Designing a Resource Pooling Transport Protocol

Michio Honda*
Keio University
micchie@sfc.keio.ac.jp

Elena Balandina
Nokia Research Center
elena.balandina@nokia.com

Pasi Sarolahti
Nokia Research Center
pasi.sarolahti@nokia.com

Lars Eggert
Nokia Research Center
lars.eggert@nokia.com

***Abstract*— This paper presents a design for an end-to-end transport protocol for multi-homed end systems that pools the communication resources of multiple network paths to support a single communication session. This approach offers improved performance and resilience compared to communicating over a single path. Compared to previous efforts, deployability in the current commercial Internet, *i.e.*, in the presence of middleboxes, filtering and restricted connectivity, was a key driver for the design of the Resource Pooling Protocol (RPP).**

I. Introduction

Efficient, lightweight and scalable allocation of network resources is among the key requirement for a future Internet. Resource pooling [1] has recently been proposed as a principle by which an end-to-end transport architecture can simultaneously utilize multiple, distinct paths between two communicating end systems in a way that will result in improved resource allocation.

The theoretical foundation for resource pooling was laid in earlier work by others, including [2]–[4], and has demonstrated significant performance and reliability improvements for end systems, as well as significant traffic engineering benefits for the network. For end systems, these benefits derive from the ability to pool the bandwidth and reliability offered by all their network interfaces to support each individual end-to-end path, compared to using only the network resources along a single path for each. For the network, the benefit comes from a much finer-grained traffic engineering control loop, which executes at timescales of network round-trips and operates at the flow level, compared to traditional traffic engineering mechanisms that operate on longer timescales and on coarser traffic aggregates.

The majority of bytes on the Internet are being transmitted over TCP. Compared to about ten years ago, the fraction of longer-lived TCP flows has been increasing, likely fueled by the widespread use of video streaming and peer-to-peer file sharing [5], [6]. This development is important, because the resource allocation provided by TCP's congestion control only becomes fully effective for connections that reach steady-state, which short connections rarely do. It also illustrates that a resource pooling protocol should be designed as a TCP extension that is transparent to applications and the network, if it is to have any chance at deployment. Hence, an RPP connection exposes the same communication primitive and API to applications as TCP, and its transmissions look like a set of TCP connections to the network.

During the same ten years, more and more end systems have shipped with multiple network interfaces. Most notebooks ship at least with Ethernet and WLAN interfaces, and many handhelds and mobile phones offer several different wireless interfaces, such as WLAN and 3G. This means that a significant fraction of current end systems already has the necessary equipment to benefit from resource pooling transport protocols.

This paper is by no means the first to describe a transport protocol based on multipath transmission. pTCP [7], mTCP [8], cTCP [9], AMS [10] and MPLOT [11], among others, extend TCP for multipath communication, and CMT [12] extends SCTP. The design of RPP presented in this paper has many of the same mechanisms explored in these earlier designs. The key difference is that the design of RPP is driven by the deployment necessities of today's commercial Internet, *i.e.*, the presence of middleboxes, filtering and otherwise restricted connectivity. The design is also as minimal as possible, in order to be tractable for eventual IETF standardization.

This paper discusses these design requirements and presents the rationale behind the key features of RPP. Due to space constraints, companion papers will describe the prototype implementations of RPP under development and present an experimental evaluation of RPP.

II. Design Overview

The design of RPP was driven by the desire for widespread deployability. This section discusses the de-

* Michio Honda was as an intern at Nokia Research Center in Finland during most of the duration of this research effort.

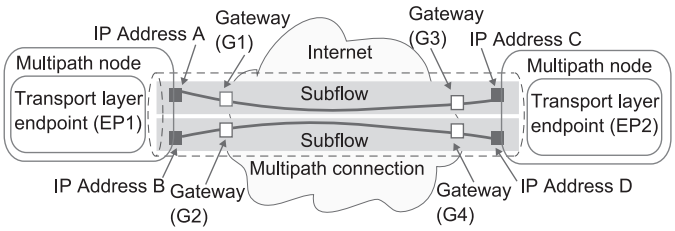

Fig. 1. One RPP connection consisting of two subflows.

tails of this overarching requirement and how it relates to applications, backwards compatibility and middleboxes.

