Towards a Multipath Transport Protocol for Serval

Torin Rudeen
Department of Computer Science, Princeton University
Under the advisement of Michael J. Freedman

Abstract

Multipath transport protocols provide practical and theoretical advantages over traditional single path protocols, enabling better congestion balancing and higher throughput without violating TCP fairness conditions. Multipath TCP (MPTCP) is the main ongoing effort to create a multipath transport protocol, but its efficiency and flexibility is hampered by the need to be entirely backwards compatible with a network designed for single path connections. What would a multipath transport protocol look like if it could be created without these restrictions?

Serval is an experimental network architecture designed to support service-centric networking. It supports seamless mobility across networks and interfaces, multipath communication, and a service-centered network abstraction by creating a Service Access Layer which sits on top of an unmodified Network Layer and below the Transport Layer. In this paper I draw on the ongoing research on MPTCP to design and implement a multipath transport protocol that takes advantage of the abstractions provided by Serval to offer improved efficiency, flexibility, and simplicity, while still playing fair with legacy TCP connections.

1 "Worse is Better" and the Need for Multipath

In his widely read article *The Rise of "Worse is Better"* Richard Gabriel offered a novel theory of why technically imperfect software systems often gain traction and widespread acceptance, even where more sophisticated systems fail [4]. Consider two systems: the first is simple to implement, but sacrifices completeness, consistency, and sometimes even correctness in service of this simplicity. The second is created to be completely correct, consistent, and complete, but sacrifices simplicity to this end. The first system (using what Gabriel calls the "New Jersey approach") is considerably "worse" than the second (the "MIT approach"): it is unreliable, limited in what it can do, and imposes extra burdens on the user. However, it will be created faster, ported more easily, and by the time the second system has been perfected the first will already have been widely adopted. Thus, Gabriel argues, the New Jersey approach will generally create more successful technologies than the MIT approach. C is a classic example of the New Jersey approach; Lisp is an example of the MIT approach. C was so successful not because it was technically superior to Lisp, but because it was much easier to write compilers for.

The Internet is one of the best examples of the New Jersey approach to design. Crucial to the widespread adoption of the Internet is the simplicity of the core protocols. Each piece of functionality is shunted as far up the network stack as possible: for example, instead of relying on the Link Layer or the Network Layer to guarantee reliable delivery, it's left to the Transport Layer. While this may be inefficient, because packets must be sent across the entire network when a loss occurs on any link, it makes switches and routers far simpler and cheaper to implement. Similarly, instead of explicitly allocating bandwidth to each connection, as in telephone networks, the Internet uses statistical multiplexing, relying on Transport level congestion avoidance algorithms to ensure an approximately fair allocation of resources. In fact, the well known end-to-end principle[13] can be viewed as an application of "Worse is Better" to networking: lower layers in the network stack should be as simple as possible, leaving all possible functionality to the higher layers.

1.1 The Resource Pooling Principle

While the "Worse is Better" approach may explain the Internet's widespread adoption, modern web applications demand reliablity, consistency, and scalability that the TCP/IP stack cannot provide on its own. This need has given rise to a wide array of technologies that address these issues from the Application Layer, including load balancers, network address translators (NATs), and content distribution networks (CDNs). For each of these systems, all or part of their functionality consists of taking care of problems which would have been addressed within the network, had the Internet had been designed with the MIT approach in mind. Now, this is not necessarily a bad thing: many of these technologies address needs which could not have been forseen when TCP/IP was designed, and solving these problems at the Application Layer allows for flexibility and quick deployment.

However, as Wischik argues in [14], these technologies all use variations of a single technique, *resource pooling*. The basic idea is to make multiple heterogenous resources behave like a larger single resource, for improved efficiency,

capacity, and reliability. For example, a load balancer spreads traffic across a number of servers, providing the abstraction of a single server with much greater procesessing power and bandwidth. The problem is that when this technique is implemented over and over in every part of the network, massive redundancy and complexity is created. Wischik states the problem as his *resource pooling principle*: "Resource pooling is such a powerful tool that designers at every part of the network will attempt to build their own load-shifting mechanisms. A network architecture is effective overall, only if these mechanisms do not conflict with each other."[14]

Wischik's key insight that motivates the development of a multipath transport protocol is this: many forms of resource pooling currently performed by diverse Application Layer technologies can be handled elegantly and efficiently by a multipath-capable Transport Layer. When a single connection can simultaneously utilize multiple links, it is effectively pooling the capacity of all of these links. If one link fails, traffic can simply be sent over the others. When one link is heavily congested, instead of backing off transmission altogether, more traffic can be shifted to uncongested links. By having two paths go through different ISPs, one can even reap the benefits of multihoming without putting pressure on BGP.

2 Multipath TCP

For the past several years, there has been an ongoing effort to create a multipath transport protocol that can achieve resource pooling in the Transport Layer. The result of this effort is Multipath TCP (MPTCP). Extensive research has gone into designing a congestion control algorithm for MPTCP that balances congestion and improves throughput while still being fair to legacy TCP connections[12][15][16]. Furthermore, MPTCP has been shown to work in practice: there is an open source implementation of MPTCP in the Linux Kernel[11], and MPTCP is undergoing standardization by an IETF working group[6].

