorporate a simple, but powerful priority weighting bandwidth allocation scheme.

This allows the application to configure levels of precedence among individual flows, which is useful for ensuring Quality of Service (QoS). Unlike most other QoS solutions for Internet traffic, this scheme requires no assistance from the network, which unfortunately has proved to be a stumbling block for many interesting standards, e.g. Differentiated Services [RFC2475].

4.3.2 Algorithm

Variable	Description
c * †	Connection identifier
cwnd	TCP congestion window value
ssthresh	TCP slow-start threshold value
fse_cwnd *	FSE-calculated congestion window value
fse_ssthresh *	FSE-calculated slow-start threshold value value
state *	TCP state machine state: Slow-Start (SS), Congestion Avoidance (CA), or Fast Recovery (FR)
P *	Priority weight of a TCP flow
sum_cwnd †	sum of all the cwnd values in the flow group
sum_ssthresh †	sum of all the ssthresh values in the flow group
sum_P †	sum of all the priority values in the flow group
CoCo †	Coordinating Connection
fg_touched †	Timestamp of the last time an update, join or leave event was posted to flow group

Table 4.1: Variables used in algorithms 1, 2 and 4. Variables marked with an asterisk * are maintained in the FSE on a per-flow basis. Those marked with a dagger † are maintained per-flow group, while unmarked variables are part of each flow's TCP Control Block (TCB).

The idea behind the algorithm we are using is to designate one connection as a leader of the pack, the *Coordinating Connection* (CoCo), and control the rest of the ensemble according to the congestion state seen by that connection. To ensure appropriate reaction to loss, the CoCo duty is

Algorithm 1 FSE – connection registering

```
1: Input: c, cwnd, ssthresh, P
2: Output: fse_cwnd(c), fse_ssthresh(c)
3: if first connection in new coupled group then
4:     fse_cwnd(c) $\leftarrow$ cwnd
5:     sum_P $\leftarrow$ 0
6:     sum_cwnd $\leftarrow$ 0
7:     sum_ssthresh $\leftarrow$ 0
8:     CoCo $\leftarrow$ c
9: end if
10: fg_touched $\leftarrow$ now
11: fse_P(c) $\leftarrow$ P
12: sum_P $\leftarrow$ sum_P + P
13: sum_cwnd $\leftarrow$ sum_cwnd + cwnd
14: fse_cwnd(c) $\leftarrow$ P $\times$ sum_cwnd / sum_P
15: fse_ssthresh(c) $\leftarrow$ ssthresh
16: if sum_ssthresh $>$ 0 then
17:     fse_ssthresh(c) $\leftarrow$ P $\times$ sum_ssthresh / sum_P
18: end if
19: Send fse_cwnd(c) and fse_ssthresh(c) to c
```

Algorithm 2 FSE – connection leaving

```
1: Input: c
2: if c = CoCo then
3:     Coco $\leftarrow$ any active connection
4: end if
5: sum_P $\leftarrow$ sum_P - fse_P(c)
6: Free state associated with c
7: fg_touched $\leftarrow$ now
```

shifted to flows that experience congestion, but we are careful to avoid sending the entire ensemble into slow-start should a single RTO timer fire. This algorithm is an improved and corrected variant of the one specified in [82].

Data structures

The main data structures (state information) that make up the FSE storage element are listed in table 4.1.

The congestion control state (or phase) is stored in the FSE for two reasons: 1) this variable records what state each *individual* flow's congestion controller determined that flow should be in at the time of the update; the FSE may override the `cwnd` and `ssthresh` values as part of the update, so the state cannot be inferred from these values later on. 2) This information is needed about other flows during updates; storing it in the FSE means there is no need to access another flow's TCB as part of the update

Algorithm 3 FSE – flow group reaper

```
1: Input: fg_list
2: for all fg ∈ fg_list do
3:     if empty(fg) and (now - fg_touched(fg)) > 180s then
4:         Dispose of fg
5:     end if
6: end for
7: Sleep 90s
8: Goto 2
```

algorithm, which is prone to causing implementation problems related to locking.

Arrivals and departures

Algorithms 1 and 2 show how flows check in and out of the FSE.

When a new flow joins, it is joined to a *flow group* according to some coupling logic (not shown in the algorithm). In our case the logic groups those flows which are tunneled over the same TiU connection, i.e. those running between the same pair of source- and destination addresses that are in the **enabled** TiU state.

The new flow's existing congestion window size is added to the flow group's total, likewise for its priority weight. After updating the sums, it is then allocated its share of the total.

Notice how `ssthresh` is not propagated to the flow group at this time. It will only be inherited by the flow group after an actual loss event. In this way, we avoid propagating the initial "infinitely high" `ssthresh` of flows that have not experienced congestion yet.

When a flow leaves, we only remove its contribution to the sum of priorities. This is appropriate, since its share of the congestion window represents a part of the aggregate congestion window that the ensemble as a whole has measured. On their next updates, the other flows will each gain their rightful part of the departed flow's `cwnd`.

If the departing flow was CoCo, a new one is elected. While we don't show it here, we will prefer a flow that is in congestion avoidance if one can be found.

Algorithm 3 regularly purges old cached state. By allowing flow groups to persist for a short while after they empty, we provide *temporal sharing*. Joining flows will always seek to repopulate an already existing flow group if they find one. Due to how we do not subtract from the aggregate `cwnd` on departure, such flows will immediately gain the full window that was sensed by the previous ensemble. This is entirely appropriate as long as that information is still reasonably fresh (we expire cached flow groups after 180s by default), but poses an opportunity for potentially deleterious bursting if the flow does not pace the rate at which it starts consuming that window, see section 4.4.1.

Congestion control updates

Algorithm 4 on page 55 is the hearth of our FSE: this algorithm is executed each time the flows' native congestion controller has modified their congestion control state.

It makes decisions about which flow should become (or continue to be) the CoCo, and updates the aggregate congestion state. Each call to this algorithm will usually return modified congestion control variables to the flow, representing its share of the aggregate, except if a flow has suffered a loss event and meets certain criteria.

Non-CoCo update When a flow that isn't the CoCo makes an update, we first check if it is in a state that requires a change of leadership. At the moment, we always let the first flow to enter fast recovery take over the lead, since it has the most up to date congestion information. In all other cases, the individual flow's connection controller is overridden by assigning it its updated share of the aggregate.

CoCo update When the CoCo flow, which may change in the first part of the update algorithm, is making an update, the FSE will use the congestion control variables as updated by its individual congestion controller to update the aggregate's. In congestion avoidance, the aggregate is simply adjusted proportionally.

If, however, the CoCo is in fast recovery, then we let that mechanism do its work in peace so as to restore the normal operation of that flow as fast as possible. Only the aggregate `ssthresh` is updated at first, the `cwnd` correction comes as the CoCo leaves fast recovery.

If the CoCo has gone to slow-start, then either it is a) undergoing a ramp-up after initially starting up or having been idle for too long, or b) its RTO timer has expired. Allowing it to continue slow-start, and indeed forcing the entire ensemble into that mode with it, is only appropriate if all other flows are in the same situation. In case a), there is obviously no point in doing a ramp-up to probe the network if other flows already have a working ACK clock. Case b) entails, as we noted in section 2.1.3, that all packets in flight have gone missing – something severe has happened to the network. If other flows in the ensemble are receiving ACKs and are not experiencing timeouts, then obviously this is not the case. Therefore, we only allow this to happen if all other flows recently tried to enter slow-start.

Note that if the FSE is signaled after a timeout, the update will have been triggered from a special RTO-related code path. A single packet will always be retransmitted, ensuring the ACK clock does not completely stall should this have been a real timeout. The connection could be severely impacted however, as it will take several RTTs before it receives enough duplicates ACK to enter into fast recovery. Clearly, this part of the algorithm could be improved, but that is out of the scope of this project.

4.4 Limitations

Our TCP FSE design has some limitations, some of which are due to outside factors, others are due to the prototypical nature of the present mechanism. We will enumerate the most important caveats here.

4.4.1 Lack of pacing

As there is no built-in support for TCP pacing in FreeBSD, and as designing and implementing such a mechanism is very much a non-trivial undertaking, we will have to function without pacing. Recalling that we identified this as an important building block of a well-functioning TCP CCC solution in section 2.4.3, the performance of our solution is clearly going to suffer from this shortcoming.

4.4.2 Underutilization when idle

Since we do not have any scheduler, the FSE will blindly allocate a share of the congestion window to each flow according to their configured weights, without taking into account whether they are actually able to make use of that share. In situations where coupled flows undergo idle periods, this will lead to underutilization of the available bandwidth.

A possible enhancement to reduce this performance loss would be to temporarily remove flows from the pool that gets allocated a share of the `cwnd` if they have not posted an update to the FSE for some (relatively short) period of time.

4.4.3 Hardware acceleration

Most modern Network Interface Cards (NICs) provide a range of hardware acceleration features. These span from simply offloading packet checksum calculations to advanced hardware processing of transport layer logic, e.g. TCP Offload Engines. Less advanced transport protocol hardware assists such as Large Segment Offload / TCP Segmentation Offload (TSO) and the converse Large Receive Offload lie somewhere in-between on this spectrum.

Aside from checksum offloading, all of the aforementioned features move important parts of the TCP logic into the firmware of the NIC, where it is difficult or impossible for us to control it. Incidentally, TiU encapsulation is impaired for the same reasons.

Therefore, we cannot combine CCC with the use of such advanced acceleration features. On the one hand, for typical end-user client machines, this is not likely to have any appreciable impact. On the other hand, it may be a disadvantage for busy servers potentially on the other end of the connection. When that is said, hardware acceleration can also sometimes induce sub-optimal performance on its own, see [87].

Algorithm 4 FSE - connection update

```
Input: c, cwnd, ssthresh, state
Output: fse_cwnd(c), fse_ssthresh(c)
fse_state(c) ← state    ▷ Store what state the controller wanted to be in
fg_touched ← now

if CoCo ≠ c then
  if state=FR ∧ fse_state(x) = CA ∀x ≠ c then
    CoCo ← c    ▷ If c is the only flow in FR, make it the new CoCo
  else
    if state=CA ∨ state=SS then
        ▷ If c is neither CoCo nor in FR, update c's share and override
      fse_cwnd(c) ← fse_P(c) × sum_cwnd / sum_P
      fse_ssthresh(c) ← fse_P(c) × sum_ssthresh / sum_P
    end if
  end if
end if

if CoCo = c then
  if state = CA then    ▷ Normal happy day update
    if cwnd ≥ fse_cwnd(c) then    ▷ Increased cwnd
      sum_cwnd ← sum_cwnd + cwnd - fse_cwnd(c)
    else    ▷ Proportional reduction of sum_cwnd
      sum_cwnd ← sum_cwnd × cwnd / fse_cwnd(c)
    end if

    fse_cwnd(c) ← fse_P(c) × sum_cwnd / sum_P
    fse_ssthresh(c) ← ssthresh

    if sum_ssthresh > 0 then
      fse_ssthresh(c) ← fse_P(c) × sum_ssthresh / sum_P
    end if
  else if state = FR then    ▷ Allow FR to carry on
    sum_ssthresh ← sum_cwnd/2
  else if state = SS then    ▷ Check if SS is really necessary
    if fse_state(x) = SS ∀x ≠ c then    ▷ Everyone wants SS
      sum_ssthresh ← sum_cwnd/2
      sum_cwnd ← sum_cwnd × cwnd / fse_cwnd(c)
      fse_cwnd(c) ← fse_P(c) × sum_cwnd / sum_P
    else    ▷ Someone is still getting ACKs. Let them lead.
      CoCo ← first connection where fse_state ≠ SS
    end if
  end if
end if

if state ≠ FR then
  Send fse_cwnd(c) and fse_ssthresh(c) to c
end if
```

Chapter 5

FreeBSD implementation

The TCP-in-UDP mechanism with TCP CC coupling will be implemented in the FreeBSD[1] operating system's network stack. FreeBSD was chosen because it is a high quality Free Software operating system, with a well documented (see e.g. [61, 85]), standards-compliant [21] and well-performing IP network stack.

5.1 Application layer interface

5.1.1 Control interface

Our implementations is controllable using a combination of both the `sysctl` (system control) configuration system for administratively setting system-wide policies, as well as socket options allowing per-connection configuration by the applications.

Both TiU and the FSE fully support the VIMAGE network virtualisation [86] feature of FreeBSD, allowing them to be selectively enabled and otherwise configured per virtual network stack. This proved advantageous to us during development and testing.

5.1.2 Transparency

A major goal of the implementation of both the TCP-in-UDP and CCC schemes has been to make them as transparent as possible to the application layer. This way, legacy applications can benefit from them without any adaption, much like they would from other TCP congestion control improvements.

However, as discussed in sections 3.2.1 and 3.4.1, TiU does replace the TCP header information supporting the Urgent Data feature of TCP. Consequently, applications that rely on that feature will not function correctly if TiU is enabled. If such applications need to run on a system where one desires to use TiU, there are still some options, though:

- Modifying the offending application to explicitly disable TiU on its sockets.

[1] http://freebsd.org

- Setting TiU off by default, and modifying other applications to explicitly request it on their sockets.
- Enabling VIMAGE network virtualisation, and confining either the applications that require Urgent Data, or the applications that should use TiU, in appropriately configured jails.[2]

The latter option may often be preferable, since it requires no modifications to the applications themselves.

5.1.3 Conventions

In the following sections, we will present implementation details, including some source code listings.

You may notice many calls to macros with names like `CTR`*x*. These are calls to the kernel tracing facility, which is a kernel debugging log framework. All of these calls are compiled out in release mode builds, such as those we used to evaluate the implementation in chapter 6.

For brevity's sake, we will skip over source excerpts that only consist of logging output or lengthy comments wherever it makes sense. Hence, there are some "jumps" in line numbering in a few places. Likewise, we will refrain from drilling down into the most mundane functions that only perform actions like looking up entries in hash tables, etc. The particularly interested reader is invited to consult the full source code, see appendix A.

A number of variables have names of the form `V_`*name*. These are in reality macro invocations that will fetch variables attached to the current virtual network stack (VNET) instance; if VIMAGE is not enabled, there is only one such instance and the macros compile to a direct reference to a single variable.

We have opted not to detail the data structures we use in great detail here, since the code should make it quite obvious how they work. Both the encapsulation and coupled congestion control components make use of the standard queue (list) and hash table libraries included in FreeBSD.

5.2 TCP-in-UDP encapsulation

Conceptually, our implementation of TCP-in-UDP encapsulation (TiU) can be considered as a kind of shim sub-layer that sits near the bottom of the transport layer, wedged between TCP and UDP. Figure 5.1 illustrates the flow of data through the TiU-enabled network stack.

All changes necessary to implement TiU are made at the transport layer, the only visible change to upper layers is the addition of socket options that allow applications to (optionally) configure TiU use. Lower layers simply see TiU encapsulated traffic as UDP traffic and treat it as such.

[2]The jail mechanism [52] is a lightweight virtualisation framework built into FreeBSD. The previously discussed VIMAGE feature enables each jail to have its own virtualised network stack.

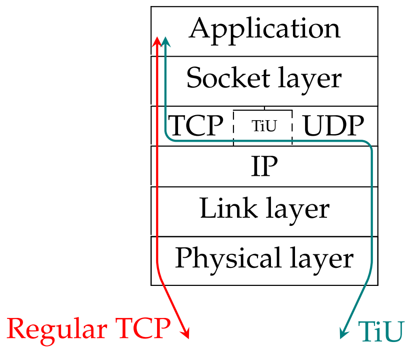


Figure 5.1: TiU data flow

Most of the present section will detail some of the most important code sections that make up our TiU implementation. We present them in an approximation of the order they will be executed when initiating a connection and transmitting data.

5.2.1 Modularity

In a bid to keep our implementation easily understandable, we have strived to modularize the code to the extent it was reasonable. By separating out most of our code from the existing network stack implementation, it becomes easier to keep track of the changes introduced by our work.

Most of the actual code-wise changes have been made in the form of new functions defined in a new source file, `tcp_hijack.c` [3].

Hooks that call into these functions, and some code related to connection setup, have been inserted directly into the main TCP input and output code, as well as in the TCP SYN- and host-caches code, in `tcp_input.c`, `tcp_output.c`, `tcp_syncache.c` and `tcp_hostcache.c`, respectively. Some other initialization and management code is also distributed among the other TCP source files.

In order to capture incoming TiU segments, we have inserted a hook into the UDP input path. Likewise, we've also hooked the UDP control input routine, which lets us handle ICMP errors destined for the tunnel endpoint. Both of these changes are made in the file `udp_usrreq.c`.

5.2.2 Connection setup

The TCP connection establishment routines have been extended to support the TiU happy-eyeballs connection establishment mechanism as described in section 3.3.2.

[3]We have used the prefix `tcp_hijack` to define a kind of namespace for most our functions and variables in the kernel. The name is due to the fact that TiU "hijacks" packets, diverting their normal processing flow.