TCP provides a communication primitive to applications that is a reliable and ordered byte stream. To be a drop-in replacement for TCP—the protocol carrying the majority of Internet bytes—RPP must provide exactly the same communication primitive through exactly the same API (with possible optional extensions [13]).

RPP will be deployed incrementally. This means that RPP end systems will need to be able to communicate with end systems that implement standard TCP. This could be accomplished by designing RPP as a standalone protocol and determine at connection start whether to use TCP or RPP. However, using a separate IP protocol number for RPP has the serious drawback of being incompatible with many deployed firewalls, NATs and other middleboxes. SCTP and DCCP suffer from this issue, and to this day have therefore failed to see much deployment. RPP has consequently been designed as a strictly modular extension to standard TCP, reusing the same IP protocol number. RPP implementations will try to negotiate the use of RPP during the SYN/ACK exchange and fall back to standard unicast TCP transmission when this negotiation fails.

Figure 1 illustrates the components of the RPP design by showing one RPP *connection* between endpoints EP1 and EP2. EP1 and EP2 are multi-homed, each having two network interfaces with two different IP addresses (A and B, and C and D). The next-hop routers for each of those interfaces are labeled G1-4. Hence, four distinct paths exist between EP1 and EP2: A-C, A-D, B-C and B-D. The connection shown uses only the two paths A-C and B-D through two *subflows* transmitting along them.

An RPP *connection* is the entity over which applications communicate. It provides a reliable and ordered byte stream that has exactly the same semantics as a TCP connection. An RPP connection transmits application data over one or more *subflows*. It is up to the implementation to schedule which chunk of application data is transmitted over which subflow or subflows. The receiving end must read chunks of application data from all subflows and deliver them in order. An RPP connection is implicitly created when its first subflow is created and terminated when its last subflow terminates.

An RPP *subflow* is the entity over which RPP transmits chunks of application data along a single path. In order to maximize the chances of a subflow to traverse any middleboxes along "its" path, it is important that a subflow resembles a single TCP connection as closely as possible. Hence, a subflow is established through a standard three-way SYN exchange with a piggybacked new TCP option used to negotiate the use of RPP, similar to how use of many other TCP extensions is negotiated. After the initial handshake, every subflow packet contains a regular TCP segment.

When the use of RPP has been negotiated for a connection between two end systems, they start exchanging data across multiple distinct paths, using one subflow along each such path that is independently congestion controlled. Initially, RPP congestion control uses standard TCP mechanisms for each subflow; more advanced congestion control is one topic of further research.

III. Design Details

This section discusses the key design choices for RPP in more detail, including connection identification, subflow termination, path discovery, sequence number spaces, transmission scheduling and congestion control.

A. Connection Identification

An RPP connection is created when its first subflow has been established. If one end wants to create another subflow for that same connection, it needs to indicate this to the peer, so that the peer can differentiate between a subflow SYN for a new connection vs. one for a new subflow that is joining an existing connection. This can be done in different ways, and there are subtle tradeoffs.

One approach is to identify a connection for joining by referring to one of its existing subflows. TCP identifies a connection by a four-tuple consisting of source and destination IP addresses and ports, and the four-tuple of an existing subflow could be used to identify the connection that a new subflow is joining. In practice, however, this has the drawback that because NATs can rewrite source IP addresses and port numbers, both ends may refer to the same subflow with different identifiers.

A second approach is to identify connections through a shared, probabilistically unique *initial sequence number* (ISN), which is done by AMS [10]. Instead of using the four-tuple to identify a subflow, AMS re-uses the ISN of

a subflow that is part of the desired existing connection when sending a SYN for a new subflow. Unfortunately, some middleboxes are known to rewrite sequence numbers, again rendering this technique problematic from a deployment perspective.

A third approach, introduced by pTCP [7] and adopted by RPP, creates an explicit new namespace for connections that does not reuse existing header fields and is hence safe from modification by NATs. RPP exchanges these *connection IDs* inside TCP options piggybacked onto the SYN exchange. This approach has the downside of introducing a new TCP option, which decreases the success rate of subflow establishment in a minor way (by 0.3% [14]). It is still possible to fall back to standard TCP in these rare cases, so communication will not be prevented. It is also probable that as RPP gains adoption, middleboxes will be updated to pass the new option.

