A Dual-Channel Approach to Protocol Design in the Presence of Middleboxes

Steve Wang
Justine Sherry
Sangjin Han

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-205
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-205.html

December 13, 2013

A Dual-Channel Approach to Protocol Design in the Presence of Middleboxes

Steve Wang *Justine Sherry* *Sangjin Han*
UC Berkeley

Abstract

To improve security and performance, middleboxes (such as firewalls or proxies) may inspect and transform packet contents, delete and inject packets to active flows, and may even reset or terminate entire connections. However, for novel protocols which may not conform to common flow behaviors, middleboxes can interfere with or even block outright the use of these new protocols.

In this paper, we present a dual-channel design approach as a way for new protocols to achieve middlebox-friendliness. Under the dual-channel approach, data traffic is sent over a traditional TCP/UDP channel, and additional functionality is accommodated within the payload of a secondary channel. As a case study, we present our prototype implementation of Forward Error Correction for TCP which improves flow completion times by almost $2.5\times$ *under 2% loss, despite the overhead of the secondary channel.*

1 Introduction

Middleboxes – such as firewalls, caches, and WAN optimizers – are recognized as a primary challenge to the design of new protocols, especially to new proposals which extend classic protocols, *i.e.*, IP, TCP, and UDP [5, 9]. A middlebox may change packet contents (*e.g.*, rewriting sequence numbers or source/destination IP addresses), remove packet contents (*e.g.*, stripping out IP options), drop packets (*e.g.*, filtering traffic to or from a particular port), or even inject new packets (*e.g.*, injecting TCP RSTs to terminate a connection). These manipulations can interfere with protocol extensions which rely on the contents of the modified fields. As middleboxes become more and more common, these impediments to protocol deployments impact more and more users. Indeed, middleboxes are now common in enterprise, home, broadband, and cellular networks [4, 15, 17].

Consequently, a key goal in modern protocol design is *middlebox-friendliness*: that the protocol continue to operate correctly and fully even in the presence of middleboxes. Nevertheless, given the challenge of achieving middlebox-friendliness, many protocol designers have aimed for the more limited goal of *avoiding* designing protocols which are *middlebox-adverse*. A protocol which is middlebox-adverse hinders communication by performing incorrect behavior (delivering incorrect or out of order bytes) or failing to transmit at all. A protocol may avoid being middlebox-adverse yet still not be middlebox-friendly: many protocols fail in the presence of middleboxes by defaulting to basic TCP, allowing communication to continue but without the benefits of the new protocol extension. Given the widespread deployment of middleboxes, a protocol which is not middlebox-friendly (even if it is not middlebox-adverse) may expect only limited applicability.

One design approach for middlebox-friendly protocols is to make a thorough study of which fields are commonly modified in the target deployment environment, and then to simply "design around" these fields. For example, if hardware in the target deployment environment strips IP options, but leaves TCP options intact, a protocol designer might choose to place protocol-specific bits in the TCP header rather than the IP header. This approach is a headache for protocol designers, who must carefully evaluate where and how changes may occur in order to decide where to wedge in new control data for the extension.

In this paper, we propose a *dual channel approach* to protocol design with the goal of achieiving middlebox-friendliness in a general fashion. Protocols designed with a dual channel approach send data using existing, unmodified protocols – TCP or UDP over IP, with no special bits set and no new data wedged in to the protocol. Instead, all control traffic or additional data for the new protocol is transmitted within the payload over a secondary UDP channel. Because the protocol extension essentially operates at the application layer, middleboxes that interfere with any or all header fields will still leave the control data in the secondary channel intact. While this approach cannot encapsulate every possible protocol extension, we believe it is general enough to aid *many* types of extensions.

To illustrate the dual channel approach, we present an extension to TCP, 2CFEC, which augments TCP with Forward Error Correction for improved recovery from packet loss. Despite the overhead of the secondary channel, 2CFEC improves typical flow completion times by almost $2.5\times$. The rest of the paper is organized as follows. In §2 we describe our case study, forward error correction for TCP. In §3, we describe our approach, paying attention to the design choices we make to preserve middlebox-friendliness. Finally, in §4, we discuss lessons learned from our design and other protocol extensions which might benefit from the dual-channel approach.