MPTCP is designed with the primary goal of backwards compatibility with a network designed around single path connections[6]. This goal dictates many of the design choices around MPTCP. For example, MPTCP uses a layered approach to sequence numbers, where every packet is assigned a flow-level sequence number and a connection-level sequence number. Flow sequence numbers are put in the TCP header, while a mapping from the per-flow sequence number space to the connection-level sequence number space is established using TCP options[11]. This makes it so that a single flow of an MPTCP connection appears to be a legacy TCP connection from the perspective of the network, ensuring compatibility with middleboxes. However, this comes at the cost of additional per-flow state, increased complexity of the protocol, and reduced flexibility.

These and other such tradeoffs are fact a result of the "Worse is Better" design of the Internet. Because the network chooses simplicity over providing desired functionality, users (individuals, corporations, and designers) take it upon themselves to create this functionality in a variety of ways, often violating the end-to-end argument and the layering abstraction. This is the basic reason for the proliferation of middleboxes, and for the hoops that MPTCP has to jump through in order to create a protocol deployable on today's Internet.

There is undeniable value in creating a multipath transport protocol that works in the hostile environment of today's Internet. However, there is also merit in asking what a multipath TCP look like if it were not subject to the constraints imposed by backwards compatibility. What gains could be made if a transport protocol could be designed on top of a more flexible network abstraction? In this project I have set out to answer just this question, building on top of the Serval network architecture in the hopes of creating a simpler, more efficient, and more flexible multipath transport protocol.

3 Overview of Serval

The network stack, in particular the TCP and IP protocols, was designed around a host-centric abstraction, optimized for an Internet of immobile computers which use only a single network interface. This is a far cry from today's Internet, where computers are mobile, possess multiple heterogenous network interfaces, and wish to connect to web services which are split across thousands of computers around the world.

For example, IP routes packets based on their IP address, implicitly relying on the assumption that each party in communication is uniquely identified by a single IP address that does not change over time. Unfortunately, each component of this assumption fails in today's Internet: IP addresses do not uniquely a party, due to the paucity of (IPv4) addresses and the resulting prevalance of NATs; a single party often has more than one IP address, whether because it is a single device with multiple interfaces (i.e. a smartphone with 4G and WiFi), or because it is a large web service run on thousands of different machines (i.e. Google); and the IP address of one party can change over time, whether due to virtual machine migration or to physical mobility. A wide variety of methods are used to deal with these problems, ranging from sophisticated application layer technologies (i.e. NATS and load balancers) to awkward non-solutions (such as restarting the connection whenever a party to a new IP address).

Serval is one attempt to design a network architecture more in line with the needs of today's Internet. As described in [9], Serval replaces TCP/IP's host-centric abstraction with a new service-centric abstraction. A connection is a link between two services, identified by serviceIDs, each of which corresponds to one or more machines, each of which may have multiple interfaces. Each Serval connection is composed of one or more subflows, each of which is a point-

to-point link between two IP addresses. Instead of early-binding to an IP address using DNS, Serval-enabled hosts late-bind on serviceIDs, which are mapped to the IP address of a target machine using anycast routing. Once a connection is established, either host can add additional subflows or migrate existing subflows without disrupting the ongoing connection, enabling multipath communication and seamless migration. All of this is made possible by the creation of a new Service Access Layer (SAL), sitting between the Network and Transport Layers. The SAL handles resolutions of serviceIDs to IP addresses, connection establishment and termination, and subflow creation, deletion, and migration.

The SAL handles many of the thorny problems of multipath, exposing a clean and simple interface to the Transport Layer. The choice of when to add or delete subflows is made by the SAL (with input from a userspace controller). At any point in time, the Transport Layer has access to a list of available subflows, and can send data over any of them at will. The Transport Layer is notified whenever one of its subflows is being shutdown or migrated, or when a new subflow is available. Serval uses a formally verified protocol for subflow establishment, termination, and migration [2], meaning the Transport Layer need not concern itself with the many tricky issues of designing such a protocol.

4 Design

The Transport Layer is responsible for a bewildering array of functionality. Raiciu breaks up the functionality of TCP as follows: [11]

- Connection setup and state machine.
- Reliable transmission & acknowledgement of data.
- Congestion control.
- Flow control.
- Connection teardown handshake and state machine.

In fact, some researchers have suggested breaking up the functionality of the Transport Layer into several layers [3]. In effect, this is what Serval does: connection teardown and establishment are the responsibility of the SAL, leaving the Transport Layer to concern itself with reliable transmission and acknowledgements, congestion control, and flow control.