Under this third approach, the SYN and SYN/ACK for the first subflow of an RPP connection contain a new connection ID chosen by the sender. SYN and SYN/ACKs for additional subflows for the same RPP connection contain the same connection ID as the first subflow. pTCP inserts connection identifiers into all packets. This is redundant, because after a subflow is established, both ends know which connection it belongs to. RPP only includes connection IDs in the SYN and SYN/ACK, in order to conserve option space.

B. Subflow Identification

All subflows should normally be explicitly terminated by exchanging FIN packets, in order to allow any middleboxes to free up state. However, this is not always possible, *e.g.*, when one end abruptly loses connectivity on a network interface.

It is straightforward for the end to stop transmission over the subflows affected by the outage. However, the peer cannot do the same, and will likely continue to try using the defunct subflows. This can degrade performance, because application data transmitted over defunct subflows is lost and will require retransmission, causing delays until the peer eventually stops using the defunct subflows due to timeouts.

Consequently, it is a useful optimization to allow one end to notify its peer about the termination of one subflow over a *different* subflow belonging to the same connection. It is important to note that such subflow termination is an optional optimization—loss of connectivity can occur anywhere along a path, and RPP must correctly deal with this situation even when it cannot detect connectivity status directly. The TCP timeout mechanisms that RPP inherits provide the necessary timeout mechanisms to eventually tear down subflows.

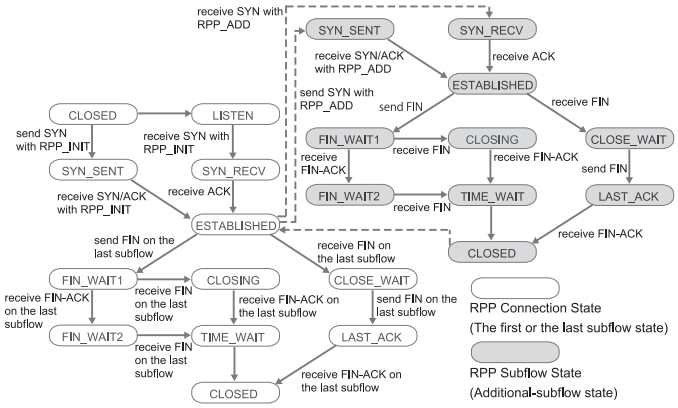

Fig. 2. The RPP state machine.

Subflow termination requires a *subflow ID*, in order to allow identification of a defunct subflow. Similar to the connection ID discussed above, four-tuples are unsuitable as identifiers. RPP defines a namespace for *subflow IDs* for this purpose and includes them during the SYN exchange of each subflow alongside its connection ID.

Figure 2 ties together RPP connection and subflow management into one state machine. An RPP connection has the same states as a TCP connection. A connection ID is exchanged during the SYN handshake of the *initial* subflow of a connection, using the RPP_INIT option. Subflow IDs are exchanged during the SYN handshakes of any *additional* subflows that are joining an existing connection, using the RPP_ADD option. Subflow states are also generally similar to those of a TCP connection. An RPP connection is established when its first subflow is established. As long as at least one subflow remains in the ESTABLISHED state, the entire connection remains there, because it can still make progress. The connection is closed when its last subflow is closed, which happens either via a FIN exchange or via a timeout event.

It is worth noting that another design possibility for subflow termination exists. It assumes that all subflows transmitted across the same network interface will be affected by loss of interface connectivity. Subflows are tagged with an interface or address ID, and when connectivity is lost, all subflows sharing an interface are terminated. RPP did not choose this variant, because subflow IDs are no more complicated, but allow finer-grained per-subflow operations.

C. Path Discovery and Subflow Establishment

The benefits of resource pooling become stronger with an increasing number of distinct paths that are being used to support a single end-to-end connection. It is hence important for RPP to be able to discover as many potential candidate paths as possible. Equally important are techniques to establish subflows across those candidate paths even when they suffer from restricted connectivity. Some end systems might want to use only specific network interfaces or paths, hence advanced APIs and administrative parameters *e.g.*, *sysctl* will also support such operation in future work.

The current design of RPP focuses on multi-homed end systems and assumes that subflows between different source and destination IP addresses will be routed along different paths. This means that candidate paths can be discovered by exposing all available local IP addresses to the peer, and keeping track of which addresses the peer exposes. It is a design goal to keep RPP independent of IP versions; it should be possible to create an RPP connection that has some subflows traversing an IPv6 network while others traverse the IPv4 Internet.