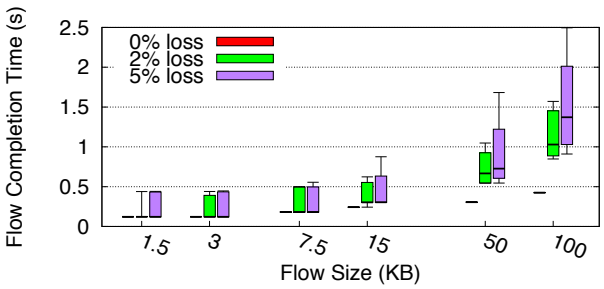


Fig. 1: Flow completion time in controlled experiments (10Gbps BW/100ms RTT) with random loss. Boxplots show 1-5-50-95-99 percentiles.

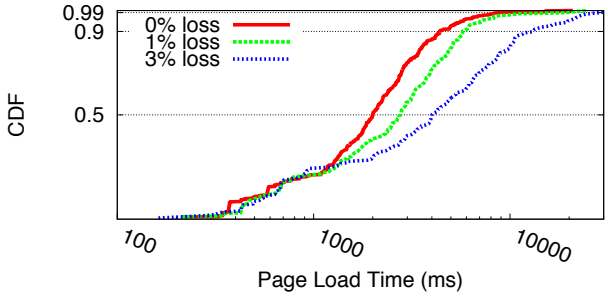


Fig. 2: Page load times for Alexa Top-200 Web sites, on the same emulated link.

2 Case Study: FEC

The vast majority of TCP flows are small, and in the absence of loss, they complete in a few round-trips. However, they are also much more severely penalized by packet loss than long flows. Because they have so few packets, loss may not trigger TCP 'fast retransmit', but instead leave the connection stalled until a timeout triggers retransmission, which can be often orders of magnitude higher than the actual RTT [6]. The short flow problem can especially hinder web users, as most HTTP connections carry small objects [2, 6]. Even worse, Web requests involve many interdependent flows, leading to a straggler problem when any single one of the flows suffers a loss.

In order to avoid the timeout + RTT penalty from TCP retransmissions, some have proposed adding forward error correction (FEC) to TCP to recover from packet loss more quickly [3, 6]. FEC is a technique for reliable data transmission over a lossy/noisy channel. Under FEC with systematic coding, a sender transmits redundant encoded data along with the original data; when the bit stream has gaps, this redundant data allows the receiver to reconstruct the missing data immediately.

An overly-idealized FEC scheme might be able to recover from all loss. In Figure 1, we show microbenchmarks of flow completion times over a 10 Gbps BW/100 ms RTT link when loss is introduced randomly with 5% of packets lost, 2% of packets lost, or, the ideal case, when either no packets are lost or an FEC scheme allows for recovery from all lost packets. With no loss, we see that flow completion times have very low variance and complete entirely in under 0.5s at all the shown flow sizes; loss leads to high variance in FCTs and overall, 3-5× worse performance. In Figure 2, we see the potential benefits in web page load times from an ideal FEC implementation. The figure shows a CDF of page load times for the Alexa top-200 sites, once again with the same link configuration. We see that performance penalties are primarily in the median page load times: at 3% loss median pages take 2× more time than in the optimal (0% loss) scenario. These results confirm that Web browsing can greatly benefit from FEC in the presence of any packet loss.