Connection creation

When an initiator application makes calls using the sockets API to connect a TCP socket, the `tcp_connect` function initializes the state of the TCB.

tcp_usrreq.c: initialize initiator's TCB TiU state

```
  /* Init. TCPinUDP flags from hostcache: */
  if (V_tcp_hijack_enabled) {
    tp->t_hijackflags |= tcp_hc_get_tcpinudp(&inp->
        inp_inc);
    if (V_tcp_hijack_default_disabled
        && !(tp->t_hijackflags & TCP_HIJACK_WANTED))
      tp->t_hijackflags |= TCP_HIJACK_DISABLED;
    /*
     * If we end up with the all clear to try TiU to a
     * host with unknown TiU capability, enable parallel
     * TCP/UDP probing:
     */
    if (!(tp->t_hijackflags & (TCP_HIJACK_DISABLED |
        TCP_HIJACK_CAPABLE)))
        tp->t_hijackflags |= TCP_HIJACK_PROBING;

    CTR2(KTR_TCPINUDP, "tcp_connect: TiU flags=%0x faddr
        =%x\n",
         tp->t_hijackflags, inp->inp_inc.inc_faddr.s_addr
             );
  }
```

When doing so, we check the host cache to see if we know anything about the other end's TiU capability. If it is known to support it, immediately try with TiU. If we know it has resulted in permanent errors before, do not attempt it at all. In other cases, use Happy Eyeballs to see if it is possible to use TiU.

Three-way-handshake: SYN

tcp_output.c: SYN construction

```
    if ((flags & TH_SYN) && V_tcp_hijack_enabled) {
      CTR1(KTR_TCPINUDP, "tcp_output: SYN: TiU flags=%0x\
          n",
          tp->t_hijackflags);
      if (!(tp->t_hijackflags & TCP_HIJACK_DISABLED)) {
        CTR0(KTR_TCPINUDP,
             "tcp_output: TCPinUDP not disabled for this
                 connection\n");
```

This is part of the code, in the function `tcp_output`, that constructs SYN segments. First, check whether TiU is globally enabled in this VNET, and then whether TiU has been explicitly disabled for this connection.

tcp_output.c: Flow ID assignment

```
755          /* After the max retry limit we fall back to
                 regular TCP */
756          if (tp->t_tiucb->tiu_setup_attempts <
                 V_tcp_tiumaxretries) {
757            /* Avoid reassigning flowIds on retransmits: */
758            if (tp->t_tiucb->tiu_flowid ==
                   TCPINUDP_INVALFLOWID)
759              to.to_tiuflow = tcp_hijack_connect(tp);
760            else
761              to.to_tiuflow = tp->t_tiucb->tiu_flowid;
762
763            CTR1(KTR_TCPINUDP, "tcp_output: flowid=%hhu\n",
764                 to.to_tiuflow);
765
766            if (to.to_tiuflow != TCPINUDP_INVALFLOWID) {
767              to.to_flags |= TOF_TCPINUDP;
768              tp->t_tiucb->tiu_setup_attempts += 1;
769              CTR1(KTR_TCPINUDP,
770                   "tcp_output: TiU SYN attempt %u\n",
771                   tp->t_tiucb->tiu_setup_attempts);
772            }
```

Attempt to reserve a flow ID for this connection, as long as this isn't a SYN retransmission that would exceed the configured retry limit. The default is to only try twice, then give up on TiU.

tcp_output.c: SYN retry limit

```
773          } else {
774            tp->t_hijackflags &=
775              ~(TCP_HIJACK_ACTIVE | TCP_HIJACK_PROBING);
776            CTR0(KTR_TCPINUDP,
777                 "tcp_output: exceeded max TiU SYN rxmit
                       limit.\n");
778          }
779        }
780      }
```

This code ensures TiU is disabled and no further probing will take place if we exceeded the retry threshold (see line 756).

tcp_output.c: parallel output of TiU and TCP SYN

```
1386     /* NB SYN-ACK doesn't take this code path, hence no
1387      * special treatment for it */
1388     struct mbuf *m_tcp;
1389     int tiu_dorace = tp->t_hijackflags &
             TCP_HIJACK_PROBING;
1390     if (tiu_dorace) {
1391       /* We create a copy of the mbuf chain, since
1392        * the original chain will be consumed when it
1393        * goes down the stack */
1394       m_tcp = m_dup(m, M_NOWAIT);
1395     }
1396
```

```
    error = tcp_hijack_output(m, tp->t_inpcb->inp_options
        , &ro,
            ((so->so_options & SO_DONTROUTE) ?
                IP_ROUTETOIF : 0),
            0, tp->t_inpcb);

    if (tiu_dorace) {
      /* NOP out TiU option */
      ip = mtod(m_tcp, struct ip *);
      th = (struct tcphdr *)(ip + 1);
      memset((uint8_t*)(th + 1) + to.to_tiuoff,
          TCPOPT_NOP, TCPOLEN_TCPINUDP);
      error = ip_output(m_tcp, tp->t_inpcb->inp_options,
          &ro,
            ((so->so_options & SO_DONTROUTE) ?
                IP_ROUTETOIF : 0), 0,
            tp->t_inpcb);
    }
```

This code is responsible for handing the segment to the TiU shim-layer. If we are doing the Happy Eyeballs probing procedure, the segment is duplicated and the duplicate segment's TiU setup option is overwritten with no-op sequences to avoid confusing any potential middleboxes, before being handed straight to the IP layer. This code is executed sequentially on the same thread (ensured by locking), so the TiU segment will be processed first and should get a head start onto the wire.

The actual encapsulation and decapsulation is documented in section 5.2.4.

Listener side SYN handling

tcp_input.c: detect TiU segments

```
  /*
   * We need to know if a packet came over the tunnel
       when
   * handling the three way handshake.
   */
  if (V_tcp_hijack_enabled
      && ((thflags & TH_SYN) || (tp->t_hijackflags &
          TCP_HIJACK_OFFERED))) {
    tiu_tunneled = (m_tag_locate(m, TCP_TIUMTAG_COOKIE,
        TCP_TIUMTAG, NULL)
          != NULL);
  }
```

During setup, some of the code we have embedded in the main TCP input handling needs to know if the current segment cave from the TiU tunnel. We recognize this by checking for the presence of a tag [54] attached to the mbuf by the decapsulation function.

tcp_input.c: parse TiU setup option

```
    case TCPOPT_TCPINUDP:
      if (optlen != TCPOLEN_TCPINUDP)
        continue;
      if (ntohs(*(uint16_t*)(cp + 2)) !=
          TCPOEXID_TCPINUDP)
        continue;
      to->to_flags |= TOF_TCPINUDP;
      to->to_tiuflow = cp[4];
      CTR1(KTR_TCPINUDP, "Saw a TCP-in-UDP enabled flag,
          flowId=%hhu\n",
            cp[4]);
      break;
```

Here we recognize and parse the TiU setup option.

SYN arrival: syncache_add The `syncache_add` function is called when TCP receives a SYN.

tcp_syncache.c: handle incoming SYN: TiU SYN

```
  if ((to->to_flags & TOF_TCPINUDP) &&
      V_tcp_hijack_enabled) {
    /* raise TCP-in-UDP flag */
    sc->sc_flags |= SCF_TCPINUDP;
    /* Try to reserve flow; if fails, send invalid id
     * back to signal failure */
    if (to->to_tiuflow > TCPINUDP_MAXFLOWID ||
        tcp_hijack_reserveflow(inc, to->to_tiuflow) != 0)
      sc->sc_tiuflow = TCPINUDP_INVALFLOWID;
    else
      sc->sc_tiuflow = to->to_tiuflow;
    /* Mark this host as TiU capable */
    tcp_hc_update_tcpinudp(inc, TCP_HIJACK_CAPABLE);
    CTR0(KTR_TCPINUDP, "syncache_add: saw TiU option.\n")
        ;
  }
```

Here an incoming SYN with the TiU setup option is handled. The offered flow id. is checked against the local map for collisions; if there is one, it will be rejected and we will fall back to regular TCP, otherwise it is echoed back in the SYN-ACK.

tcp_syncache.c: handle incoming SYN: late TCP SYN

```
    /* With TCP-in-UDP we send simultaneous SYNs over
     * both TCP and UDP, so we are bound to get dups most
     * the time */
    if (V_tcp_hijack_enabled) {
      int tunneled = (m_tag_locate(m, TCP_TIUMTAG_COOKIE,
           TCP_TIUMTAG, NULL)
           != NULL);
      /* XXX This check really means
         (TiU enabled XOR tunneled) -> drop */
      if (sc->sc_flags & SCF_TCPINUDP &&
           !tunneled) {
            /* TCP SYN arriving late, silently drop */
            /* XXX what if this was a rxmit instead?! */
            CTR0(KTR_TCPINUDP, "syncache_add: late TCP SYN,
               dropping\n");
            SCH_UNLOCK(sch);
            goto done;
```

Usually duplicates just trigger a retransmit and an increment of the retransmit counter, but we need to handle that a bit differently. This first part detects a TCP SYN arriving after a TiU SYN already created a SYN cache entry; it will be ignored.

tcp_syncache.c: handle incoming SYN: late TiU SYN

```
          } else if ((sc->sc_flags & SCF_TCPINUDP) == 0 &&
              tunneled)
          {
            /*
             * UDP SYN arriving late, we need to
             * send a reply to let the other end
             * know we support TiU
             */
            CTR0(KTR_TCPINUDP,
                 "syncache_add: late UDP SYN, sending dummy
                    response\n");
            sc->sc_flags |= SCF_TCPINUDP;
            sc->sc_tiuflow = TCPINUDP_INVALFLOWID;
            syncache_respond(sc, sch, 1);
            sc->sc_flags &= ~SCF_TCPINUDP;
            SCH_UNLOCK(sch);
            goto done;

          }
```

This is the same case for a late TiU SYN. It will be ignored, but we do send a reply to allow the initiator to know that we in fact do support TiU.

tcp_syncache.c: handle incoming SYN: retransmission

```
          /* If we got here, this was a SYN rxmit over
           * the same transport as the first we saw,
           * proceed to rxmit SYN-ACK */
          CTR2(KTR_TCPINUDP, "syncache_add: TiU dup SYN -
             flags=%0x tun=%d\n",
              sc->sc_flags, tunneled);
        }
```

If we get here, it really was a retransmission we received; the standard behaviour will ensue.

SYN response: syncache_respond

tcp_syncache.c: Echo back the flow id.

```
    if (sc->sc_flags & SCF_TCPINUDP) {
      /* TCP-in-UDP capability back, inform of
       * flowid decision */
      to.to_flags |= TOF_TCPINUDP;
      to.to_tiuflow = sc->sc_tiuflow;
    }
```

Part of the syncache_respond function, which constructs and sends back a SYN-ACK segment. If we got an acceptable TiU flow id. offer, echo it back using a TiU setup option.

tcp_syncache.c: divert response over TiU

```
    if (sc->sc_flags & SCF_TCPINUDP)
      error = tcp_hijack_output(m, sc->sc_ipopts, NULL,
          0, NULL, NULL);
```

This code makes sure a TiU SYN-ACK response is sent for encapsulation.

SYN-ACK retransmissions: syncache_timer

tcp_syncache.c: limit number of TiU SYN-ACK retransmissions

```
    if ((sc->sc_flags & SCF_TCPINUDP) && sc->sc_rxmits >
        V_tcp_tiumaxretries) {
      /* Give up UDP, might be blocked */
      sc->sc_flags &= ~SCF_TCPINUDP;

    }
```

This is part of the syncache's separate timeout handling code. We will only allow a few (default: 2) retransmissions of the SYN-ACK segment over TiU, because we might be losing those segments due to some permanent problem, like a firewall blocking all outbound UDP traffic (or vice-versa at the initiator's end).

5.2.3 Initiator side SYN-ACK handling

tcp_input.c: SYN-ACK handling; TiU comes back

```
    if (V_tcp_hijack_enabled) {

      if ((to.to_flags & TOF_TCPINUDP) && tiu_tunneled) {
        tp->t_hijackflags |= TCP_HIJACK_CAPABLE;
        CTR0(KTR_TCPINUDP, "tcp_input: saw TCP-in-UDP
            capability\n");
        tcp_hc_update_tcpinudp(inc, tp->t_hijackflags);

        /* Other end rejected the flowid, fallback to TCP
           */
        if (tp->t_hijackflags & TCP_HIJACK_OFFERING &&
            to.to_tiuflow != tp->t_tiucb->tiu_flowid) {
          tcp_hijack_disconnect(tp);
```

```
            tp->t_hijackflags &= ~(TCP_HIJACK_ACTIVE | TCP_HIJACK_PROBING);
        } else {
            /* Tunnel confirmed */
            tp->t_hijackflags |= TCP_HIJACK_ENABLED;
            tp->t_hijackflags &= ~TCP_HIJACK_PROBING;
            CTR2(KTR_TCPINUDP, "tcp_input: tunnel confirmed to %x "
                 "with flowid=%d\n",
                 inc->inc_faddr.s_addr,
                 to.to_tiuflow);
        }
    } else if (!tiu_tunneled && tp->t_hijackflags & TCP_HIJACK_PROBING) {
```

This is where we handle the SYN-ACK segment and determine whether TiU or TCP should continue to be used. If it came back over TiU, the host cache entry for the corresponding host is updated to reflect that we have learnt that it supports TiU. Then the flow id. passed in the TiU setup option is checked for validity. In case of an error, TiU is abandoned for the current connection, otherwise the use of TiU has been confirmed and will be used for the remainder of the session.

tcp_input.c: SYN-ACK handling; TCP comes back

```
    } else if (!tiu_tunneled && tp->t_hijackflags & TCP_HIJACK_PROBING) {
        /* Got SYN-ACK back over TCP => TCP
         * won the race this time */
        tcp_hijack_disconnect(tp);
        tp->t_hijackflags &= ~(TCP_HIJACK_ACTIVE | TCP_HIJACK_PROBING);
        CTR1(KTR_TCPINUDP, "tcp_input: TCP won SYN-race connecting to %x\n",
             inc->inc_faddr.s_addr);
    }
```

This is what happens in the opposite case, of a TCP SYN-ACK. TiU will not be used any more for the remainder of this session.

tcp_input.c: late TiU SYN arrival

```
    if (tiu_tunneled && (tp->t_hijackflags & TCP_HIJACK_ACTIVE) == 0) {
        /* UDP lost, but endpoint supports TiU */
        tp->t_hijackflags |= TCP_HIJACK_CAPABLE;
        tcp_hc_update_tcpinudp(inc, tp->t_hijackflags);
        CTR1(KTR_TCPINUDP, "tcp_input: endpoint %x supports TCP-in-UDP\n",
             inc->inc_faddr.s_addr);
        goto drop;
    }
```

Here we detect a TiU SYN segment at a time we have already decided to use TCP. The capability of the other end is cached and the segment otherwise ignored.

Listener side handshake conclusion

tcp_input.c: end of three-way-handshake: TiU confirmed

```
    if (V_tcp_hijack_enabled) {
      if (tiu_tunneled && (tp->t_hijackflags & TCP_HIJACK_OFFERED)) {
        tp->t_hijackflags |= TCP_HIJACK_ENABLED;
        tp->t_hijackflags &= ~TCP_HIJACK_OFFERED;
        CTR2(KTR_TCPINUDP, "tcp_input: 3WHS succeeded, tunnel to "
            "%x up on flowid %d.\n", inc->inc_faddr.s_addr,
            tp->t_tiucb->tiu_flowid);
      }

    }
```

Here the listener receives the third segment of the three-way-handshake (ACK) over TiU after having been offered a TiU tunnel, so the TiU encapsulation is confirmed at the server side.

tcp_syncache.c: save TiU state from SYN cache into TCB

```
#ifdef TCP_IN_UDP
  /*
   * Copy in TCPinUDP state we negotiated.
   */
  if (sc->sc_flags & SCF_TCPINUDP) {
    tp->t_hijackflags |= TCP_HIJACK_OFFERED;
    tp->t_tiucb->tiu_flowid = sc->sc_tiuflow;
  }
#endif
```

This code is part of the function `syncache_socket`, which creates the permanent data structures representing a TCP connection's state upon a three-way-handshake concluding successfully. We cache the TiU flow id. in the SYN cache, so we must update the newly created TCB with it. The TiU state at this point is **offered**, which is necessary for further processing in the main TCP input path to make the right choices.

5.2.4 Encapsulation

In the encapsulation step proper, the TCP segment header is rewritten as described in section 3.2. Then the segment is passed as outgoing payload data to UDP.

Proper TCP segments will be reconstructed before being passed to normal TCP processing upon reception at the destination. This involves splicing out the UDP header and rewriting both the IP and TCP headers – the FreeBSD transport layer expects to be presented with a full IP datagram.

The encapsulation and decapsulation steps introduce some minor delay into packet processing, notably due to having to carry out some additional copy operations on the message buffers in order to rewrite them.

It would have possible to alleviate this slightly by embedding the implementation deeper within the existing TCP and UDP implementation, however we have avoided this to reduce the complexity of the code.