The path discovery process begins at the initiator of a connection, *i.e.*, the end that sends the SYN of the first subflow. The initiator knows about its local IP addresses, and it knows at least one IP address of the peer. *e.g.*, via *getaddrinfo()*. If the initiator in Figure 1 is EP1 and the initial peer address it knows is C, it can for example initiate the subflow A-C, creating the RPP connection. After that, based on its local IP addresses, it can also initiate subflow B-C. At this point, EP1 has discovered all the paths it can, based on its current information, and it has also exposed all of its local addresses to the peer by opening one subflow from each. EP2, however, has local address D that is not yet being used for the connection and it can hence create subflows D-A and D-B.

The general principle is that the two ends of a connection establish new subflows whenever they learn about a new local or remote address that results in the availability of new path candidates. In other words, information about endpoint addresses is implicitly shared by establishing subflows to or from them. SCTP, in contrast, has protocol messages that let the ends explicitly add and remove IP addresses from the peer's address set. RPP shares addresses implicitly, because successful establishment of a subflow tests connectivity—the peer never learns about addresses that have connectivity issues.

As described above, subflows are established with a normal SYN handshake. However, middleboxes often restrict connectivity by dropping "inbound" SYNs. For example, if G1 in Figure 1 is a NAT, EP2's attempt to create the D-A subflow will fail, because G1 will drop the inbound SYN. This reduces the number of subflows of the RPP connection, which in turn limits performance and reliability. This issue depends on the direction of connection establishment—establishment of a subflow from A to D across G1 will succeed.

RPP consequently offers the feature to also reverse the direction of subflow establishment along a path. This is done by notifying the peer of an IP address to make the peer send a SYN to the address. This IP address is notified by using protocol option inside packets transmitted on a different, already established subflow. In the example, EP2 would include IP address D in such a " SYN requested" option transmitted on the A-C subflow, causing EP1 to send a SYN from A to D. In general, RPP has no way of knowing if a middlebox exists along a path, and when one end attempts to create a new subflow, it will *both* send a SYN *and* piggyback a "SYN requested" option on another subflow, if possible.

Finally, RPP provides a "hole-punching" mechanism [15] to establish a subflow along a path where *both* endpoint addresses are behind middleboxes. This mechanism requires at least one end to have at least one globally reachable address, from which it can create an initial subflow. In the example in Figure 1, suppose that G2 and G4 are NATs. The B-D subflow cannot be established by any of the above techniques. In order to create it, EP1 sends a SYN from B to C, causing G2 to create a binding entry for B and the chosen source port. EP2 replies with a SYN/ACK from D (instead of C), causing G4 to create a binding entry for D and the chosen source port. EP1 sends the final ACK to the address it received the SYN/ACK from, and because G4 a binding entry for this address and port, it forwards it to D.

Mechanisms by which single-homed hosts could exploit the availability of multipath routing deeper inside the network for resource pooling are currently being investigated [16].

D. Sequence Number Spaces

TCP detects loss events based on its sequence numbers by observing holes in the acknowledgment stream. Loss events drive both TCP's reliability (retransmission) mechanisms as well as its congestion control mechanisms. RPP connections also provide a reliable and ordered byte stream, but congestion control happens per subflow, because it needs to be based on the measured characteristics of each path. RPP consequently teases

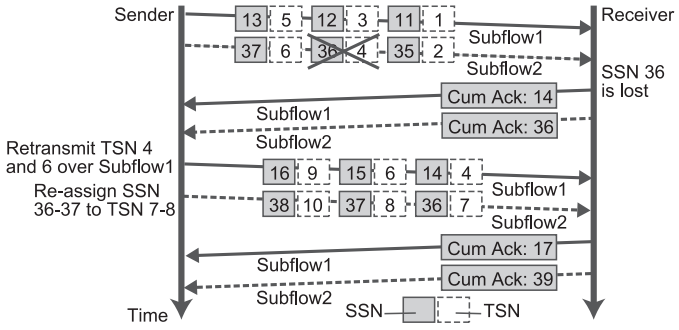

Fig. 3. One RPP connection sending over two subflows, showing transport sequence numbers and subflow sequence numbers.

apart reliability and congestion control, resulting in two sequence number spaces, the connection-level *transport sequence number* (TSN) and the *subflow sequence number* (SSN). RPP transmits TSNs using a new TCP option included with each segment; SSNs are carried in the sequence number field in the TCP header.