We are not the first to propose augmenting FEC to TCP [3, 6]. A common thread among existing designs is to introduce new TCP options and divert existing TCP fields into different usages. This approach has advantages in terms of efficiency and simplicity but can lead to incompatibility with middleboxes. For example, the FEC scheme proposed by Flach et al. [6] carries the encoded redundancy data in-band, along with special TCP options set. This implementation has two potential issues with TCP-understanding middleboxes: *i)* they can strip out unknown TCP options or even block the entire flow [9]; *ii)* since the original data and redundancy data share the same TCP sequence number, middleboxes may mistake this behavior as a subterfuge attack [12]. As we'll discuss below, the two-channel approach avoids the above challenges with middlebox-friendliness by placing control and redundancy information in a secondary channel, while achieving the same goals as the traditional single-channel approach.

3 FEC in a Dual-Channel Design

In this section, we describe our dual-channel implementation of Forward Error Correction for TCP. We start by describing our protocol, 2CFEC, in detail in §3.1. In §3.2, We then discuss deployability of 2CFEC, relying on measurement studies of middlebox behaviours in the wild. Finally, in §3.3, we compare flow completion times with 2CFEC against the ideal FEC implementation from §2.

3.1 Protocol Design

We now present the key aspects of 2CFEC which enable middlebox-friendliness.[1] Using the dual channel approach, 2CFEC is robust to modification of several header fields at a time.

We refer to the two channels as the *data channel* (primary) and the *FEC channel* (secondary). The data channel

[1] A full design description will be available in a forthcoming technical report.

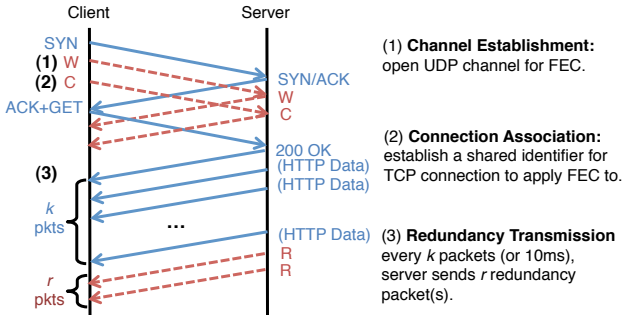


Fig. 3: Data and FEC channel between client and server.

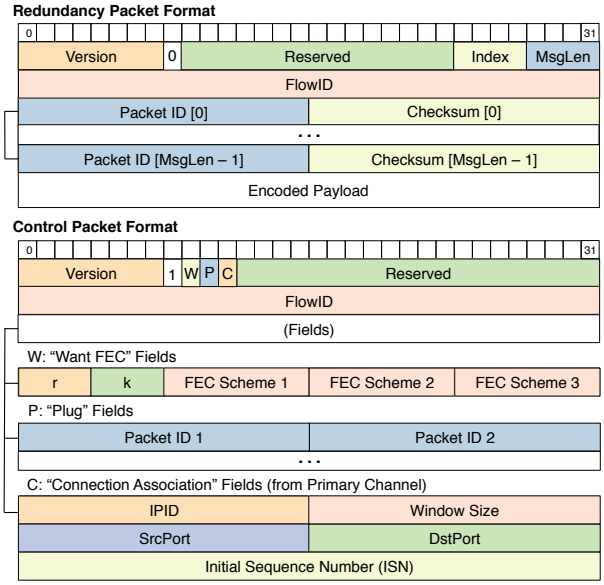


Fig. 4: Packet formats in FEC channel.

operates over standard TCP and requires no additional explanation. The FEC channel runs in parallel to the data channel over a UDP connection. There are three main phases in FEC channel operation: channel establishment, connection association, and redundancy transmission. All three phases are illustrated in Figure 3.

Channel Establishment. The first stage is "channel establishment" – setting up the parallel UDP connection itself and agreeing on a mutually understood redundancy protocol. In Figure 4, we illustrate the packet formats for traffic sent over the FEC channel. The lower packet format is for "control packets". The client initiates[2] channel establishment by sending a control packet with the `W` flag enabled. The client also appends the relevant options for the W flag: r, an integer representing the redundancy factor for the connection; k, the maximum number of packets to include in a single redundancy packet; and finally a list of known redundancy schemes (such as XOR, Reed-Solomon [14], or Tornado Codes [8]). The server then replies with a W packet, agreeing to the proposed r and k values, but supplying only one FEC scheme: a scheme which both the client and server support which will be used for future redundancy packets.