Encapsulation preparation: segment construction

tcp_output.c: copy from socket buffer

```
#ifndef TCP_IN_UDP
    if (len <= MHLEN - hdrlen - max_linkhdr) {
      m_copydata(mb, moff, (int)len,
          mtod(m, caddr_t) + hdrlen);
      m->m_len += len;
    } else
#endif  /* TCP_IN_UDP */
    {
      m->m_next = m_copy(mb, moff, (int)len);
      if (m->m_next == NULL) {
        SOCKBUF_UNLOCK(&so->so_snd);
        (void) m_free(m);
        error = ENOBUFS;
        sack_rxmit = 0;
        goto out;
      }
    }
```

Copy the appropriate amount of data from the upper layer socket buffer into the *mbuf* we are constructing the current segment in. Mbufs are buffers with attached meta-data that can be chained together, a data structure used throughout the network stack. Normally, TCP will append data into the first mbuf if there is enough space left in it, after the IP and TCP headers as well as space reserved for the Ethernet link layer header, to hold the amount of data we are about to send. When TiU is enabled in the kernel build, we change this behaviour to always copy data into a new mbuf, which is then chained onto the one containing the headers. This way, we can minimize the amount of buffer copying in the encapsulation step, because we will only need to move the relatively short TCP header in order to fit the UDP header in, without having to touch payload data. [4]

Diversion to encapsulation

tcp_output.c: output to TiU/IP

```
#ifdef TCP_IN_UDP
  if (V_tcp_hijack_enabled && (tp->t_hijackflags &
      TCP_HIJACK_ACTIVE))
```

[4]It should be noted here that there may be "gaps" at the beginning and end of an `mbuf`'s data field, but not in the middle of it.

```
  {
    error = tcp_hijack_output(m, tp->t_inpcb->inp_options
        , &ro,
             ((so->so_options & SO_DONTROUTE) ?
                 IP_ROUTETOIF : 0),
             0, tp->t_inpcb);
  } else
#endif  /* TCP_IN_UDP */
  {
    error = ip_output(m, tp->t_inpcb->inp_options, &ro,
          ((so->so_options & SO_DONTROUTE) ? IP_ROUTETOIF
              : 0), 0,
          tp->t_inpcb);
  }
```

At this point, the segment has been constructed fully. Determine whether to output it to IP as normal, or divert it through the TiU encapsulation routine. We have omitted code covered in section 5.2.2.

Encapsulation

Actual encapsulation of the segment is performed by the `tcp_hijack_output` function.

tcp_hijack.c: initial encapsulation checks

```
int
tcp_hijack_output(struct mbuf *m, struct mbuf *opt,
    struct route *ro, int flags,
      struct ip_moptions *imo, struct inpcb *inp)
{
  int error = 0;
  struct ip *ip = NULL;
  struct in_addr src, dst;
  struct sockaddr_in sin;
  struct tcpcb *tp = NULL;
  uint8_t flowid = TCPINUDP_INVALFLOWID;

  if (!V_tcp_hijack_enabled) {
    return ENETDOWN;
  }

  if (inp != NULL) {
    tp = intotcpcb(inp);
    if (tp != NULL)
      flowid = tp->t_tiucb->tiu_flowid;
  }

  if (!PFIL_HOOKED(&V_tiu_pfil_hook))
    goto rewrite;

  if (pfil_run_hooks(&V_tiu_pfil_hook, &m, NULL, PFIL_OUT
      , inp)) {
```

```
    CTR0(KTR_TCPINUDP, "tcp_hijack_output: pfil ate our packet!\n");
    return IPPROTO_DONE;
  }
```

Do some sanity checks and find the flow id. of this connection. Before rewriting the segment, feed it into the `pfil` framework, which will let SIFTR (see section 6.1.2) trace it.

tcp_hijack.c: header rewrite and UDP pass-down

```
 rewrite:
  /* Copy out src and dst addresses from IP header, then rip it off */
  ip = mtod(m, struct ip *);
  src = ip->ip_src;
  dst = ip->ip_dst;
  m_adj(m, sizeof(struct ip));

  memset(&sin, 0, sizeof(struct sockaddr_in));
  sin.sin_family = AF_INET;
  sin.sin_port = htons(V_tcp_hijack_port);
  sin.sin_addr = dst;
  sin.sin_len = sizeof(struct sockaddr_in);
  tcp_hijack_thcomp(m, flowid, (tp != NULL && tp->t_state >= TCPS_ESTABLISHED));
  error = udp_usrreqs.pru_send(V_tcp_hijack_tsock, 0, m, (struct sockaddr *)&sin,
             NULL, curthread);
  CTR1(KTR_TCPINUDP, "tcp_hijack_output: UDP returned %d\n", error);
  return (error);
}
```

Extract the IP addresses from the IP header, and remove it; UDP will all one instead. The actual rewriting is performed by a helper function, `tcp_hijack_thcomp` which we present below. Finally, pass the modified packet buffer into UDP as payload. It will not receive any special handling in the UDP transmission code path.

tcp_hijack.c: header rewrite, common case

```
static __inline void
tcp_hijack_thcomp(struct mbuf *m, uint8_t flowid, int rewrite)
{
  struct tcphdr *th;
  uint8_t swapb[4];   /* For reordering FIXME magic number? */

  th = mtod(m, struct tcphdr *);
  if (rewrite) {
    /* Encode flowid (4 highest bits into reserved bits, lowest
     * bit into URG flag) */
    th->th_x2 = (flowid >> 1); /* 4 highest bits */
    /* lowest bit; can I make a bitwise expression? */
    if (flowid & 1) {
      th->th_flags |= TH_URG;
    } else {
      th->th_flags &= ~TH_URG;
    }

    /* Reorder offset, overwriting ports */
    bcopy(TCPHDR_OFF(th), &th->th_sport, sizeof(swapb));

    /* Overwrite Urgent Pointer + Checksum fields by moving up
     * whatever comes after (options) */
    if (m->m_len > sizeof(struct tcphdr)) {
      bcopy(th + 1, TCPHDR_OFF(th), m->m_len - sizeof(struct tcphdr));
    }

    m->m_len -= 8;  /* Length of cksum + urgptr + {s,d} ports fields */
    m->m_pkthdr.len -= 8;
```

Here we see how the `tcp_hijack_thcomp` routine modifies the header to the format presented in section 3.2, for the common, non-SYN/SYN-ACK case.

tcp_hijack.c: header rewrite, SYN/SYN-ACK case

```
  } else {
    th->th_off = TCPINUDP_SETUPSEGMENT;

    /* Swap ports and offset... */
    bcopy(&th->th_sport, swapb, sizeof(swapb));
    bcopy(TCPHDR_OFF(th), &th->th_sport, sizeof(swapb));
    bcopy(swapb, TCPHDR_OFF(th), sizeof(swapb));
  }
}
```

The final part of the routine only swaps the ports and offset-row to produce the setup header format in figure 3.3.

Receive side UDP handling

udp_usrreq.c: redirect TiU segments into TiU decapsulation routine

```
  /* Detect TCP-in-UDP tunnel receive */
  if (V_tcp_hijack_enabled && inp->inp_socket == V_tcp_hijack_tsock) {
    CTR1(KTR_TCPINUDP, "TCP-in-UDP received tunneled data (%u bytes)... \n",
        m_length(n, NULL));

    int eredir;
    /* Drop lock before continuing */
    INP_RUNLOCK(inp);
    eredir = tcp_hijack_input(&n, &off, IPPROTO_TCP);
    CTR1(KTR_TCPINUDP, "udp_append: eredir=%d.\n", eredir
       );

    /* Reacquire lock because udp_input unlocks it */
    /* TODO change this behaviour? */
    INP_RLOCK(inp);
    return;
```

The function `udp_append` usually appends the payload attached to an incoming datagram to the end of the socket buffer. We hook this to instead divert datagrams that contain TiU encapsulated TCP segments into our decapsulation code, which will subsequently pass it into the TCP stack.

udp_usrreq.c: detect ICMP errors destined for TiU endpoint

```
      /* TCPinUDP tunnel errors need
       * special handling: */
      /* XXX: Tunnel socket doesn't get bound to
       * each destination, hence no inp match for it */
      if (V_tcp_hijack_enabled &&
          uh->uh_dport == htons(V_tcp_hijack_port)) {
        CTR0(KTR_TCPINUDP, "ICMP error matched tunnel");
        tcp_hijack_ctlinput(ip, inetctlerrmap[cmd]);
      } else {
```

Sometimes we might get a response back from the listener side in the form of an ICMP "Connection Refused" error. This normally means the port was closed. Obviously, if the TiU port is not open, there can be no TiU tunnel. Here we detect and properly handle such error feedback within the `udp_common_ctlinput` function.

TCP error responses

tcp_subr.c: capture RST segments that should be encapsulated

```
    if (m_tag_locate(m, TCP_TIUMTAG_COOKIE, TCP_TIUMTAG,
       NULL) != NULL ||
        (tp != NULL && tp->t_hijackflags &
           TCP_HIJACK_ENABLED))
      tcp_hijack_output(m, NULL, NULL, ipflags, NULL, inp
         );
```

Sometimes a host may respond to a connection attempt or other message with a reset, e.g. if a port is closed or the connection becomes completely desynchronized. These resets are handled by a special function, `tcp_respond`, which sends the appropriate response by checking the header of the segment that triggered the response. Therefore, we must add another hook for TiU encapsulation here, based on detecting the TiU mbuf tag which the decapsulation mechanism appends to all incoming segments.

Decapsulation

`tcp_hijack_input` is the function that receives UDP datagrams that are identified as being destined for the tunnel endpoint.

tcp_hijack.c: UDP decapsulation

```
int
tcp_hijack_input(struct mbuf **mp, int *offp, int proto)
{
  int error = 0;
  struct mbuf *m = *mp;
  caddr_t ip;      /* We want to do ptr arithmetic by
         * the byte with these */
  caddr_t ip_orig;
  struct ip *ip_hdr;
  struct tcphdr *th;
  int inflated;
  int off = *offp;  /* This will be set to the IP hdr
         * length */
  uint8_t flowid;
  int rewritten;
  struct tcp_tiumap *map = NULL;
  struct tcp_tiumtag *tag;
  struct inpcb *inp = NULL;

  /* Relocate IP header, overwriting UDP header */

  ip_orig = mtod(m, caddr_t);

  ip = (ip_orig + sizeof(struct udphdr));
  CTR2(KTR_TCPINUDP, "ip=%p src=%p\n", ip, ip_orig);
  memmove(ip, ip_orig, off);
  m->m_data = ip;
  m->m_len -= sizeof(struct udphdr);
  m->m_pkthdr.len -= sizeof(struct udphdr);
```

First, the UDP header is removed.

tcp_hijack.c: rewrite back to TCP format

```
  /* Inflate the TCP header to be as normal */
  inflated = tcp_hijack_thinfl(mp, off, &flowid, &
     rewritten);
  if (inflated == -1) {
    /* mbuf will have been freed in _thinfl */
    CTR0(KTR_TCPINUDP,
         "tcp_hijack_input: inflater could not rewrite
             header;abort\n----\n");
    error = IPPROTO_DONE;
    *mp = NULL;

    return (error);
  }
```

The `tcp_hijack_thinfl` routine does the hard work of rewriting the header. If it encountered invalid input, it will have freed the buffer and we can return.

Reverse rewrite We now jump into the actual (reverse) rewrite:

tcp_hijack.c: prepare buffer

```
static __inline int
tcp_hijack_thinfl(struct mbuf **mp, int off, uint8_t *flowid, int *rewritten)
{
  struct tcphdr *th;
  struct mbuf *m = *mp;
  caddr_t ip;
  int tcplen, expectedsize;
  uint8_t swapb[4];
  ip = mtod(m, caddr_t);

  /* TCP header might be in next mbuf... */
  if (m->m_len < off + 1) {
    /* Check if there is actually another (filled) mbuf */
    if (m->m_next == NULL || m->m_len < 1)
      return -1;
    th = (struct tcphdr *)(m->m_next->m_data);
  } else
    th = (struct tcphdr *)(ip + off);

  if (TCPHDR_REWRITTENOFF(th) == TCPINUDP_SETUPSEGMENT) {
    *rewritten = 0;
    expectedsize = sizeof(struct tcphdr);
  } else {
    *rewritten = 1;
    /* -8 because we compress by that much (TODO get rid of magic number!) */
    expectedsize = sizeof(struct tcphdr) - 8;
  }
  /*
   * Pull up TCP header, which might be in the next mbuf.
   * This will save an m_pullup in tcp_input when that is the
   * case.
   */
  if ((m = m_pullup(m, expectedsize + off)) == NULL) {
    /* Ouch, this was a bad packet. m_pullup freed m for us. */
    return -1;
  }

  /* In case there was a realloc in m_pullup we have to do this: */
  *mp = m;
  ip = mtod(m, caddr_t);
```

```
        th = (struct tcphdr *)(ip + off);

  /* length of TCP segment present in first mbuf */
  tcplen = m->m_len - off;
```

Locate header offsets within the datagram. The buffer is adjusted, if necessary, so that the entire TCP fixed TCP header is located within the first mbuf. This might have to be repeated if we are unlucky and TCP options end up split between two mbufs, but it is not possible to know that without beginning to parse the TCP header.

tcp_hijack.c: rewrite header: common case

```
  if (*rewritten) {
    /* Move TCP opts + anything after down 8 bytes */
    if (m->m_len > (expectedsize + off)) {
      bcopy(TCPHDR_OFF(th), th + 1, tcplen - sizeof(
          struct tcphdr) + 8);
    }
    memset(&(th->th_sum), 0, 4); /* Zero out checksum  +
        urgentptr */
    m->m_len += 8;  /* Length of inflated fields */
    m->m_pkthdr.len += 8;

    /* Put offset and friends back where TCP expects them
        */
    bcopy(&th->th_sport, TCPHDR_OFF(th), sizeof(swapb));

    *flowid = th->th_x2 << 1;
    if (th->th_flags & TH_URG) {
      *flowid +=1;
      th->th_flags &= ~TH_URG;
    }

    if (*flowid > TCPINUDP_MAXFLOWID) {
      m_freem(m);
      *mp = NULL;
      return -1;
    }
```

Rewrite the non-SYN/SYN-ACK header format into regular TCP format.

tcp_hijack.c: rewrite header: SYN/SYN-ACK

```
  } else {
    /* Swap ports and offset etc */
    bcopy(TCPHDR_OFF(th), swapb, sizeof(swapb));
    bcopy(&th->th_sport, TCPHDR_OFF(th), sizeof(swapb));
    bcopy(swapb, &th->th_sport, sizeof(swapb));

    /* Recalculate what offset should be */
    /*
     * NB! Uses length of *whole* mbuf chain, we assume
     * there won't be any data anyway when using the
     * magic offset
     */
    th->th_off = (m_length(m, NULL) - off) / 4;
  }
  return (*rewritten ? 8 : 0);
}
```

If this was a SYN or SYN-ACK, just swap the reordered fields back around. Because we overwrite the real offset value with a special value (4), we must infer what the offset really should be. As the comment implies, we assume that the entire datagram is header information; when we generate such segments, that is true. If someone else were to generate them differently, any discrepancies will be caught be input validation code in the main TCP input code path. Finally, let the caller know if this was a setup or common segment.

Demultiplexing At this point, we have a partially reconstituted TCP segment. The next big job is to consult the flow id. to port pair map in order to insert the correct port numbers.

tcp_hijack.c: demultiplex flow id. to port pair

```
  /* Fix IP total length field to reflect the removed UDP
      header: */
  /* old m might no longer be valid after decompression
      */
  m = *mp;
  ip_hdr = mtod(m, struct ip *);
  ip_hdr->ip_len = htons(ntohs(ip_hdr->ip_len) - sizeof(
      struct udphdr) + inflated);
  ip_hdr->ip_p = IPPROTO_TCP; /* Or else *_respond get
      confused */

  if (rewritten) {
    /* Set ports based on flowid lookup */
    TCP_TIUMAPINFO_RLOCK(V_tcp_tiumapinfo);
    map = tcp_tiumap_lookup(ip_hdr->ip_src, ip_hdr->
        ip_dst);
    TCP_TIUMAPINFO_RUNLOCK(V_tcp_tiumapinfo);

    if (map == NULL) {
      m_freem(m);
      *mp = NULL;
      return IPPROTO_DONE; /* Is this right? */
    }

    th = (struct tcphdr *)(ip_hdr + 1);

    TCP_TIUMAP_RLOCK(map);
    if (map->tum_flow[flowid].fport == 0) {
      TCP_TIUMAP_RUNLOCK(map);
      m_freem(m);
      *mp = NULL;
      return IPPROTO_DONE;
    }
    th->th_sport = map->tum_flow[flowid].fport;
    th->th_dport = map->tum_flow[flowid].lport;
    if (map->tum_flow[flowid].tp != NULL)
      inp = map->tum_flow[flowid].tp->t_inpcb;
    TCP_TIUMAP_RUNLOCK(map);
  }
```

First of all, the length and protocol type fields of the IP header are adjusted. Then, if this is not a setup segment (recall that those actually carry full port numbers), locate the correct mapping table for the given pair of source and destination addresses. The directory of such maps is protected by a "master lock" called `mapinfo`. Notice how we are using reader/writer locks here; this locking mechanism allows several threads to gain simultaneous read-only access to shared resources. If a mapping table was located, it is consulted and if there is a match for the flow id. of the segment, the corresponding port pair is written into the header. Each per-host-pair map is protected by a separate lock, reducing lock contention between threads servicing packets coming from different associations.

tcp_hijack.c: final adjustments

```
  /*
   * Use the hardware-offloaded checksumming flags to convince
   * TCP that the checksum was okay.
   */
  m->m_pkthdr.csum_flags |= (CSUM_DATA_VALID | CSUM_PSEUDO_HDR);
  m->m_pkthdr.csum_data = 0xffff; /* Magic value */

  tag = (struct tcp_tiumtag *)m_tag_alloc(TCP_TIUMTAG_COOKIE, TCP_TIUMTAG,
          sizeof(struct tcp_tiumtag) -
          sizeof(struct m_tag), M_NOWAIT);
  /*
   * We need tags to work out states on conn. establishment,
   * drop if we can't tag
   */
  if (tag != NULL) {
    tag->tiu_map = map;
    m_tag_prepend(m, (struct m_tag *)tag);
  } else {
    return IPPROTO_DONE;
  }
```

Before passing the now reconstructed segment along to TCP, two things need to be done: the checksum must be reset in such a manner that TCP will accept it, and we need to append a so-called tag to the mbuf to be able to easily identify this segment as having been encapsulated, for reasons explained above. It would be possible to simply recompute the checksum of the packet after modifying it, but that would be wasteful – after all, UDP (and IP) already verified checksums. The solution is to set a special value that will always pass the checksum test, along with a flag used by hardware checksum offloading to indicate the checksum already was checked.

tcp_hijack.c: hook SIFTR and pass up to TCP

```
  if (!PFIL_HOOKED(&V_tiu_pfil_hook))
    goto passup;

  /* SIFTR needs a read lock: */
  if (inp != NULL)
    INP_RLOCK(inp);
  if (pfil_run_hooks(&V_tiu_pfil_hook, &m, NULL, PFIL_IN,
      inp)) {
    return IPPROTO_DONE;
  }
  if (inp != NULL)
    INP_RUNLOCK(inp);

 passup:
  error = tcp_input(mp, &off, IPPROTO_TCP);

  return (error);
}
```

The last action done by the decapsulation routine is to optionally send the segment through SIFTR, if it is enabled. Note that in this case, we acquire a lock on the Internet Protocol Control Block (which is a superset of the TCB, containing state information for IP as well), which we already held in the equivalent section of the encapsulation routine. SIFTR obviously needs such a lock, since it will inspect internal connection state. Finally, the segment is passed into the main TCP input processing function, `tcp_input`.