Each TSN identifies one unique byte of application data transmitted over the RPP connection. In other words, the TSN establishes the order in which received data is delivered. SSNs, on the other hand, have no meaning at the connection level. They are purely subflow-specific, and have the purpose of allowing loss detection. Detected losses will affect the congestion control state of the subflow on which they were detected. The ACK packets of individual subflows contain SSN information but no TSN information. The sender is responsible for maintaining a mapping of which application bytes (which TSNs) it transmitted inside which subflow bytes (which SSNs). When data loss is detected for a subflow, the sender determines which application data was lost using this mapping. Application data lost during transmission along one subflow can be retransmitted over a different subflow. In this case, the SSN of the original subflow is adjusted accordingly.

Figure 3 illustrates how the two sequence number spaces play together for an example consisting of two subflows of one RPP connection transmitting application data segments with TSNs 1–10. Subflow 1 carries TSNs 1, 3 and 5 in subflow segments with SSNs 11, 12, and 13. Subflow 2 carries TSNs 2, 4 and 6 in subflow segments 35, 36, and 37. SSN 36 (carrying TSN 4) is lost. The connection retransmits TSN 4 over subflow 1 in SSN 14 and *reuses* SSNs 36 and 37 on subflow 2 for transmitting TSNs 7 and 8. Another design option under investigation is to not reuse SSNs in this way, but instead transmit gratuitous "get over it" ACKs on the original subflow for TSNs successfully retransmitted over another subflow.

The design choice of two sequence number spaces is different from CMT [12], mTCP [8] and cTCP [9], which only employ connection-level sequence numbers. Our choice was motivated by several reasons. First, in the presence of middleboxes, the sequence numbers of different subflows can be rewritten, causing the receiver to be unable to reassemble the application data received on different subflows. Second, a data stream transmitted along several paths is likely to have many more chunks arriving out-of-order than one transmitted along a single path. These reorderings are indistinguishable from loss events, causing issues for standard TCP congestion control. Even when selective acknowledgments (SACK) [17] are used, frequent reorderings will necessitate transmitting large SACK blocks, which is problematic due to limited option space.

E. Transmission Scheduling

Because an RPP connection has the same semantics and the same API as a TCP connection, applications interact with it through a single socket and socket buffer. A *transmission scheduler* is responsible for determining which chunk of application data should be transmitted over which subflow. These scheduling decisions affect flow control and might cause performance issues.

On one hand, a sender should transmit data in the send buffer as soon as possible, to minimize delay and maximize utilization. On the other hand, different subflows have different RTTs, and data transmitted along one subflow may arrive significantly later than data transmitted over another. Because RPP provides a reliable byte stream, data received "early" cannot be delivered to the application and must be buffered. A bad transmission scheduler can hence exhaust the available receive buffer with out-of-order data, degrading performance and even requiring additional retransmissions. A bad transmission scheduler will also tie up more memory for the send buffer, because transmitted data cannot be freed until all gaps in the acknowledgment space have been filled.

The design of a transmission scheduler that efficiently transmits application data over multiple subflows in a way that maximizes throughput and minimizes delay is future work.

F. Congestion Control

Each subflow of an RPP connection traverses a different network path with different characteristics. Congestion control, which adapts network transmissions to what

a given path can sustain, hence needs to happen on a per-subflow basis. The current design of RPP is targeted at multi-homed end systems, and assumes that subflows between unique source and destination IP address pairs will not have a shared bottleneck.

It is not clear if this assumption will hold in all cases. For example, with an over-provisioned core, a network interface at the end systems can become the shared path bottleneck for all subflows using it. Another example occurs when the paths of multiple subflows between unique source and destination IP address converge in the core of the network and are routed together.

When several subflows of one connection share a bottleneck, their resource consumption adds up. Each subflow is as aggressive as a single TCP connection, and a bundle of n TCP-friendly subflows will hence use an approximately n times greater share of the bottleneck resource than they should. This behavior can lead to unfairness, when multipath connections with large n or many multipath connections compete with a smaller number of regular TCP connections.