Connection Association. The second stage is "connection association." Since the client and server may have several active connections between them at once, any redundancy packets sent must be *associated* with one of the connections between client and server. Either client or server may initiate connection association as soon as a SYN packet has been transmitted and received for the primary data connection. To initiate connection association, an end host transmits a control packet with the `C` — "Connection Association" flag set. In the fields for a Connection Association request, the sender includes five values which should uniquely identify the data channel the sender wishes to enable FEC for. These fields are the IPID value for the first packet in the data channel, the first window size announced in the data channel, the source and destination ports from the data channel, and the initial sequence number from the data channel. The sender also includes a FlowID which will henceforth be included in all redundancy packets for data from the specified channel. Under IPv4, FlowID is an arbitrary 32-bit value randomly generated by the sender; under IPv6 it is the FlowID value from the data channel.

In the simplest case – where there are no middleboxes present and the client and server have only one connection between them – connection association is merely a quick step to establish a FlowID. When there are multiple data connections present between the server and client, the receiver uses the appended options data to determine to which connection the new FlowID should apply. Although traditionally the traditional flow identifier is the classic 'five-tuple' (protocol, Src/Dest IP, and Src/Dest port), we choose to use the initial IPID value as our primary identifier since it is the field least often modified by middleboxes: in a recent study IPID was been observed to be modified by middleboxes in than 1% of paths [1]. For the rare cases where IPID fails to identify a connection, we also include the initial window size, the full tuple, and the initial sequence number – any of which may be modified, but are unlikely to *all* be modified at once.

Redundancy Transmission. After Connection Association, either sender (or both) may begin sending FEC redundancy packets. Redundancy packets are sent after every k packets (for k negotiated in the Channel Establishment phase), or pre-emptively after 10ms of idleness (if

[2] A server could just as easily initiate channel establishment. However, as clients are often behind NATs, channel establishment is more likely to be successful when initiated by the client. In P2P scenarios, one could extend channel establishment to make use of hole-punching and port-mapping protocols [16, 18]; as these techniques are well-known, they are not relevant to the contribution of this work.

MBox	Behavior	Probability	2	Fl.
NAT	Src IP Rewrite	90% [10, 11]	F	F
	Src Port Rewrite	27% [10]	F	F
WAN Opt.	Compression	-	F	F
	Protocol Accel.	-	I	A/I
Firewall	IPID Rewrite	0.025% [1]	I	F
	TCP Opt. Rewrite	4-14% [9]	F	I
	IP Opt. Rewrite	34-66% [7]	F	F
	Seq. No. Rewrite	7-18% [9]	F	F
	Port Filtering	5.9% [10]	I	F
	Gap Monitoring	24-33% [9]	F	F
	Dup. Interference	1-13% [9]	F	A/I
Proxy	Inline Caching	7.6% [10]	I	A/I
	Conn. Termination	2.7% [10]	I	I
	Resegmentation	1-13% [9]	I	A/I

Table 1: Protocol compatibililty with common middlebox behaviors. 'Probability' indicates observed fraction of end hosts behind middleboxes with named behavior. For each protocol, 'F' indicates 'Friendly', 'I' indicates 'Incompatible' (but not adverse - the protocol fails over to standard TCP), and 'A' indicates adverse (the connection is unable to proceed).

there remain no more packets in the transmission buffer). The receiver checks the stored k packets against the checksums listed in the redundancy packet to protect against payload-modifying middleboxes (such as proxies) — if the receiver attempted to regenerate a dropped packet using proxy-manipulated packets, it might generate invalid data. If a packet is missing and k stored packets are valid, the receiver uses this information to recover the lost packet(s).