5.2.5 Connection teardown

The TIMEWAIT state

After a TCP connection closes, the connection is put into a special TIMEWAIT state to make sure that the other end really has understood that the connection is closed. In FreeBSD, connection state is actually modified quite heavily during this state transition; the normal TCB is transferred into a special structure for this state, which conserves memory. In order to properly clean up after TiU connections, we must make sure TiU state is included in that remnant data structure.

tcp_timewait.c: Save TiU state

```
  tw->tw_flags = tp->t_hijackflags; /* TCPinUDP state */
```

In `tcp_twstart`, we save the TiU state.

tcp_timewait.c: Divert responses in TW into the tunnel

```
    if (tw->tw_flags & TCP_HIJACK_ENABLED)
      error = tcp_hijack_output(m, inp->inp_options, NULL
          ,
              ((tw->tw_so_options & SO_DONTROUTE) ?
               IP_ROUTETOIF : 0),
              NULL, inp);
    else
```

Sometimes the other end in fact *did not* get the message that we wanted to close, and will send more messages that require a response. This piece of code in `tcp_twrespond` makes sure that if the connection ran over TiU, then those responses are sent through the TiU tunnel.

5.2.6 Housekeeping routines

The `tcp_hijack.c` file contains a few hundred source lines of code of additional routines that perform various "housekeeping duties" like maintaining the mapping tables and so on. We opt not to detail these here, as doing so would not add much to the understanding of the implementation.

There is also code that deals with setting and getting the TiU socket option in `tcp_usrreq.c`.

5.2.7 Implementation particularities

Optimizations

We will detail a few important optimizations that we made in order to ensure TiU performance is acceptable.

Checksum calculations Per-byte overheads have long been known to dominate the total TCP processing overhead, as shown originally by Clark et al. [20]. While memory copying operations make up the bulk of this category, the data checksumming operation is a significant contributor to overhead. In a more recent study, Chase et al. [18] reaffirm this by showing that checksum offloading to hardware yields a significant performance boost.

Therefore our implementation must take care to handle checksumming efficiently and avoid redundancy, both with respect to computation and header space occupied in the generated packets.

Only the TCP pseudo-header checksum will be calculated for segments going out over the tunnel. This is because FreeBSD defers calculating the full packet checksums until packets are about to be handed down from the IP layer to the link layer. Indeed, if there is hardware checksumming support, it is further deferred until it hits the NIC.

In our case, the actual checksumming will be done on the UDP datagram, both on the send and receive paths. It will possibly be calculated in hardware if the NIC supports UDP checksum offloading. Since our shim layer does not receive an incoming tunneled datagram if the checksum does not match, we set flags usually employed by hardware TCP checksum offload to signal to TCP that it does not need to verify the checksum any further.

Buffer copying As noted in the previous section, the largest overhead in TCP processing is caused by copying data between memory buffers. The conventional BSD TCP implementation will perform at least three copying operations: first, the data is copied from the application buffer in userland to the socket buffer in kernel memory. This is mainly done to provide isolation between userspace and kernel memory for memory protection reasons. Later, TCP segments the data by copying it from the socket buffer into a temporary `mbuf`chain in which the segment, along with the encapsulating IP packet and link layer frame, will be constructed within. Finally, the NIC device driver copies the end result into the NIC's on-board hardware buffer.

In order not to introduce any unnecessary additional copying-related overhead, TCP-in-UDP is carefully implemented to carry out as few copy operations as possible.

On the sending path, we enforce our own header splitting by never allowing payload data to be copied into the first mbuf, see section 5.2.4.

When rewriting the TCP header, the rearranging of field ordering can be achieved by block copy instead of more expensive modifications to single fields.

On the receive path it is harder to avoid copying operations. The encapsulated TCP header will be treated as part of the payload by the lower layers of the network stack, yet we must ensure it is located in its entirety in the first mbuf before handing it on, as TCP expects this. Apart from this, we use block copying and zeroing to rewrite much of the header back to its normal form as efficiently as possible.

5.3 Coupled Congestion Control

5.3.1 Overview

The coupled congestion control implementation could be separated entirely from the TCP-in-UDP encapsulation. It would work fine without the tunneling, as long as no multi-path routing takes places. However, both because multi-path routing has become so common in practice, and to simply the coupling decision logic, we have decided to directly tie our current implementation with TiU.

Similarly to the implementation of TiU, we have sought to implement the FSE as non-intrusively as possible. To that effect, we leveraged the modular congestion control framework [40] introduced in FreeBSD 9.

In this prototype, the FSE is only available as a pluggable congestion control module, which is a modified version of the "New Reno" module. Again, this is in part due to the fact we were contemplating FSE designs that were tightly integrated with the internal workings of the congestion controller, as we previously discussed in section 4.3. The passive FSE design we opted for in the end could probably lend itself to generalization to almost any congestion control algorithm, however.

5.3.2 Flow State Exchange

The actual Flow State Exchange itself is implemented in the file `cc_fse.c`. We will now present the most important parts of that implementation.

Registering

The function `cc_fse_register` implements algorithm 1 on page 51.

cc_fse.c: register flow

```
struct fse_flow*
cc_fse_register(struct inpcb *inp)
{
  struct fse_flow *flow;
  struct tcpcb *tcb;

  flow = malloc(sizeof(struct fse_flow), M_FSE, M_NOWAIT|
      M_ZERO);

  if (flow == NULL)
    goto out;

  tcb = intotcpcb(inp);

  flow->f_p = tcb->t_fse_prio;
  flow->f_inp = inp;

  FSE_STATE_WLOCK(V_cc_fse_state);
  flow->f_fi = V_cc_fse_state.fse_next_fid++;

  /* The rest of the flow data MUST be set BEFORE
      grouping! */
  cc_fse_group(flow);
  CTR6(KTR_FSE, "cc_fse_register: new flow %u (%p), tcb=%
      p, fg=%p (%u/%u flows)",
        flow->f_fi, flow, tcb, flow->f_fg, flow->f_fg->
            fg_num_coupled,
        flow->f_fg->fg_membercount);
  FSE_STATE_UNLOCK(V_cc_fse_state);
 out:
  return flow;
}
```

First, storage is allocated to store flow-specific state information. State information is copied from the TCB, before a call to `cc_fse_group` takes care of joining the flow to the appropriate flow group:

cc_fse.c: find or create flow group

```
static struct fse_fg*
cc_fse_group(struct fse_flow *f)
{
  struct fse_fg *fg;
  struct fse_fg_head *fgt_head;
  struct tcpcb *tp;

  FSE_STATE_WLOCK_ASSERT(V_cc_fse_state);

  INP_LOCK_ASSERT(f->f_inp);
  tp = intotcpcb(f->f_inp);
  fg = cc_fse_lookup_addrs(f->f_inp->inp_faddr, f->f_inp
     ->inp_laddr);
  if (fg == NULL) {
    fg = malloc(sizeof(struct fse_fg), M_FSE, M_NOWAIT|
       M_ZERO);

    if (fg == NULL) {
      return NULL;
    }

    rw_init_flags(&fg->fg_memberslock, "FSE FG lock",
       RW_RECURSE);
    LIST_INIT(&fg->fg_members);

    fgt_head = &V_cc_fse_state.fse_fgtable[ \
      FSE_FG_ADDRHASH(f->f_inp->inp_faddr.s_addr,
          f->f_inp->inp_laddr.s_addr,
          V_cc_fse_state.fse_fgtable_mask)];
    LIST_INSERT_HEAD(fgt_head, fg, fg_fgte);
  }
```

A flow group (FG) is located by host-address-pair, and if none is found a new one is instantiated.

cc_fse.c: join flow to flow group

```
  FSE_FG_MBR_WLOCK(fg);
  LIST_INSERT_HEAD(&fg->fg_members, f, f_fge);
  fg->fg_membercount += 1;
  f->f_fg = fg;

  if (tp->t_hijackflags & TCP_HIJACK_ENABLED)
    cc_fse_couple(f);

  getmicrouptime(&fg->fg_lastupdate);
  FSE_FG_MBR_UNLOCK(fg);

  return fg;
}
```

The joining flow is added to the FG's member list. We do not actually add the flow's congestion control state to the aggregate before making a coupling decision on line 259. This decision will be reconsidered at every update the flow makes.

This is a slight deviation from the algorithm as we presented it in algorithm 1 on page 51, but it makes management of FGs much simpler in practice to always register flows with them for the entire lifetime of every flow.

Deregistering

When a flow ends, it makes a call to `cc_fse_deregister`:

cc_fse.c: deregister flow

```
297 void
298 cc_fse_deregister(struct fse_flow *f)
299 {
300   struct fse_fg *fg;
301
302   fg = f->f_fg;
303
304   FSE_FG_MBR_WLOCK(fg);
305   LIST_REMOVE(f, f_fge);
306   fg->fg_membercount -= 1;
307   cc_fse_decouple(f);
308   getmicrouptime(&fg->fg_lastupdate);
314   FSE_FG_MBR_UNLOCK(fg);
315
316   free(f, M_FSE);
317 }
```

Flows always contain a pointer to their containing FG, so decoupling and removing them is easy.

Coupling and decoupling

cc_fse.c: couple flow

```
131 static void
132 cc_fse_couple(struct fse_flow *f)
133 {
134   struct fse_fg *fg;
135   struct tcpcb *tcb;
136
137   tcb = intotcpcb(f->f_inp);
138
139   fg = f->f_fg;
140   FSE_FG_MBR_WLOCK_ASSERT(fg);
141
142   fg->fg_s_p += f->f_p;
143   fg->fg_num_coupled += 1;
144
145   if (fg->fg_num_coupled == 1) {
146     /*
```

```
      * FG inherits cwnd/ssthresh of first flow coupled to
      * it.
      * If an FG went stale, but is revived, we reuse the
      * old params. This is temporal sharing.
      */
     if (fg->fg_s_cwnd == 0) {
       fg->fg_s_cwnd = tcb->snd_cwnd;
       fg->fg_s_ssthresh = 0;
     }
     fg->fg_coco = f;
   }
```

When the decision has been made to couple a flow with the ensemble, its priority weight is added to the ensemble total. Before integrating the congestion window and slow-start threshold, we must check if this is the first ever flow to be coupled in this FG; in that case, the FG is bootstrapped as specified in algorithm 1 on page 51. If the flow is becoming the only coupled flow, but is not the *first*, it will actually have its state overridden by what is stored in the FG – this is how we achieve temporal sharing. Either way, a lone flow must flow become CoCo.

cc_fse.c: allocate cc state

```
  f->f_cwnd = f->f_p * fg->fg_s_cwnd / fg->fg_s_p;
  f->f_ssthresh = tcb->snd_ssthresh;

  if (fg->fg_s_ssthresh > 0)
    f->f_ssthresh = f->f_p * fg->fg_s_ssthresh / fg->
        fg_s_p;

  cc_fse_writeback(f);
  f->f_state = FSE_FLOW_COUPLED;
}
```

Now we can be sure that the FG contains valid congestion control state. Calculate the flow's appropriate share, and write it back to its TCB by calling `cc_fse_writeback`.

Decoupling is mostly the same in reverse, the other big task here is to elect a new CoCo if the current one is leaving:

cc_fse.c: decouple flow

```
static void
cc_fse_decouple(struct fse_flow *f)
{
  struct fse_fg *fg;
  struct fse_flow *fg_f, *candidate;
  struct tcpcb *tp;

  if (!(f->f_state & FSE_FLOW_COUPLED))
    return;

  fg = f->f_fg;
  FSE_FG_MBR_WLOCK_ASSERT(fg);

  fg->fg_s_p -= f->f_p;
  fg->fg_num_coupled -= 1;

  if (fg->fg_coco == f) {
    /*
     * Select new CoCo:
     * Pick the first flow in CA we find. If there are
     * none, we prefer a flow in FR over one in SS.
     */
    candidate = NULL;
    LIST_FOREACH(fg_f, &fg->fg_members, f_fge) {
      if (fg_f->f_state & FSE_FLOW_UNCOUPLED)
        continue;

      tp = intotcpcb(fg_f->f_inp);
      if (candidate != NULL && (fg_f->f_state &
          FSE_FLOW_WANTED_SS)) {
        /* Skip SS flows except if we have
         * nothing better */
        continue;
      }

      candidate = fg_f;

      /* Flow in CA; it will do! */
      if (!(fg_f->f_state & FSE_FLOW_IN_FR))
        break;
    }

    fg->fg_coco = candidate;
  }

  f->f_state = FSE_FLOW_UNCOUPLED;
}
```

Congestion control state updates

Every time an individual flow's congestion controller makes some kind of update to the flow's congestion control state, we make sure that the `cc_fse_update` function gets hooked afterwards.

cc_fse.c: read state out of TCB and reconsider coupling

```
void
cc_fse_update(struct fse_flow *f, uint32_t flags)
{
  struct tcpcb *tcb;
  struct fse_fg *fg;
  struct fse_flow *fg_f, *new_coco;
  u_long cc_cwnd, cc_ssthresh;

  KASSERT(f != NULL, ("updating a NULL flow"));
  INP_WLOCK_ASSERT(f->f_inp);

  tcb = intotcpcb(f->f_inp);
  fg = f->f_fg;
  cc_cwnd = tcb->snd_cwnd;
  cc_ssthresh = tcb->snd_ssthresh;
  FSE_FG_MBR_WLOCK(fg);
  if ((f->f_state & FSE_FLOW_UNCOUPLED) &&
      tcb->t_hijackflags & TCP_HIJACK_ENABLED) {
    cc_fse_couple(f);
  } else if ((f->f_state & FSE_FLOW_COUPLED) &&
        !(tcb->t_hijackflags & TCP_HIJACK_ENABLED)) {
    cc_fse_decouple(f);
  }
```

First, this function reads what state the native congestion controller set, and checks if anything has changed which would require either coupling or decoupling. In reality this will only happen during the handshake, since the TiU implementation does not allow live migration between TiU and regular TCP at other times.

cc_fse.c: check for special cc states

```
  if (f->f_state & FSE_FLOW_COUPLED) {

    /* Check if CC algo tried to go into SS */
    if (cc_cwnd < cc_ssthresh)
      f->f_state |= FSE_FLOW_WANTED_SS;
    else
      f->f_state &= ~FSE_FLOW_WANTED_SS;

    /* Check if flow is currently in FR */
    if (IN_RECOVERY(tcb->t_flags))
      f->f_state |= FSE_FLOW_IN_FR;
    else
      f->f_state &= ~FSE_FLOW_IN_FR;
```

The rest of routine implements algorithm 4 on page 55 and is only executed for coupled flows. First, we check if the flow was in some special state (i.e. slow-start or fast recovery).

cc_fse.c: non-CoCo updates

```
    if (fg->fg_coco != f) {
      if (IN_FASTRECOVERY(tcb->t_flags)) {
        /* Search for any other flows not in CA */
        int non_ca = 0;
        LIST_FOREACH(fg_f, &fg->fg_members, f_fge) {
          if (fg_f->f_state & FSE_FLOW_IN_FR) {
            non_ca = 1;
            break;
          }
        }
        /* Everyone else in CA, become CoCo */
        if (!non_ca) {
          fg->fg_coco = f;
        }
      } else {
        f->f_cwnd = f->f_p * fg->fg_s_cwnd / fg->fg_s_p;
        if (fg->fg_s_ssthresh > 0)
          f->f_ssthresh = f->f_p * fg->fg_s_ssthresh / fg
              ->fg_s_p;

        cc_fse_writeback(f);
        goto out;
      }
    }
```

The first part of the algorithm deals with updates coming from flows which are not currently CoCo. The implementation matches the design closely.

cc_fse.c: CoCo update: CA or FR state

```
    /* CoCo might have changed above */
    if (fg->fg_coco == f) {
      if (cc_cwnd > cc_ssthresh && !IN_RECOVERY(tcb->
          t_flags)) {
        /* Normal CA update */
        if (cc_cwnd >= f->f_cwnd) /* AI... */
          fg->fg_s_cwnd += cc_cwnd - f->f_cwnd;
        else /* ..MD */
          /* XXX Operand order significant! */
          fg->fg_s_cwnd = fg->fg_s_cwnd * cc_cwnd / f->
              f_cwnd;

        f->f_cwnd = f->f_p * fg->fg_s_cwnd / fg->fg_s_p;
        f->f_ssthresh = cc_ssthresh;
        if (fg->fg_s_ssthresh > 0)
          f->f_ssthresh = f->f_p * fg->fg_s_ssthresh / fg
              ->fg_s_p;

        cc_fse_writeback(f);
      } else if (IN_FASTRECOVERY(tcb->t_flags)) {
        fg->fg_s_ssthresh = fg->fg_s_cwnd / 2;
```

This part of the algorithm takes updates from a CoCo (which may have been reassigned above). The formula for calculating the correct share of the window is expressed differently than in the formal algorithm because we only use integer arithmetic. It is technically possible to use floating point operations within the kernel, but it is heavily discouraged as doing so is complicated. It also hurts performance.

cc_fse.c: slow-start handling

```
      } else if (f->f_state & FSE_FLOW_WANTED_SS) {
        /* Try to find a new CoCo not in SS */
        new_coco = NULL;
        LIST_FOREACH(fg_f, &fg->fg_members, f_fge) {
          if (!(fg_f->f_state & FSE_FLOW_WANTED_SS)) {
            new_coco = fg_f;
            break;
          }
        }
        if (new_coco == NULL) {
          /* Everyone in SS! */
          fg->fg_s_ssthresh = fg->fg_s_cwnd / 2;
          fg->fg_s_cwnd = fg->fg_s_cwnd * cc_cwnd / f->f_cwnd;
          f->f_cwnd = f->f_p * fg->fg_s_cwnd / fg->fg_s_p;
          tcb->snd_cwnd = max(f->f_cwnd, tcb->t_maxseg);
        } else {
          fg->fg_coco = new_coco;
          cc_fse_writeback(f);
        }
      }
    }
  }

 out:
  getmicrouptime(&fg->fg_lastupdate);
  FSE_FG_MBR_UNLOCK(fg);
}
```

The most complicated part of the algorithm is the appropriate response to a CoCo that has been sent into slow-start. We must first walk the FG's flow list to check if all flows have been experiencing the same thing, then either enact slow-start if that is the case, or elect a new coco if not.