Two approaches exist to mitigate these issues. First, detection mechanisms for shared bottlenecks [18], [19] allow suppressing transmissions on some of the subflows sharing the bottleneck. mTCP [8] adopts this approach and finds that it can take around 15 seconds, during which significant unfairness could occur. A second approach is to use some form of integrated congestion control for all the subflows belonging to an RPP connection, along the lines of [20]. This latter approach has been explored in more detail for use with RPP [21].

IV. Conclusion

This paper has presented the design of RPP, a transport protocol that pools transmission resources along multiple Internet paths in order to support one end-to-end connection. Compared to previous proposals, the design of RPP is driven by the deployment necessities of today's commercial Internet, *i.e.*, the presence of middleboxes, filtering and otherwise restricted connectivity. The design is also as minimal as possible, in order to be tractable for eventual IETF standardization. An implementation of RPP for Linux 2.6 is currently underway, and companion papers will describe the details of the prototype implementation as well as experimental results. A second implementation of some key features of RPP has been completed for the ns2 network simulator.

Acknowledgment

This research effort was partly funded by *Trilogy* [16], a research project supported by the European Commission under its Seventh Framework Program. This work is partially supported by JST, CREST.

References

[1] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *ACM CCR*, vol. 38, no. 5, pp. 47–52, 2008.

[2] F. Kelly, "Loss networks," *Annals of Applied Probability*, vol. 1, no. 3, pp. 319–378, 1991.

[3] F. Kelly and T. Voice, "Stability of end-to-end algorithms for joint routing and rate control," *ACM CCR*, vol. 35, no. 2, pp. 5–12, 2005.

[4] P. Key, L. Massoulie, and P. Towsley, "Path selection and multipath congestion control," *Proc. IEEE INFOCOM*, pp. 143–151, May 2007.

[5] K. Thompson, G. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network*, vol. 11, no. 6, pp. 10–23, Nov/Dec 1997.

[6] W. John and S. Tafvelin, "Analysis of Internet backbone traffic and header anomalies observed," *Proc. ACM IMC*, pp. 111–116, 2007.

[7] H.-Y. Hsieh and R. Sivakumar, "A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts," *Proc. ACM MobiCom*, pp. 83–94, 2002.

[8] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang, "A transport layer approach for improving end-to-end performance and robustness using redundant paths," *Proc. USENIX ATC*, pp. 99–112, 2004.

[9] Y. Dong, N. Pissinou, and J. Wang, "Concurrency handling in TCP," *Proc. IEEE CNSR*, pp. 255–262, May 2007.

[10] S. Saito, Y. Tanaka, M. Kunishi, Y. Nishida, and F. Teraoka, "AMS: An adaptive TCP bandwidth aggregation mechanism for multi-homed mobile hosts," *IEICE Trans. Inf. Syst.*, vol. E89-D, no. 12, pp. 2838–2847, 2006.

[11] V. Sharma, S. Kalyanaraman, K. Kar, K. Ramakrishnan, and V. Subramanian, "MPLOT: A transport protocol exploiting multipath diversity using erasure codes," *Proc. IEEE INFOCOM*, pp. 121–125, April 2008.

[12] J. R. Iyengar, P. D. Amer, and R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," *IEEE/ACM ToN*, vol. 14, no. 5, pp. 951–964, 2006.

[13] L. Eggert and W. M. Eddy, "Towards more expressive transport-layer interfaces," *Proc. ACM MobiArch*, pp. 71–74, 2006.

[14] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the Internet," *ACM CCR*, vol. 35, no. 2, pp. 37–52, 2005.

[15] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," *Proc. USENIX ATC*, pp. 13–13, 2005.

[16] Trilogy Project, http://trilogy-project.org/.

[17] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," *RFC 2018*, Oct. 1996.

[18] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting shared congestion of flows via end-to-end measurement," *IEEE/ACM ToN*, vol. 10, no. 3, pp. 381–395, 2002.

[19] D. Katabi, I. Bazzi, and X. Yang, "An information theoretic approach for shared bottleneck inference based on end-to-end measurements," *MIT-LCS Technical Memo 604*, 1999.

[20] L. Eggert, J. Heidemann, and J. Touch, "Effects of ensemble-TCP," *ACM CCR*, vol. 30, no. 1, pp. 15–29, 2000.

[21] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda, "Multipath congestion control for shared bottleneck," *Proc. PFLDNeT*, May. 2009, (to appear).