The receiver sends a P ('Plug') packet whenever a redundancy packet is used for loss recovery: the Plug packet contains a list of which packets were recovered so that the sender may re-transmit these lost packets. The sender retransmits the lost packets, despite the fact that they have already been recovered, in order to ensure that any intermediary middleboxes keeping session state can note the transmission of that sequence number.[3] Every estimated RTT, the sender re-sends Plug packets with a list of all 'gap' packets until there are no remaining gaps. Redundancy transmission phase continues until the connection terminates, at which point FEC is done.

3.2 Deployability

Having described some key design choices in our protocol, we now evaluate the deployability of 2CFEC using data from Honda et al. [9] and Kreibich et al. [10] on common edge-network middlebox behaviors in the wild. We present our analysis of our 2CFEC (column '2'), in comparison to the single-channel FEC implementation by Flach et al. (column 'Fl.') [6], in Table 1. The table shows common protocol manipulations broken down by the type of middlebox with which they are most commonly associated. If known, we present the observed probability of this behavior in the wild.

Both protocols are friendly under typical NAT behaviors. It is theoretically possible that a NAT might rewrite the source address of the data channel and the FEC channel to different addresses, but that has not been typically observed in practice.[4]. Under these conditions, 2CFEC would fail back to basic TCP for security reasons.

Neither protocol is friendly when a middlebox changes payload contents, such as via any proxying behavior, or when protocol acceleration injects new packets. This behavior is a fundamental incompatibility with FEC, as packet payload contents at source and receiver no longer match. We note that while the Flach design as presented is technically adverse as presented in the paper – it would result in incorrect data being delivered to the receiver – the design could easily be extended to include packet hashes (as 2CFEC does) to safeguard against incorrect data being delivered.

Firewall/IPS behaviors are the most varied. 2CFEC is incompatible when (a) IPID is rewritten (and there are two or more connections to the server, and the basic five-tuple, window size, and initial sequence number are rewritten simultaneously — $\leq$0.025% of cases), or (b) when access to unknown ports is restricted — 5.9% of cases. One could extend the protocol to transmit the FEC channel over HTTP (port 80) [13], which is less frequently restricted: only 3.6% of the time [10]. The Flach design is incompatible when TCP Options are rewritten (4-14% of cases, most often when the data channel is over port 80) or when the firewall does not allow duplicate, non-matching packets with the same sequence number (1-13% of the time, once again with more interference over port 80 connections).

Overall, both protocols succeed when faced with NAT, neither succeeds when faced with proxying or protocol acceleration, and 2CFEC is more often successful when faced with firewalls, but both protocols success is dependent on the particular configuration of the Firewall/IPS.

Before moving on, we note one further class of middlebox which is not common in end-user networks, but data centers: load balancers. If a load balancer is configured to direct traffic from a single client over multiple servers (rather than 'pinning' individual clients to servers), this too will break 2CFEC. However, as data centers are typically under a single administrative domain, it seems unlikely that an administrator choosing to deploy 2CFEC on their servers would leave their firewalls configured

[3]This technique is also proposed by Flach et al. [6]

[4]The only evidence for this type of mapping found inconsistent IP addresses between connections for <1% of clients over the course of multiple experiments, but the authors were unable to distinguish whether these inconsistent IP addresses were from a NAT with multiple IP addresses or if it was due to user mobility. [10]

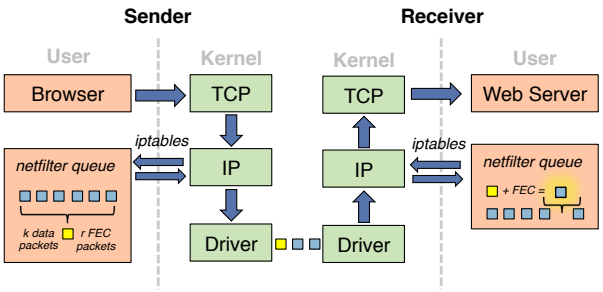


Fig. 5: Architecture diagram.