Writing back congestion control state

When the FSE has computed new congestion control state values for a flow, they are written back into the flow's TCB by `cc_fse_writeback`.

cc_fse.c: write back values to TCB

```
static inline void
cc_fse_writeback(struct fse_flow *f)
{
  struct tcpcb *tcb;

  tcb = intotcpcb(f->f_inp);
  tcb->snd_cwnd = max(f->f_cwnd, tcb->t_maxseg);
  tcb->snd_ssthresh = max(f->f_ssthresh, tcb->t_maxseg *
      2);
}
```

Values are clamped so as not to risk stalling individual connections.

5.3.3 Setting the priority weight

The FSE priority weight can be set to some value other than the default (which is the middle of the range) using a socket option. This is handled in `tcp_ctlinput`.

tcp_ctlinput.c: set FSE priority weight

```
    case TCP_FSEPRIO:
      INP_WUNLOCK(inp);

      error = sooptcopyin(sopt, &optval, sizeof optval,
              sizeof optval);
      if (error)
        return (error);

      /*
       * 8 bit value, for now restrict to multiples
       * of 2
       */
      if (optval < 1 || optval > 255 || optval % 2)
        return EINVAL;

      INP_WLOCK_RECHECK(inp);
      tp->t_fse_prio = optval;
      goto unlock_and_done;
```

The weight value is read, checked for validity and set. The range is restricted to what can be stored in a single byte, and we also require it to be a multiple of two. This leads to fewer rounding errors.

tcp_ctlinput.c: read FSE priority weight

```
    case TCP_FSEPRIO:
      optval = tp->t_fse_prio;
      INP_WUNLOCK(inp);
      error = sooptcopyout(sopt, &optval, sizeof optval);
      break;
```

This snippet allows the current FSE priority weight value to be read in userspace.

5.3.4 Implementation particularities

Hooks in "New Reno" module

Our modified version of the "New Reno" congestion control module hooks `cc_fse_update` each time it has changed `cwnd` or `ssthresh`. The congestion control module also takes care of registering and deregistering flows as they are created and ended. We will not detail the implementation here because it consists of trivial changes to the original module.

Main congestion control framework changes

As the FSE needs to infer what each flow's own congestion controller has done, it is imperative that the call to `cc_fse_update` comes after all changes have been made.

In order to satisfy this requirement, we were forced to reorder some calls within the main congestion control framework functions (`cc_*`) in `tcp_input.c`. We made sure this did not have other side-effects and remained functionally equivalent.

Incremental updates

All calculations performed during updates are incremental, without the need for costly looping in order to update the aggregate total state variables.

Concurrency

The passive FSE designs allows us to keep relying on each TCB's individual locks in addition to a per-FG lock held when updating FG state. Compared to attempts we made at updating state across TCBs directly, this makes for a dramatically simpler implementation that is far less prone to deadlocking bugs.

Chapter 6

Evaluation

In this chapter, we present the experiments we carried out in order to evaluate our TCP-in-UDP and Flow State Exchange implementations and analyse the results thereof. Section 6.1 describes the testbed environment we set up to perform measurements, including the synthetic cross-traffic we generated and the tools we used. The following section detail the setup and results of the different experiments we ran.

6.1 Testbed setup

In order to reliably evaluate the performance of the TCP-in-UDP encapsulation and FSE, we ran tests on a small hardware testbed. All endpoints running the modified code were separate physical machines. This way, we can have confidence that the results are not influenced by timing issues which one might expect from an entirely virtualised setup.

In order to emulate realistic network conditions, such as reasonable transmission delay, we integrated the CORE [1] [1] network emulator into the testbed network. Using CORE, we can easily manage a virtual network and set various parameters. It would also be possible to emulate a much more complex network than would be practical to set up in the lab, although this was not necessary for our purposes. Using a traffic generator such as D-ITG [16], we can also inject realistic background traffic into the network at this point.

The physical machines are all identical desktop computers (Intel i7-870 2.93GHz CPU, 8GB RAM) equipped with Gigabit Ethernet Network

[1]See http://www.nrl.navy.mil/itd/ncs/products/core.

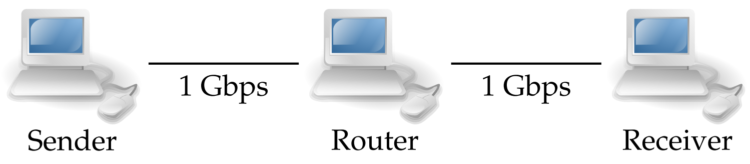


Figure 6.1: Evaluation testbed physical setup

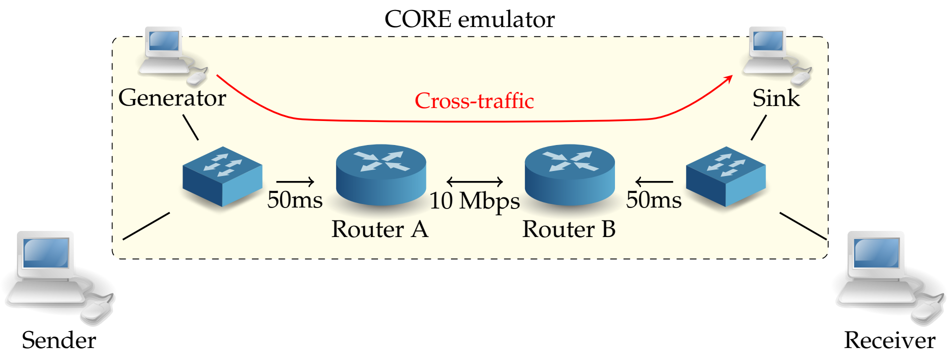


Figure 6.2: Emulated testbed topology

Interface Cards (NICs). These computers were arranged in a chain topology, with the CORE emulator machine linking the two endpoints, see figure 6.1. The endpoint machines are running our modified version of FreeBSD, whilst the emulator machine (router) is running Ubuntu Linux[2] 15.04 and CORE 4.7.

All hardware acceleration features of all NICs were disabled save for checksum offloading, since these can interfere with both network emulation and measurements. Additionally, as discussed in section 4.4.3 our implementation is not compatible with such mechanisms anyway.

All tests were run on top of the hybrid virtual-physical dumbbell network topology depicted in figure 6.2. Unless otherwise mentioned, the RTT between end hosts was configured to be 100ms, the bottleneck link was restricted to a capacity of 10Mbps and its queue was dimensioned according to the bandwidth-delay product, yielding a queue size limit of about 83 MTU-sized packets in this case. On Linux, CORE implements its network emulation using a combination of the network namespaces and traffic control (tc) kernel features. To avoid previously reported issues (see e.g. [63]) concerning the combination of tc queueing disciplines related to the netem network emulation subsystem responsible for emulating delays and other disturbances, with those that provide traffic shaping (bandwidth limitation), we made sure to apply these treatments on different virtual links within the emulator.

6.1.1 Cross-traffic

To eliminate effects caused by synchronisation and obtain a more realistic behaviour, we inject cross-traffic into our testbed that was generated using the D-ITG [16] traffic generator.

To obtain cross-traffic with realistic Internet traffic behaviour, we have made use of heavy-tailed traffic distributions. Such distributions have been shown to model both Web [22] and more general wide-area network traffic [34] well.

[2]CORE advertises support for FreeBSD, however this does not seem to be maintained any longer - see also http://pf.itd.nrl.navy.mil/pipermail/core-users/2014-February/001472.html. Recent versions do not run out of the box under FreeBSD and were not trivial to fix.

The Hurst parameter of a Pareto distribution is given by

$$H = \frac{3-\alpha}{2} \tag{6.1}$$

where α is the shape parameter or tail index. Distributions with

$$0.5 < H < 1$$

are heavy-tailed, and

$$0.7 < H < 0.8$$

is usually recommended for traffic generation purposes [36, pp. 471–474].

In keeping with this, we configured D-ITG to generate a superposition of 11 bursty On/Off streams, with Pareto heavy-tailed on-time distributions ($H = 0.8$), and exponentially distributed off-times (mean ranging from 1 to 2 s). The packet sizes were normally distributed ($\mu = 1000$, $\sigma = 200$, in bytes), whilst the inter-departure times (i.e. during on-periods) were exponentially distributed (mean ranging from 50 to 150 pps).

We chose parameters and number of superposed streams so that cross-traffic occupied approximately half of the bottleneck link capacity, on average. All of the cross-traffic is inelastic, i.e. it does not react to congestion or loss in any way.

Because we had some difficulty in getting D-ITG to reliably generate replicatable traffic sequences, we opted to prerecord packet traces instead, which we split into 345s runs. These were played back across the bottleneck link using the tcpreplay tool[3] during measurement runs. Each run within a measurement series saw a different trace, but the corresponding runs in different configurations experienced the same trace.

6.1.2 Measurement methods

In order to measure the performance and behaviour of the flows in the emulated testbed, we used a combination of tools.

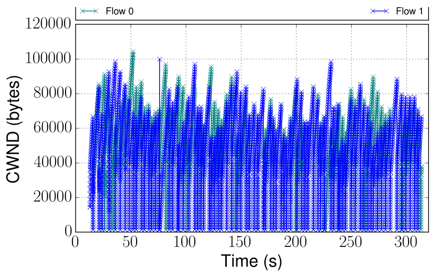
On the sender machine, we used the Statistical Information For TCP Research (SIFTR) FreeBSD kernel module [77] to observe the internal state and dynamics of the TCP implementation. SIFTR outputs the current values of an array of interesting TCP state variables every time a TCP segment is received or transmitted. We slightly modified SIFTR to allow it to place additional hooks in the appropriate places to pick up TiU-encapsulated segments.

For analysing on-the-wire behaviour, we captured packet traces using the `tshark` tool (the command-line version of the common Wireshark[4] capture software), which we later analysed using a combination of the `captcp` tool[5] and a suite of custom Python scripts we created. We modified

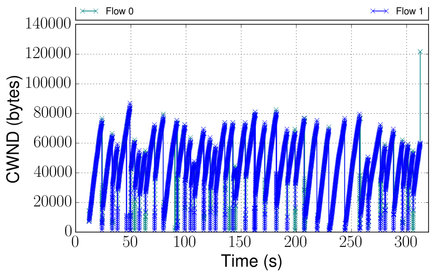
[3]See http://tcpreplay.synfin.net/.

[4]See http://www.wireshark.org.

[5]See http://research.protocollabs.com/captcp/.

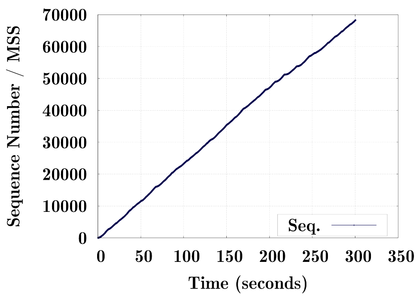


(a) Regular TCP

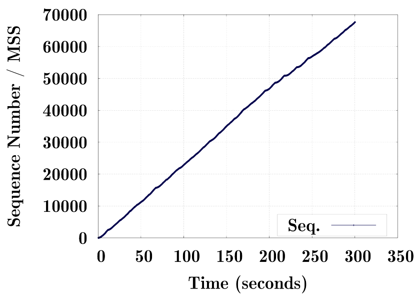


(b) FSE

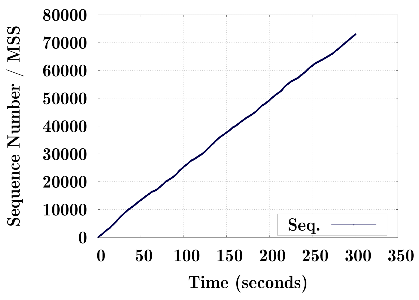
Figure 6.3: Congestion window size in bytes, 2 flows



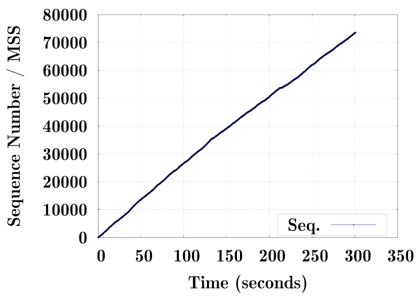
(a) FSE: flow 1



(b) FSE: flow 2



(c) Regular TCP: flow 1



(d) Regular TCP: flow 2

Figure 6.4: Time sequence diagrams

`captcp` to add support for parsing TiU headers and fixed some bugs along the way, see appendix A.

In order to observe the behaviour of the (virtual) bottleneck link, we used the standard `tc` Linux utility to output statistics about queue occupation and drop counts at regular intervals.

6.2 Experiments

6.2.1 Dynamic behaviour

We begin by taking a quick look the dynamic behaviour of both regular "New Reno" TCP and our FSE-enhanced "New Reno". Figure 6.3 graphs the congestion window sizes of two concurrent flows that run for 300s, with and without the FSE, experiencing the same cross-traffic. We measure the internal `cwnd` variable using SIFTR at the sender.

Unfortunately, these graphs are somewhat difficult to interpret due to a quirk of the FreeBSD TCP implementation. To prevent bursting in fast recovery, when changing the congestion window at the onset of that phase, FreeBSD triggers a "mini" slow-start by temporarily moving the congestion window lower than it is intended to be, rapidly (much faster than regular slow-start due to the fact that the starting point is higher) ramping up to the correct level by temporarily setting `ssthresh` to where `cwnd` should be. This is the reason the congestion window appears to go down to 1MSS (as after a timeout) repeatedly in figure 6.3. The authors of [5] also describe this phenomenon.

In the interest of illustrating the actual on-the-wire behaviour, we also plot time-sequence diagrams of the same flows in figure 6.4. These look less interesting, but are easier to understand.

From this collection of graphs, it is clear that with the FSE, we get smoother, more coordinated behaviour. The flows do not "fight" each other, although the negative repercussions of such behaviour are not very clear from this example of only two flows. The next experiment will give a clearer picture of that.