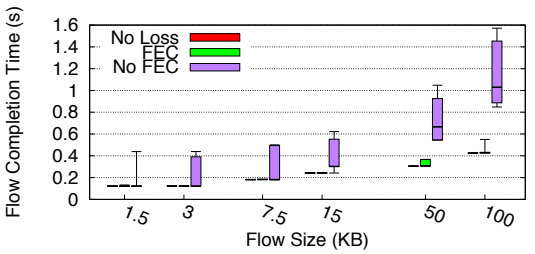
so as to be incompatible. This contrasts to the end-user network scenario, where administration for end hosts and network middleboxes is managed by separate parties and thus the trouble of incompatibility.

3.3 Implementation & Evaluation

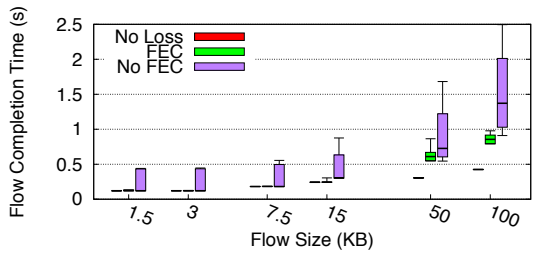
We built a prototype of 2CFEC to test its expected improvements in FCT (§2). Both sender and receiver use `netfilter_queue` to manipulate the packet stream. `netfilter_queue` (combined with iptables) captures packets upon arrival and enqueues them for inspection; from there the developer can use `netfilter_queue` to drop, inject, or modify packets before releasing them back to the protocol stack and up to the transport layer. Our code stores copies of the last k packets and uses these packets, in combination with redundancy packets, to re-inject missing packets and pass them up to the TCP layer. Missing packets are passed up with an ECN flag set, in order not to interfere with TCP's congestion control. Redundancy packets are dropped once no longer useful. Figure 5 shows an architecture diagram of our prototype. Our current scheme encodes redundancy into the channel using Reed-Solomon codes.

Our primary goal in building our prototype was to see whether a two-channel FEC design could provide comparable performance to an idealized FEC implementation that recovers all lost packets; as in §2 we quantify gains from FEC both in terms of flow completion times.

We evaluate 2CFEC with microbenchmarks: the impact of 2CFEC on individual flow completion times, shown in Figures 6(a) and (b). For smaller flow sizes, flow completion times using 2CFEC are very close to the ideal at both 1 and 3% loss. For larger flow sizes (50-100KB), performance significantly improves over the baseline TCP, but fails to quite reach ideal performance in call cases. This results from the fact that in longer flows, the probability of multiple loss events coinciding – resulting in an unrecoverable loss – increases in probability. Overall, these results indicate that 2CFEC provides substantial gains – often near optimal – despite the potential overheads of the two-channel approach.



(a) 2% packet loss



(b) 5% packet loss

Fig. 6: 2CFEC FCTs compared with baseline TCP and with ideal performance. Testbed used 10Gbps BW/100ms RTT links; boxplots show 1-5-50-95-99 percentiles.

4 Discussion

As we have shown, our dual-channel FEC approach can effectively work around many deployability issues imposed by middleboxes while still providing strong improvements in FCT compared to TCP. We now conclude with a discussion of 2CFEC and the dual-channel approach.

Does 2CFEC incur bandwidth overhead?
Like all FEC schemes, 2CFEC introduces some bandwidth overhead. 2CFEC's overhead comes from both the FEC encoding itself (which all designs will introduce), as well as additional control overhead such as the connection association and plug packets (due to the two-channel approach and our design specifically). Nonetheless, 2CFEC's overhead is reasonable; assuming a 10 KB flow, the bandwidth overhead over the single-channel approach is only around 2%.

Does the dual-channel approach introduce design or implementation complexity?
Using two separate channels for a single flow implies that we need to be able to correlate the secondary channel with the primary channel. As we discussed, the IPID field is today a reliable identifier, but may not remain so for the future. For this reason, we include additional data which may be aesthetically 'bulky', but is necessary to ensure channel association succeeds whenever possible.