6.2.2 Quantitative performance test

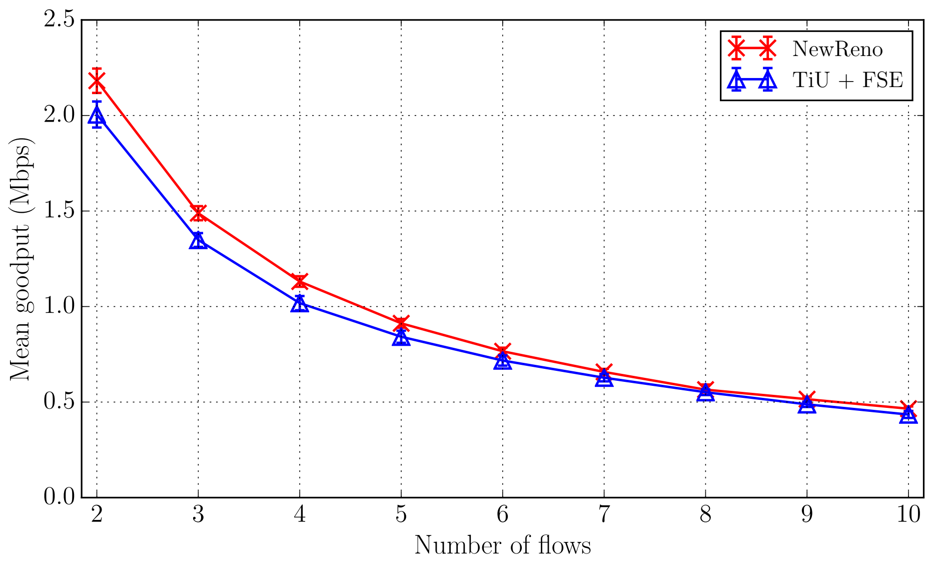


Figure 6.5: Mean goodput in megabits per second

To quantify the performance of our implementation of "New Reno" coupled using the FSE and compare it to that of standard TCP "New Reno" as implemented in FreeBSD, we ran a variable number of TCP connections in parallel and measured the following performance metrics:

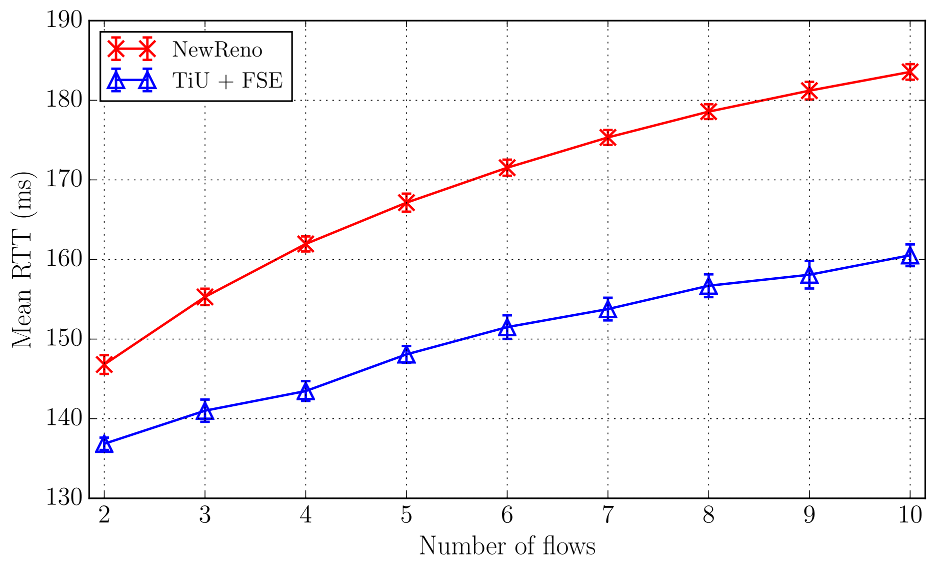


Figure 6.6: Mean delay in milliseconds

- Goodput, i.e. throughput corrected for retransmissions. Measured using the captcp tool, based on packet traces captured at the egress of the bottleneck link.
- Delay as RTT experienced by the TCP sender. Measured using SIFTR.
- Queue filling degree at the bottleneck link, in packets. Measured by periodic sampling at the bottleneck link.
- Loss ratio at the bottleneck link. Note that this includes cross-traffic packets being dropped as well as TCP packets. Measured by periodic sampling of drop count totals at the bottleneck link.

The number of flows was varied between 1 and 10 concurrent flows. The testbed was configured as described in section 6.1 and cross-traffic enabled. TCP flows were generated using the `iperf` benchmarking tool and ran for 300s each. All flows were started within the first second of the run. During analysis, data from this first second has been removed to correct for different start times and to cut away the transient period. The data plotted is based on measurement series of 10 runs each. We plot 95% confidence intervals as error bars on all graphs.

Goodput

The mean goodput per flow is plotted in figure 6.5. We see that goodput of flows part of the coupled ensemble only slightly trails that of the independently congestion controlled flows, and that the gap closes with an increasing number of concurrent flows. The fact that multiple TCP flows are more aggressive in aggregate than a single one is well documented [23, 24], so this is an expected result.

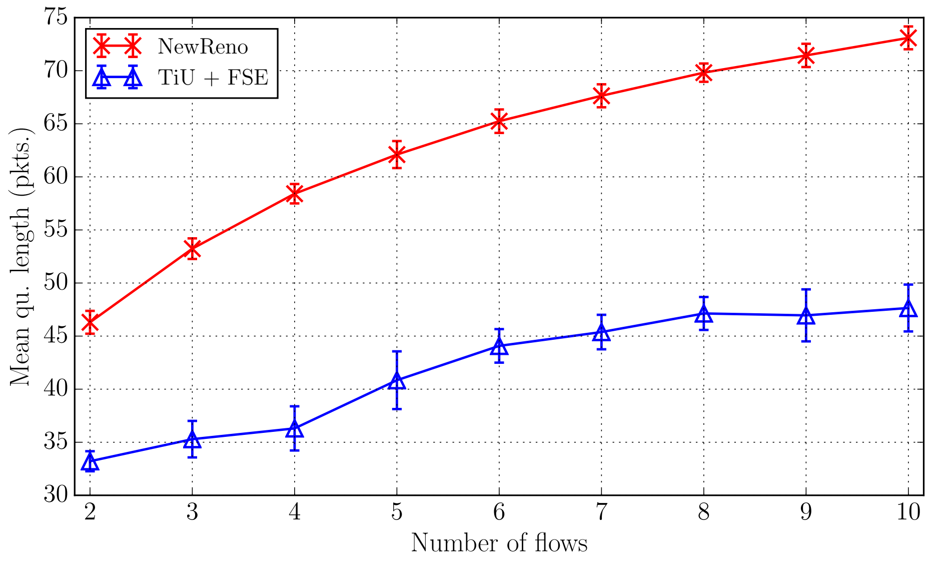


Figure 6.7: Mean queue length in packets

Delay and queue length

Figures 6.6 and 6.7 show the results for RTT and queue length, respectively. We can observe that the graphs reflect the same trend, as we would expect – aside from the 100ms propagation delay and quasi-negligible processing delays, queueing delay constitutes the remainder of the RTT.

These graphs demonstrate the benefits of tempering the aggressiveness of the ensemble of TCP flows. Delay is significantly reduced compared to regular "New Reno", and the difference grows with an increasing number of flows. The fact that the pressure on the queue is diminished also leads to fewer packet losses, which add a lot of delay to application layer delivery time due to head of line blocking as well as lowering throughput.

Loss ratio

This last statement is confirmed by the plot in figure 6.8. When running with standard TCP behaviour, the loss ratio increases nearly linearly with the number of parallel flows, whilst it only increases very moderately when enabling the FSE.

Overhead characterization

Runs with a single flow are not shown in the graphs because the behaviour is identical for both standard and FSE "New Reno". This largely validates that our implementation is efficient. During development, profiling (on an unoptimized binary) revealed that the processor only spent about 1% longer in network stack when TiU was enabled, and most of that time was spent in a "hot spot" which could be optimized further.

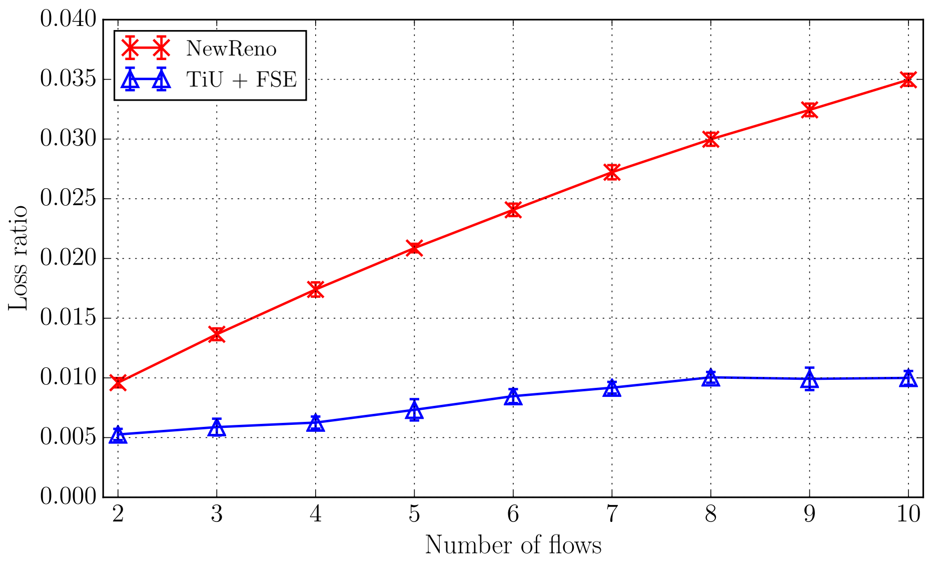


Figure 6.8: Mean loss ratio at bottleneck link

6.2.3 Prioritization

In order to demonstrate the weighted prioritization mechanism we built into the FSE, we ran streams at different prioritization ratios and compared the effect on the throughput ratio.

The testbed was configured as in section 6.1, but cross-traffic was disabled for this test.

We used `iperf` to generate two parallel 300s long flows, as in the previously described quantitative test. Our version of `iperf` has been modified to allow a configurable FSE priority weight to be assigned to each flow. We ran the experiment 10 times for each configuration of priority ratios.

Figure 6.9 shows the results. Please note that error bars are plotted for all points, but only the last two are easily visible. We can see that the measured throughput ratio nicely follows the ideal 1 : 1 relation between ratio of priority weights to ratio of throughput achieved, except for the lowest ratios. This increased variability at lower ratios can at least partially be attributed to the fact that the least prioritized stream will be allocated a very low share of the available bandwidth in those cases.

6.2.4 Transfer completion time

In this final test, we will demonstrate how the FSE could yield a clear user-felt benefit in a common use case.

Web pages usually consist of a number of objects that need to be requested separately from the Web server. Browsers usually do this in parallel to reduce the time it takes to finish rendering the web page, i.e.

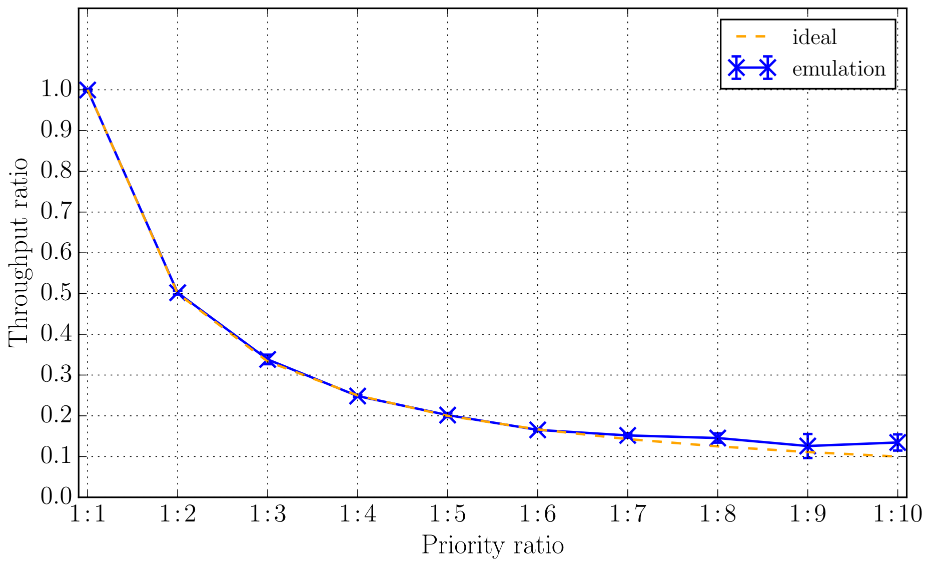


Figure 6.9: Throughput ratio versus priority ratio

page load time. Shorter page load times obviously make for a better user experience.

To simulate a situation somewhat similar to this, we will start two file-size-limited flows in a staggered manner. First, a long flow starts, downloading 2 megabytes of data. After 2 seconds, it is joined by a second parallel request, this time downloading 200 kilobytes.

Figure 6.10 shows the time to complete for both flows, with varying bottleneck link capacity and with the FSE both enabled and disabled. Otherwise, the testbed is configured as previously described. The data plotted is based on measurement series of 6 runs for each configuration.

When the FSE is enabled, we see that the short flow tends to complete faster than the same request without the FSE. At the same time, the long flow tends to take slightly longer to complete; this is expected, since it is having to give away some of its share of the bandwidth.

The difference is not very large, however. Analysing the underlying packet traces, we saw that the lack of pacing is causing some unfortunate micro-bursts both when the short flow joins the ensemble, and when it leaves again. This is in effect neutralizing much of the positive effect the FSE could have provided, underpinning our argument that a TCP congestion control coupling mechanism needs to be paired with a pacing mechanism in order to fully unlock its potential.

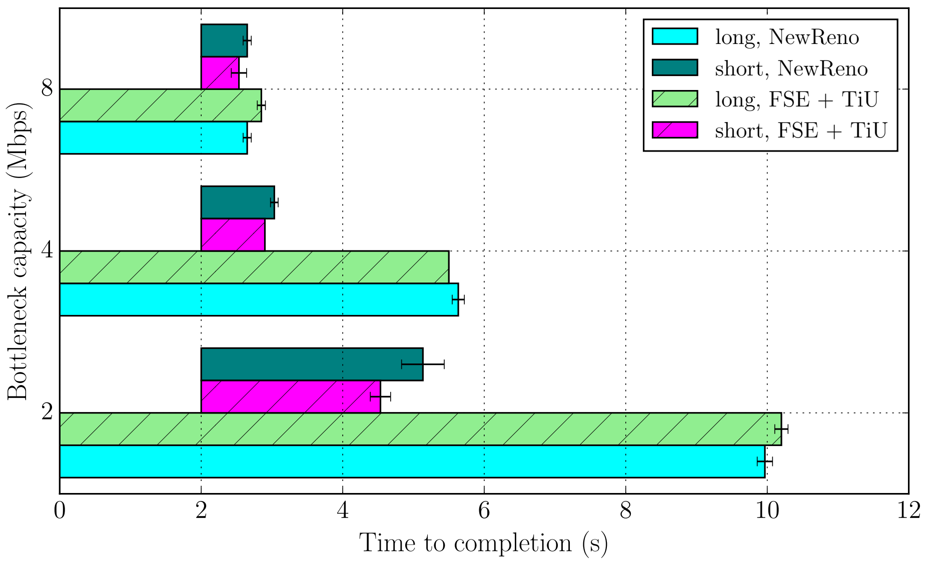


Figure 6.10: Flow's time to completion for one long and one short flow

Chapter 7

Conclusion

We will now briefly sum up how we have answered the research questions, explore some potential directions for further research, before finally concluding.

7.1 Research Findings

Here is a summary of the findings we made in answering the research questions posed in section 1.3.

Multi-path forwarding

> In chapter 3 we presented the TCP-in-UDP encapsulation method, which can successfully work around multi-path forwarding by ensuring all packets carry the same header five-tuple.

TCP-in-UDP

> This encapsulation method is designed in such a way as not to reduce Maximum Segment Size at all, and is suited to an efficient implementation.

TCP congestion control coupling

> The TCP passive Flow State Exchange presented in chapter 4 is a highly practical and useful mechanism for performing TCP congestion control coupling. We prove this by implementing it, as detailed in section 5.3.

Interface

> We deliver a flexible interface by simultaneously enabling our mechanism to be used without applications needing to have knowledge about it, while at the same time providing extensions to the socket options interface to meet the needs of applications which are actually

aware of it. System administrators have full control of how the mechanism is applied through a series of `sysctl` configuration options, with full support for FreeBSD virtualised network stacks.

Performance

The evaluation we have performed and presented in chapter 6 clearly demonstrates that TCP-in-UDP combined with the TCP passive FSE can deliver tangible benefits. We also show how the FSE prioritization mechanism provides new QoS options to applications.

In answering these research questions, we have shown how TCP congestion control coupling can be made deployable across the Internet, entirely by making changes at the end hosts, and that there are performance benefits to doing so.

7.2 Further work

7.2.1 Alternative encapsulation methods

While we have shown that our TCP-in-UDP encapsulation method works in a satisfactory manner, it does have one problem: the receiver side must be changed to be able to use it. Our solution remains gradually deployable, in that we need no additional cooperation from other elements within the network itself, but congestion control coupling is still restricted to being used between hosts that have both adopted TiU.

Some alternative methods for working around the multi-path issue do exist:

- The IPv6 flow label, which we described in section 2.2.1, can be set at the sender side only and *should* ensure that all flows with the same label are forwarded along the same path. Whether this really happens is of today unknown, further research could uncover if this is a viable option.
- There are many other kinds of tunneling approaches possible, and in common use today. As long as there are no multi-path network segments between the hosts and the tunnel endpoints, many kinds of Virtual Private Network (VPN) tunnels ought to have the same effect as TiU given that they operate over a single port-pair. The same can be said about Generic UDP Encapsulation (GUE) [41]. Classifying what existing tunneling protocols are suitable candidates for multi-path avoidance could be an interesting topic for further investigation.

7.2.2 Improved FSE algorithm

The FSE algorithm presented and implemented in this thesis is only meant as a proof of concept to demonstrate the potential that lies in applying coupled congestion control to TCP. Although we have addressed some of the weaknesses we found (see section 2.4.4) in previous algorithms we have studied, the algorithm could surely be improved and refined.

In particular, it would be highly beneficial to augment it with some kind of pacing mechanism to smooth out the manner in which flows consume large congestion window increases.

Another possible avenue for further research would be to make FSE controlled flows more aggressive in the face of competition from other TCP flows. The algorithm presented herein will be quite at a disadvantage if there are a large number of regular, independently congestion controlled TCP flows traversing the same bottleneck. In that case, it might be possible to use an algorithm similar to MulTCP [23] or MulTFRC [24] to "gear up" the ensemble behaviour to match that of some number of flows larger than one, but lower than the actual number of member flows. In that manner one could perhaps keep some of the advantages of a less aggressive behaviour, while still competing on a more even footing with the other regular TCP flows.

7.2.3 Pluggable TCP stacks

During the last months, the upstream FreeBSD TCP stack has seen some interesting changes. In particular, there is a project underway to modularize the TCP implementation [56] so that different implementations can coexist and be selected on the fly, on a per-connection basis. It could help a potential effort to drive adoption of the solutions we have developed to refactor the implementation into this framework, allowing users to sample them without having to accept an overhaul of their entire TCP implementation.

7.3 Closing remarks

Benefits	Disadvantages
Gradually deployable	Requires receiver-side changes
No middlebox interference (access to new TCP options, etc.)	No help from PEPs
Lower latency	Slightly lower throughput
Support of priorities	

Table 7.1: Highlighted benefits and disadvantages of TCP-in-UDP and FSE

In summing up, the TCP-in-UDP encapsulation proposed herein overcomes the multi-path routing problem that has hindered TCP coupled congestion control, allowing such mechanisms to practically be used across the Internet. The main drawbacks are that this encapsulation requires support on both ends of a connection, unlike most other TCP congestion control modifications, and that it introduces some (very) minor processing overhead.

We have demonstrated how even a relatively simple, proof of concept TCP congestion control coupling scheme based on a passive Flow State Exchange architecture achieves improved performance based on several metrics.

Using the TCP-in-UDP Happy Eyeballs probing and fallback mechanism, we are able to deliver these advantages to unmodified applications completely without manual intervention, and we also provided rich configuration options both for applications and administrators.

The FSE weighted prioritization mechanism offers further added value for applications, allowing them to tune the Quality of Service different flows receive from the TCP transport layer.

In other words, we have achieved what we set out to do and created a solution based on an efficient UDP encapsulation method, that enables gradually deployable TCP coupled congestion control over the Internet.

Appendix A

Source Code

The full source code of the modified FreeBSD kernel described in this thesis can be obtained from https://bitbucket.org/kristahi/tcpinudp/src. The relevant changes are located in the `sys/netinet/` subdirectory.

All descriptions herein are made in reference to the code as it was at revision `da370ee` (tagged as "thesis_final").

Our modifications are published under the BSD license, see the source files for full copyright and license statements.

Any inquiries about the source code or associated software may be directed to the author by email at kristian.a.hiorth@ieee.org.

Kernel configuration

The kernel configuration files we used are located in the `sys/amd64/conf/` subdirectory. In addition to the standard configurations supplied by the upstream FreeBSD project, we defined the following:

TCPINUDP Development configuration with TCP-in-UDP, TCP CCC and VIMAGE support. Many performance-impacting debugging features are enabled in this configuration.

TCPINUDP-LPROF Release-like configuration with TCP-in-UDP, TCP CCC, VIMAGE and lock profiling support.

TCPINUDP-NODEBUG Release-like configuration with TCP-in-UDP, TCP CCC and VIMAGE support. Used to run the experiments presented in this thesis.

Utilities

In the process of developing and testing the solutions described in this thesis, we extended the `netcat` and `iperf` utilities to interface with the socket options we added.

`netcat` is part of the FreeBSD base system and the source code may be found in the `contrib/netcat` subdirectory.

A patch to the `iperf` FreeBSD port can be found at http://folk.uio.no/kristahi/tcpinudp/.

In order to be able to analyse TCP-in-UDP traffic, we modified the `captcp` tool and the `dpkt` library it depends on. In the process we discovered and fixed defects in captcp (which were reported and accepted upstream). For our modified source code, see https://github.com/kristahi/captcp and https://github.com/kristahi/dpkt.

Bibliography

[1] Jeff Ahrenholz, Claudiu Danilov, Thomas R Henderson and Jae H Kim. 'CORE: A real-time network emulator'. In: *Military Communications Conference, 2008. MILCOM 2008. IEEE*. IEEE. 2008, pp. 1–7. DOI: 10.1109/MILCOM.2008.4753614.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta and Murari Sridharan. 'Data Center TCP (DCTCP)'. In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 63–74. ISSN: 0146-4833. DOI: 10.1145/1851275.1851192.

[RFC3390] Mark Allman, Sally Floyd and Craig Partridge. *Increasing TCP's Initial Window*. RFC 3390. RFC Editor, Oct. 2002. URL: http://www.rfc-editor.org/rfc/rfc3390.txt.

[3] Mark Allman and Vern Paxson. 'On Estimating End-to-end Network Path Properties'. In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 263–274. ISSN: 0146-4833. DOI: 10.1145/316194.316230.

[RFC5681] Mark Allman, Vern Paxson and Ethan Blanton. *TCP Congestion Control*. RFC 5681. RFC Editor, Sept. 2009. URL: http://www.rfc-editor.org/rfc/rfc5681.txt.

[RFC2581] Mark Allman, Vern Paxson and W. Richard Stevens. *TCP Congestion Control*. RFC 2581. RFC Editor, Apr. 1999. URL: http://www.rfc-editor.org/rfc/rfc2581.txt.

[RFC6437] Shane Amante, Brian Carpenter, Sheng Jiang and Jarno Rajahalme. *IPv6 Flow Label Specification*. RFC 6437. RFC Editor, Nov. 2011. URL: http://www.rfc-editor.org/rfc/rfc6437.txt.

[4] David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan and Hari Balakrishnan. 'System Support for Bandwidth Management and Content Adaptation in Internet Applications'. In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation*. OSDI'00. San Diego, California: USENIX Association, 2000. URL: http://dl.acm.org/citation.cfm?id=1251229.1251244.

[5] Grenville Armitage, Lawrence Stewart, Michael Welzl and James Healy. 'An Independent H-TCP Implementation Under FreeBSD 7.0: Description and Observed Behaviour'. In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 27–38. ISSN: 0146-4833. DOI: 10.1145/1384609.1384613.

[6] Vivek Arora, Narin Suphasindhu, John S. Baras and Douglas Dillon. 'Asymmetric Internet access over satellite-terrestrial networks'. In: *Proceedings of the AIAA: 16th International Communications Satellite Systems Conference*. American Institute of Aeronautics and Astronautics, 1996, pp. 476–482. DOI: 10.2514/6.1996-1044.

[7] James Aweya, Michel Ouellette and Delfin Y. Montuno. 'A self-regulating TCP acknowledgment (ACK) pacing scheme'. In: *International Journal of Network Management* 12.3 (2002), pp. 145–163. ISSN: 1099-1190. DOI: 10.1002/nem.426.

[8] Hari Balakrishnan, Venkata N. Padmanabhan and Randy H. Katz. 'The effects of asymmetry on TCP performance'. In: *Mobile Networks and applications* 4.3 (1999), pp. 219–241. DOI: 10.1023/A:1019155000496.

[9] Hari Balakrishnan, Hariharan S. Rahul and Srinivasan Seshan. 'An Integrated Congestion Management Architecture for Internet Hosts'. In: *SIGCOMM Comput. Commun. Rev.* 29.4 (Aug. 1999), pp. 175–187. ISSN: 0146-4833. DOI: 10.1145/316194.316220.

[RFC3124] Hari Balakrishnan and Srinivasan Seshan. *The Congestion Manager*. RFC 3124. RFC Editor, June 2001. URL: http://www.rfc-editor.org/rfc/rfc3124.txt.

[10] G. Barish and K. Obraczke. 'World Wide Web caching: trends and techniques'. In: *IEEE Communications Magazine* 38.5 (May 2000), pp. 178–184. ISSN: 0163-6804. DOI: 10.1109/35.841844.

[11] S. M. Bellovin. 'Security Problems in the TCP/IP Protocol Suite'. In: *SIGCOMM Comput. Commun. Rev.* 19.2 (Apr. 1989), pp. 32–48. ISSN: 0146-4833. DOI: 10.1145/378444.378449.

[12] Mike Belshe and Roberto Peon. *SPDY Protocol*. Internet-Draft draft-mbelshe-httpbis-spdy-00.txt. IETF Secretariat, 23rd Feb. 2012. URL: https://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00.

[RFC7540] Mike Belshe, Roberto Peon and Martin Thomson (editor). *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7540.txt.

[13] J. C. R. Bennett, C. Partridge and N. Shectman. 'Packet reordering is not pathological network behavior'. In: *IEEE/ACM Transactions on Networking* 7.6 (Dec. 1999), pp. 789–798. ISSN: 1063-6692. DOI: 10.1109/90.811445.

[14] Ryan W. Bickhart. 'Transparent TCP-to-SCTP translation shim layer'. M.S. Thesis. University of Delaware, 2005. URL: http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA497279.

[RFC2475] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang and Walter Weiss. *An Architecture for Differentiated Services*. RFC 2475. RFC Editor, Dec. 1998. URL: http://www.rfc-editor.org/rfc/rfc2475.txt.

[15] Ethan Blanton and Mark Allman. 'On Making TCP More Robust to Packet Reordering'. In: *SIGCOMM Comput. Commun. Rev.* 32.1 (Jan. 2002), pp. 20–30. ISSN: 0146-4833. DOI: 10.1145/510726.510728.

[RFC3708] Ethan Blanton and Mark Allman. *Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions*. RFC 3708. RFC Editor, Feb. 2004. URL: http://www.rfc-editor.org/rfc/rfc3708.txt.

[RFC3517] Ethan Blanton, Mark Allman, Kevin Fall and Lili Wang. *A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP*. RFC 2003. RFC Editor, Apr. 2003. URL: http://www.rfc-editor.org/rfc/rfc3517.txt.

[RFC6675] Ethan Blanton, Mark Allman, Kevin Fall, Lili Wang, Ilpo Jarvinen, Markku Kojo and Yoshifumi Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC 6675. RFC Editor, Aug. 2012. URL: http://www.rfc-editor.org/rfc/rfc6675.txt.

[RFC3135] John Border, Markku Kojo, Jim Griner, Gabriel Montenegro and Zach Shelby. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. RFC 3135. RFC Editor, June 2001. URL: http://www.rfc-editor.org/rfc/rfc3135.txt.

[RFC7323] David Borman, Bob Braden, Van Jacobson and Richard Scheffenegger (editor). *TCP Extensions for High Performance*. RFC 7323. RFC Editor, Sept. 2014. URL: http://www.rfc-editor.org/rfc/rfc7323.txt.

[16] Alessio Botta, Alberto Dainotti and Antonio Pescapè. 'A tool for the generation of realistic network workload for emerging networking scenarios'. In: *Computer Networks* 56.15 (2012), pp. 3531–3547. DOI: 10.1016/j.comnet.2012.02.019.

[RFC2309] Bob Braden, David D. Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. K. Ramakrishnan, Scott Shenker, John Wroclawski and Lixia Zhang. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC 2309. RFC Editor, Apr. 1998. URL: http://www.rfc-editor.org/rfc/rfc2309.txt.

[RFC1122] Robert Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. RFC Editor, Oct. 1989. URL: https://rfc-editor.org/rfc/rfc1122.txt.

[17] Gaetano Carlucci, Luca De Cicco and Saverio Mascolo. 'HTTP over UDP: An Experimental Investigation of QUIC'. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: ACM, 2015, pp. 609–614. ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695706.

[RFC3234] Brian E. Carpenter and Scott W. Brim. *Middleboxes: Taxonomy and Issues*. RFC 3234. RFC Editor, Feb. 2002. URL: http://www.rfc-editor.org/rfc/rfc3234.txt.

[RFC6438] Brian Carpenter and Shane Amante. *Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels*. RFC 6438. RFC Editor, Nov. 2011. URL: http://www.rfc-editor.org/rfc/rfc6438.txt.

[18] Jeffrey S. Chase, Andrew J. Gallatin and Kenneth G. Yocum. 'End system optimizations for high-speed TCP'. In: *Communications Magazine, IEEE* 39.4 (2001), pp. 68–74. DOI: 10.1109/35.917506.

[RFC7413] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan and Arvind Jain. *TCP Fast Open*. RFC 7413. RFC Editor, Dec. 2014. URL: http://www.rfc-editor.org/rfc/rfc7413.txt.

[19] Stuart Cheshire, Josh Graessley and Rory McGuire. *Encapsulation of TCP and other Transport Protocols over UDP*. Internet-Draft draft-cheshire-tcp-over-udp-00.txt. IETF Secretariat, 1st July 2014. URL: https://tools.ietf.org/html/draft-cheshire-tcp-over-udp-00.

[RFC6928] Jerry Chu, Nandita Dukkipati, Yuchung Cheng and Matt Mathis. *Increasing TCP's Initial Window*. RFC 6928. RFC Editor, Apr. 2013. URL: http://www.rfc-editor.org/rfc/rfc6928.txt.

[20] David D. Clark, Van Jacobson, John Romkey and Howard Salwen. 'An analysis of TCP processing overhead'. In: *Communications Magazine, IEEE* 27.6 (1989), pp. 23–29. DOI: 10.1109/35.29545.

[21] Various FreeBSD contributors. *tcp_rfc_compliance - FreeBSD Wiki*. 4th Jan. 2016. URL: https://wiki.freebsd.org/TransportProtocols/tcp_rfc_compliance (visited on 27/04/2016).

[22] Mark E. Crovella and Azer Bestavros. 'Self-similarity in World Wide Web traffic: evidence and possible causes'. In: *Networking, IEEE/ACM Transactions on* 5.6 (1997), pp. 835–846. DOI: 10.1109/90.650143.

[23] Jon Crowcroft and Philippe Oechslin. 'Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP'. In: *SIGCOMM Comput. Commun. Rev.* 28.3 (July 1998), pp. 53–69. ISSN: 0146-4833. DOI: 10.1145/293927.293930.

[24] Dragana Damjanovic and Michael Welzl. 'MulTFRC: providing weighted fairness for multimediaapplications (and others too!)' In: *SIGCOMM Comput. Commun. Rev.* 39.3 (2009), pp. 5–12. ISSN: 0146-4833. DOI: 10.1145/1568613.1568615.

[25] Rémi Denis-Courmont. *UDP-Encapsulated Transport Protocols*. Internet-Draft draft-denis-udp-transport-00.txt. IETF Secretariat, 4th July 2008. URL: https://tools.ietf.org/html/draft-denis-udp-transport-00.

[26] Nandita Dukkipati, Matt Mathis, Yuchung Cheng and Monia Ghobadi. 'Proportional Rate Reduction for TCP'. In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '11. Berlin, Germany: ACM, 2011, pp. 155–170. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068832.

[27] Jay Duncanson. 'Inverse multiplexing'. In: *IEEE Communications Magazine* 32.4 (Apr. 1994), pp. 34–41. ISSN: 0163-6804. DOI: 10.1109/35.275333.

[28] Brian Trammell (editor) and Mirja Kuehlewind (editor). *Requirements for the design of a Substrate Protocol for User Datagrams (SPUD)*. Internet-Draft draft-trammell-spud-req-02.txt. IETF Secretariat, 11th Mar. 2016. URL: https://tools.ietf.org/html/draft-trammell-spud-req-02.

[RFC5944] Charles E. Perkins (editor). *IP Mobility Support for IPv4, Revised*. RFC 5944. RFC Editor, Nov. 2010. URL: http://www.rfc-editor.org/rfc/rfc5944.txt.

[29] David Hayes (editor), Simone Ferlin, Michael Welzl and Kristian Hiorth. *Shared Bottleneck Detection for Coupled Congestion Control for RTP Media*. Internet-Draft draft-ietf-rmcat-sbd-04.txt. IETF Secretariat, 21st Mar. 2016. URL: https://tools.ietf.org/html/draft-ietf-rmcat-sbd-04.

[RFC793] Jon Postel (editor). *Transmission Control Protocol*. RFC 793. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[RFC4960] Randall R. Stewart (editor). *Stream Control Transmission Protocol*. RFC 4960. RFC Editor, Sept. 2007. URL: http://www.rfc-editor.org/rfc/rfc4960.txt.

[RFC7230] Roy T. Fielding (editor) and Julian F. Reschke (editor). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor, June 2014. URL: http://www.rfc-editor.org/rfc/rfc7230.txt.

[30] Lars Eggert, John Heidemann and Joe Touch. 'Effects of Ensemble-TCP'. In: *ACM SIGCOMM Computer Communication Review* 30.1 (2000), pp. 15–29. DOI: 10.1145/505688.505691.

[31] Kevin Fall and Sally Floyd. 'Simulation-based Comparisons of Tahoe, Reno and SACK TCP'. In: *SIGCOMM Comput. Commun. Rev.* 26.3 (July 1996), pp. 5–21. ISSN: 0146-4833. DOI: 10.1145/235160.235162.

[RFC3742] Sally Floyd. *Limited Slow-Start for TCP with Large Congestion Windows*. RFC 3742. RFC Editor, Mar. 2004. URL: http://www.rfc-editor.org/rfc/rfc3742.txt.

[32] Sally Floyd, Ramakrishna Gummadi and Scott Shenker. *Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management*. Technical report. AT&T Center for Internet Research at ICSI, 2001. URL: http://icir.org/floyd/papers/adaptiveRed.pdf.

[RFC2582] Sally Floyd and Tom Henderson. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 2582. RFC Editor, Apr. 1999. URL: https://rfc-editor.org/rfc/rfc2582.txt.

[33] Sally Floyd and Van Jacobson. 'Random early detection gateways for congestion avoidance'. In: *IEEE/ACM Transactions on Networking* 1.4 (Aug. 1993), pp. 397–413. ISSN: 1063-6692. DOI: 10.1109/90.251892.