In terms of implementation complexity, the dual-channel approach when combined with FEC was fairly clean. As the primary channel retains its original func-

tionality, the implementation of secondary channel can be completely decoupled to the primary one. In our implementation, 2CFEC did not require any modifications to the original TCP sender/receiver code. As compared to the single-channel approach, our 2CFEC implementation can be readily applicable to any TCP variants, or even UDP-based applications. This may not be the case for all dual-channel designs, but in our case, the decoupled design made implementation easier rather than more difficult.

Is the dual-channel approach completely middlebox-friendly?
No. In our analysis in 3.2 we show that there remain a few situations (inconsistent load balancers, proxies, and complete port rewriting) where 2CFEC must fail back to normal TCP. To the best of our knowledge, there are no situations where 2CFEC is middlebox-adverse.

We note that "terminal" proxies (those which masquerade as a connection endpoint to client and server) present both a challenge and an *opportunity* to the deployment of new protocols in general. A proxy which does not support a new protocol effectively blocks use of a new protocol, but a proxy which *does* support a new protocol can enable its use for one endpoint even when the other endpoint does not support it. For example, a network administrator whose end hosts support 2CFEC may choose to deploy a proxy which supports 2CFEC in order to provide loss recovery for packets which are lost at the edge.

More commonly though, middleboxes remain a challenge using all known design approaches. While the dual-channel approach may be effective today, nothing prevents future middleboxes from introducing new changes with new negative consequences for protocol design. We still believe that it is important to continue discussion of how to design for today's deployment scenarios. However, it is likely that the only path to return to the originial flexibility of the Internet may be to improve design processes to include protocol designers in the decision making within middlebox design and network administration.

Can we apply the dual-channel design approach to other extensions?
By the definition of the dual-channel approach, the primary channel must retain its TCP byte-stream semantics. Extensions that require modifications to the original data, such as compression of the TCP payload, cannot benefit from the dual-channel approach.

However, we expect that many TCP extensions can be implemented in a more middlebox-friendly way with the dual-channel approach. Flows which simply need to attach more control data can follow the same blueprint as our FEC design: establishing a secondary channel, performing connection association using the same identifying bits as our design, and then transmitting protocol-specific data over the secondary channel. We are currently exploring designs for SACK (where selective ACK values are sent over the secondary channel rather than the TCP header), and a multipath TCP design.

References

[1] *Personal Comm., Gregory Detal. 19 Jun 2013*.

[2] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky. Overclocking the Yahoo!: CDN for faster web page loads. In *Proc. IMC*, 2011.

[3] L. Baldantoni, H. Lundqvist, and G. Karlsson. Adaptive end-to-end FEC for improving TCP performance over wireless links. In *IEEE Communications*, volume 7, pages 4023–4027, 2004.

[4] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Denf, J. Benitez, U. Michel, H. Damker, K. Ogaki, T. Matsuzaki, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. Lopez, F. J. R. Salguero, F. Ruhl, and P. Sen. Network Functions Virtualization: An Introduction, Benefits, Enablers, Challenges & Call for Action. `http://www.tid.es/es/Documents/NFV_White_PaperV2.pdf`, Oct. 2012.

[5] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing Middlebox Intererence with Tracebox. In *Proc. IMC*, 2013.

[6] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *Proc. SIGCOMM*, 2013.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. IP Options are Not an Option. UC Berkeley Technical Report No. UCB/EECS-2005-24.

[8] R. Gallager. Low-Density Parity-Check Codes. *Transactions on Information Theory*, 8(1):21–28, 1962.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greengalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

[10] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating The Edge Network. In *Proc. IMC*, 2010.

[11] G. Maier, F. Schneider, and A. Feldman. NAT usage in Residential Broadband Networks. In *Proc. PAM*, 2011.

[12] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[13] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proc. ACM HotNets*, 2010.

[14] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[15] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, 2012.

[16] P. Srisuresh. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128.

[17] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proc. SIGCOMM*, 2011.

[18] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk. Port Control Protocol (PCP). RFC 6887.