[34] Sally Floyd and Vern Paxson. 'Difficulties in Simulating the Internet'. In: *IEEE/ACM Trans. Netw.* 9.4 (Aug. 2001), pp. 392–403. ISSN: 1063-6692. DOI: 10.1109/90.944338.

[RFC6824] Alan Ford, Costin Raiciu, Mark Handley and Olivier Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. RFC Editor, Jan. 2013. URL: http://www.rfc-editor.org/rfc/rfc6824.txt.

[35] Bryan Ford, Pyda Srisuresh and Dan Kegel. 'Peer-to-Peer Communication Across Network Address Translators.' In: *Proceedings of the USENIX Annual Technical Conference 2005 (USENIX '05)*. Anaheim, CA: USENIX, 2005. URL: https://www.usenix.org/legacy/event/usenix05/tech/general/full_papers/ford/ford_html/.

[36] Fayez Gebali. *Analysis of Computer Networks*. 2nd ed. Springer International Publishing, 2015. ISBN: 9783319156576. DOI: 10.1007/978-3-319-15657-6.

[RFC6093] Fernando Gont and Andrew Yourtchenko. *On the Implementation of the TCP Urgent Mechanism*. RFC 6093. RFC Editor, Jan. 2011. URL: http://www.rfc-editor.org/rfc/rfc6093.txt.

[37] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: A New TCP-friendly High-speed TCP Variant'. In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105.

[38] Ryan Hamilton, Janardhan Iyengar, Ian Swett and Alyssa Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-tsvwg-quic-protocol-02.txt. IETF Secretariat, 16th Jan. 2016. URL: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.

[39] David A. Hayes, Simone Ferlin and Michael Welzl. 'Practical passive shared bottleneck detection using shape summary statistics'. In: *2014 IEEE 39th Conference on Local Computer Networks (LCN)*. IEEE. Sept. 2014, pp. 150–158. DOI: 10.1109/LCN.2014.6925767.

[40] David Hayes, Lawrence Stewart and Grenville Armitage. *Evaluating the FreeBSD 9. x Modular Congestion Control Framework's Performance Impact*. Tech. rep. 110228A. Centre for Advanced Internet Architectures, Swinburne University of Technology, 28th Feb. 2011. URL: http://caia.swin.edu.au/reports/110228A/CAIA-TR-110228A.pdf.

[RFC6582] Tom Henderson, Sally Floyd, Andrei Gurtov and Yoshifumi Nishida. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. RFC Editor, Apr. 2012. URL: https://rfc-editor.org/rfc/rfc6582.txt.

[41] Tom Herbert, Lucy Yong and Osama Zia. *Generic UDP Encapsulation*. Internet-Draft draft-ietf-nvo3-gue-02.txt. IETF Secretariat, 22nd Dec. 2015. URL: https://tools.ietf.org/html/draft-ietf-nvo3-gue-02.

[42] Janey C. Hoe. 'Improving the Start-up Behavior of a Congestion Control Scheme for TCP'. In: *SIGCOMM Comput. Commun. Rev.* 26.4 (Aug. 1996), pp. 270–280. ISSN: 0146-4833. DOI: 10.1145/248157.248180.

[43] Janey C. Hoe. 'Start-up dynamics of TCP's congestion control and avoidance schemes'. M.S. Thesis. Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1995. DOI: 1721.1/36971.

[44] Toke Høiland-Jørgensen, Paul McKenney, Dave Taht, Jim Gettys and Eric Dumazet. *The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm*. Internet-Draft draft-ietf-aqm-fq-codel-06.txt. IETF Secretariat, 18th Mar. 2016. URL: https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06.

[45] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley and Hideyuki Tokuda. 'Is It Still Possible to Extend TCP?' In: *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '11. Berlin, Germany: ACM, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834.

[RFC2992] Christian E. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. RFC 2992. RFC Editor, Nov. 2000. URL: http://www.rfc-editor.org/rfc/rfc2992.txt.

[46] *IEEE Standard for Local and metropolitan area networks – Link Aggregation*. IEEE Std 802.1AX-2014. 2014. DOI: 10 . 1109 / IEEESTD.2014.7055197.

[47] Safiqul Islam, Michael Welzl and Stein Gjessing. *Coupled congestion control for RTP media*. Internet-Draft draft-ietf-rmcat-coupled-cc-02.txt. IETF Secretariat, 14th Apr. 2016. URL: https://tools.ietf.org/html/draft-ietf-rmcat-coupled-cc-02.

[48] Safiqul Islam, Michael Welzl, Stein Gjessing and Naeem Khademi. 'Coupled congestion control for RTP media'. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 21–26. DOI: 10.1145/2630088.2630089.

[49] Safiqul Islam, Michael Welzl, David Hayes and Stein Gjessing. 'Managing Real-Time Media Flows through a Flow State Exchange'. In: *Proceedings of the 28th IEEE Network Operations and Management Symposium*. Vol. 28. NOMS 2016. IEEE. Apr. 2016.

[50] Van Jacobson. 'Congestion Avoidance and Control'. In: *SIGCOMM Computer Communication Review* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: 10.1145/52325.52356.

[51] Van Jacobson. *modified TCP congestion avoidance algorithm*. Post to the end2end interest discussion mailing list. 30th Apr. 1990. URL: ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail.

[52] Poul-Henning Kamp and Robert N. M. Watson. 'Jails: Confining the omnipotent root.' In: *Proceedings of the 2nd International SANE Conference*. 2000. URL: http://fledge.watson.org/~robert/freebsd/sane2000-jail.pdf.

[53] P. Karn and C. Partridge. 'Improving Round-trip Time Estimates in Reliable Transport Protocols'. In: *SIGCOMM Comput. Commun. Rev.* 17.5 (Aug. 1987), pp. 2–7. ISSN: 0146-4833. DOI: 10.1145/55483.55484.

[54] Angelos D. Keromytis. 'Tagging Data in the Network Stack: mbuf_tags.' In: *Proceedings of BSDCon '03*. USENIX. Sept. 2003, pp. 125–132. URL: https://www.usenix.org/event/bsdcon03/tech/full_papers/keromytis/keromytis_html/.

[55] Ka-Cheong Leung, Victor OK Li and Daiqin Yang. 'An overview of packet reordering in transmission control protocol (TCP): problems, solutions, and challenges'. In: *Parallel and Distributed Systems, IEEE Transactions on* 18.4 (2007), pp. 522–535. DOI: 10.1109/TPDS.2007.1011.

[56] Jonathan Looney. *tcp_rfc_compliance - FreeBSD Wiki*. 12th Nov. 2016. URL: https://wiki.freebsd.org/TransportProtocols/TCPModularity (visited on 10/05/2016).

[57] Salvatore Loreto and Simon Pietro Romano. 'Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts'. In: *Internet Computing, IEEE* 16.5 (2012), pp. 68–73. DOI: 10.1109/MIC.2012.115.

[58] Reiner Ludwig and Randy H. Katz. 'The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions'. In: *SIGCOMM Comput. Commun. Rev.* 30.1 (Jan. 2000), pp. 30–36. ISSN: 0146-4833. DOI: 10.1145/505688.505692.

[RFC3522] Reiner Ludwig and Michael Meyer. *The Eifel Detection Algorithm for TCP*. RFC 3522. RFC Editor, Apr. 2003. URL: http://www.rfc-editor.org/rfc/rfc3522.txt.

[RFC5766] Rohan Mahy, Philip Matthews and Jonathan Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766. RFC Editor, Apr. 2010. URL: http://www.rfc-editor.org/rfc/rfc5766.txt.

[59] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi and Ren Wang. 'TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links'. In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. MobiCom '01. Rome, Italy: ACM, 2001, pp. 287–297. ISBN: 1-58113-422-3. DOI: 10.1145/381677.381704.

[RFC6937] Matt Mathis, Nandita Dukkipati and Yuchung Cheng. *Proportional Rate Reduction for TCP*. RFC 6937. RFC Editor, May 2013. URL: http://www.rfc-editor.org/rfc/rfc6937.txt.

[RFC2018] Matt Mathis, Jamshid Mahdavi, Sally Floyd and Allyn Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. RFC Editor, Oct. 1996. URL: http://www.rfc-editor.org/rfc/rfc2018.txt.

[60] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi and Teunis Ott. 'The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm'. In: *SIGCOMM Comput. Commun. Rev.* 27.3 (July 1997), pp. 67–82. ISSN: 0146-4833. DOI: 10.1145/263932.264023.

[61] Marshall Kirk McKusick, George V. Neville-Neil and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. 2nd ed. Pearson Education, 2014. ISBN: 9780133761832.

[62] Stephen McQuistin and Colin Perkins. 'Is Explicit Congestion Notification usable with UDP?' In: *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM. 2015, pp. 63–69. DOI: 10.1145/2815675.2815716.

[63] Anders Grotthing Moe. 'Implementing Rate Control in NetEm : Untying the NetEm/tc tangle'. Master thesis. University of Oslo, 2013. DOI: 10852/37459.

[64] Kathleen Nichols and Van Jacobson. 'Controlling Queue Delay'. In: *Queue* 10.5 (May 2012), 20:20–20:34. ISSN: 1542-7730. DOI: 10.1145/2208917.2209336.

[65] Florian Niederbacher. 'Beneficial gradual deployment of SCTP'. Dipl.-Arb. Universität Innsbruck, 2010. URL: http://search.obvsg.at/UIB:Blended:UIB_aleph_acc001617070.

[66] Stephen Northcutt, Lenny Zeltser, Scott Winters, Karen Kent and Ronald W. Ritchey. *Inside Network Perimeter Security*. 2nd ed. Indianapolis, IN, USA: Sams, 2005. ISBN: 0672327376.

[67] Michael F. Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin and Bryan Ford. 'Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS'. In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 383–398. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/nowlan.

[68] Jitendra Padhye, Victor Firoiu, Donald F. Towsley and James F. Kurose. 'Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation'. In: *IEEE/ACM Trans. Netw.* 8.2 (Apr. 2000), pp. 133–145. ISSN: 1063-6692. DOI: 10.1109/90.842137.

[69] Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili S. Prabhu, Vijay Subramanian, Fred Baker and Bill VerSteeg. 'PIE: A lightweight control scheme to address the bufferbloat problem'. In: *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*. July 2013, pp. 148–155. DOI: 10.1109/HPSR.2013.6602305.

[70] Charles E. Perkins. 'Mobile IP'. In: *IEEE Communications Magazine* 40.5 (May 2002), pp. 66–82. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1006976.

[71] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain and Barath Raghavan. 'TCP Fast Open'. In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. CoNEXT '11. Tokyo, Japan: ACM, 2011, 21:1–21:12. ISBN: 978-1-4503-1041-3. DOI: 10.1145/2079296.2079317.

[72] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik and Mark Handley. 'Improving Datacenter Performance and Robustness with Multipath TCP'. In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 266–277. ISSN: 0146-4833. DOI: 10.1145/2043164.2018467.

[RFC3168] K. K. Ramakrishnan, Sally Floyd and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. RFC Editor, Sept. 2001. URL: http://www.rfc-editor.org/rfc/rfc3168.txt.

[RFC5245] Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. RFC 5245. RFC Editor, Apr. 2010. URL: http://www.rfc-editor.org/rfc/rfc5245.txt.

[RFC6544] Jonathan Rosenberg, Ari Keranen, Bruce B. Lowekamp and Adam Roach. *TCP Candidates with Interactive Connectivity Establishment (ICE)*. RFC 6544. RFC Editor, Mar. 2012. URL: http://www.rfc-editor.org/rfc/rfc6544.txt.

[RFC5389] Jonathan Rosenberg, Rohan Mahy, Philip Matthews and Dan Wing. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. RFC Editor, Oct. 2008. URL: http://www.rfc-editor.org/rfc/rfc5389.txt.

[RFC3261] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley and Eve Schooler. *SIP: Session Initiation Protocol*. RFC 3261. RFC Editor, June 2002. URL: http://www.rfc-editor.org/rfc/rfc3261.txt.

[73] Dan Rubenstein, Jim Kurose and Don Towsley. 'Detecting Shared Congestion of Flows via End-to-end Measurement'. In: *IEEE/ACM Trans. Netw.* 10.3 (June 2002), pp. 381–395. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.1012369.

[74] Pasi Sarolahti, Markku Kojo and Kimmo Raatikainen. 'F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts'. In: *SIGCOMM Comput. Commun. Rev.* 33.2 (Apr. 2003), pp. 51–63. ISSN: 0146-4833. DOI: 10.1145/956981.956987.

[RFC5682] Pasi Sarolahti, Markku Kojo, Kazunori Yamamoto and Max Hata. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP*. RFC 5682. RFC Editor, Sept. 2009. URL: http://www.rfc-editor.org/rfc/rfc5682.txt.

[75] Michael Savorić, Holger Karl, Morten Schläger, Tobias Poschwatta and Adam Wolisz. 'Analysis and performance evaluation of the EFCM common congestion controller for TCP connections'. In: *Computer Networks* 49.2 (2005), pp. 269–294. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2005.01.012.

[76] *SPDY: An experimental protocol for a faster web*. Google. URL: http://www.chromium.org/spdy/spdy-whitepaper (visited on 23/04/2016).

[RFC2663] Pyda Srisuresh and Matt Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 6093. RFC Editor, Aug. 1999. URL: http://www.rfc-editor.org/rfc/rfc2663.txt.

[RFC2001] W. Richard Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. RFC Editor, Jan. 1997. URL: https://rfc-editor.org/rfc/rfc2001.txt.

[77] Lawrence Stewart, Grenville Armitage and James Healy. *Characterising the Behaviour and Performance of SIFTR v1.1.0*. Technical report 070824A. Centre for Advanced Internet Architectures, Swinburne University of Technology, Aug. 2007. URL: http://caia.swinburne.edu.au/reports/070824A/CAIA-TR-070824A.pdf.

[RFC2991] Dave Thaler and Christian E. Hopps. *Multipath Issues in Unicast and Multicast Next-Hop Selection*. RFC 2991. RFC Editor, Nov. 2000. URL: http://www.rfc-editor.org/rfc/rfc2991.txt.

[RFC7016] Michael C. Thornburgh. *Adobe's Secure Real-Time Media Flow Protocol*. RFC 7016. RFC Editor, Nov. 2013. URL: http://www.rfc-editor.org/rfc/rfc7016.txt.

[RFC6994] Joe Touch. *Shared Use of Experimental TCP Options*. RFC 6994. RFC Editor, Aug. 2013. URL: http://www.rfc-editor.org/rfc/rfc6994.txt.

[RFC2140] Joe Touch. *TCP Control Block Interdependence*. RFC 2140. RFC Editor, Apr. 1997. URL: http://www.rfc-editor.org/rfc/rfc2140.txt.

[78] Brian Trammell, Mirja Kuehlewind, Elio Gubser and Joe Hildebrand. 'A New Transport Encapsulation for Middlebox Cooperation'. In: *Proceedings of the 2015 IEEE Conference on Standards for Communications and Networking*. Tokyo, Japan, Oct. 2015. DOI: 10.1109/CSCN.2015.7390442.

[79] *Understanding ECMP Flow-Based Forwarding*. Juniper Networks, Inc. 12th May 2014. URL: http://www.juniper.net/documentation/en_US/junos14.1/topics/concept/routing-policy-security-ecmp-flow-based-forwarding-understanding.html (visited on 25/04/2016).

[80] D. Watson, M. Smart, G. R. Malan and F. Jahanian. 'Protocol scrubbing: network security through transparent flow modification'. In: *IEEE/ACM Transactions on Networking* 12.2 (Apr. 2004), pp. 261–273. ISSN: 1063-6692. DOI: 10.1109/TNET.2003.822645.

[81] Michael Welzl. *Network Congestion Control: Managing Internet Traffic*. Wiley Series on Communications Networking & Distributed Systems. Wiley, 2005. ISBN: 9780470025291.

[82] Michael Welzl, Safiqul Islam, Kristian Hiorth and Jianjie You. *TCP in UDP*. Internet-Draft draft-welzl-irtf-iccrg-tcp-in-udp-00.txt. IETF Secretariat, 21st Mar. 2016. URL: https://tools.ietf.org/html/draft-welzl-irtf-iccrg-tcp-in-udp-00.

[83] Michael Welzl, Florian Niederbacher and Stein Gjessing. 'Beneficial transparent deployment of SCTP: the missing pieces'. In: *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*. IEEE. 2011, pp. 1–5. DOI: 10.1109/GLOCOM.2011.6133554.

[RFC6555] Dan Wing and Andrew Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555. RFC Editor, Apr. 2012. URL: http://www.rfc-editor.org/rfc/rfc6555.txt.

[84] Damon Wischik, Costin Raiciu, Adam Greenhalgh and Mark Handley. 'Design, Implementation and Evaluation of Congestion Control for Multipath TCP'. In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Vol. 11. Boston, MA: USENIX, 2011, pp. 99–112. URL: https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Wischik.pdf.

[85] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated. The Implementation*. Vol. 2. Addison-Wesley Professional Computing Series. Pearson Education, 1995. ISBN: 9780321617644.

[86] Marko Zec. 'Implementing a Clonable Network Stack in the FreeBSD Kernel.' In: *USENIX Annual Technical Conference, FREENIX Track*. 2003, pp. 137–150. URL: https://www.usenix.org/legacy/events/usenix03/tech/freenix03/full_papers/zec/zec_html/.

[87] Marko Zec, Miljenko Mikuc and Mario Zagar. 'Estimating the impact of interrupt coalescing delays on steady state TCP throughput'. In: *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 2002.