4.1 "Dumb Subflow, Smart Connection"

A fundamental question in designing a multipath transport protocol is how much complexity to place at the level of the individual subflow, and how much to place at the level of the connection as a whole. At one extreme, one could imagine a protocol that functions exactly like TCP, except instead of sending out data over a single subflow, it blindly stripes it across the available subflows. In essence, this protocol would be entirely connection-level, and would be completely oblivious to individual subflows. At another extreme, one could create a "protocol" that opens a TCP connection for each subflow, sends different blocks of data over each subflow, and then reassembles them at the other end. This protocol would be almost entirely subflow-level, with very little connection-level state. In fact, as we will see below, either extreme is insufficient: a subflow-oblivious protocol would perform very poorly in the presence of varying RTTs, while an entirely subflow-oriented protocol could not coordinate the congestion windows of different subflows.

While either extreme is impractical, the question remains: is it better to have complex, largely independent subflows loosely linked together with a minimum of connection-level state, or to have simple, dumb subflows with most state maintained at the connection level? For MPTCP, the answer was dictated by the demands of backwards compatibility: since MPTCP subflows must look like legacy TCP subflows in order to pass by middleboxes that assume single-path TCP, the MPTCP developers had no choice but to maintain essentially all the state of a single-path TCP connection for every subflow. However, I believe that in the absence of this constraint, the best approach is to keep per-subflow complexity to a minimum.

I call this the "Dumb Subflow, Smart Connection" principle. There are several justifications for this approach:

- Following from the end-to-end principle, any work performed at the subflow level will have to be repeated or checked at the connection level. For example, if data is reordered and buffered for each subflow, the data will have to be reordered and buffered again at the connection level, since each subflow only receives a portion of the data. Similarly, if a subflow does not receive an ACK for data sent, it needs to check with the connection before retransmitting, since the ACK may have been received on a different subflow. Since the work will have to be redone or checked anyway, it makes more sense to just do the work once at the connection level.
- Better decisions can be made at the connection level than can be made at the subflow level, since more information and more options are available. For example, if a subflow suddenly starts experience high rates of packet loss, the connection can respond by retransmitting lost packets over a different interface, alleviating congestion on the first subflow.
- This principle implies that data is not assigned to a particular subflow until the last possible moment, which allows maximum possible flexibility in the face of changing network conditions. For example, the connection could only assign data to a subflow when it knew the congestion window was large enough that the data could be immediately transmitted. This avoids the problem where data is stuck waiting in a subflow's send buffer while other subflows are ready to send.

In the following sections, I will show how this principle can be applied to develop mechanisms for flow control, congestion control, and reliable delivery.

4.2 Flow Control

This is the simplest mechanism to design. As demonstrated in [11], flow control must be orchestrated at the connection level; if individual send and receive buffers are used for each subflow, then deadlock can arise when a when a subflow loses connectivity with outstanding data. Given this, the design is clear: use a single send buffer and a single receive buffer for the connection, with semantics identical to TCP. This is also the solution adopted by MPTCP.

4.3 Congestion Control

Congestion control is perhaps the most complicated aspect of a multipath transport protocol. Fortunately, this area has been the focus of most research on MPTCP, and the fruits of this research apply equally well to my design.

4.3.1 Policy

Wischik et al., who articulated the resource pooling principle (Section 1.1), also identified one of the key challenges in multipath congestion control: they found that supposedly stable algorithms designed around fluid models of networks, such as those of Kelly et al. [7], exhibited extremely flappy behaviour in practice, quickly switching from using one subflow exclusively to using the other subflow exclusively [15]. Raiciu et al. expanded on this work to articulate three explicit goals for a multipath congestion control algorithm, which I repeat here: [12]

- **Goal 1: Improve throughput.** A multipath subflow should perform at least as well as a single-path flow on the best available subflow.
- **Goal 2: Do no harm.** A multipath subflow should not take up more capacity on any path than a single-path flow would.
- **Goal 3: Balance congestion.** A multipath subflow should move as much traffic as possible off of its most congested paths.

Goal 1 ensures that a multipath connection is never a downgrade from a single-path connection. Goal 2 ensures that a multipath connection is fair to single-path TCP connections. Goal 3 is what provides the benefits of resource pooling: by reacting to congestion by moving data to other subflows, the connection achieves higher resource utilization and lower loss rates, as well as making the network less congested for other, non-multipath flows sharing network resources.

At first blush, Goal 2 seems somewhat counterintuitive. If single-path TCP would get 1 GB/s of throughput over subflow 1, and 1 GB/s of throughput over subflow 2, it seems obvious that a multipath protocol should get 2 GB/s of throughput when using both. However, this is not desirable. In general, the Transport Layer cannot know to what extent two subflows overlap in their use of network resources. If subflows 1 and 2 overlapped at a point of congestion (perhaps the last-mile link to the server), a multipath connection that got 2 GB/s would be using up twice its fair share of the congested resource. Taken to its logical extreme, this approach would create a subflow "arms race", where users trying to maximize throughput would add more and more subflows to get a higher share of contested network resources, creating extra overhead for multipath users and leaving single-path TCP users with only a tiny sliver of resources left.

Raiciu et al. goes on to propose an algorithm that satisfies these goals, while avoiding flappiness [12]. Their Linked Increases Algorithm (LIA) is similar to TCP's congestion avoidance algorithm: for each subflow, the congestion window is increased by a small amount with each ACK, and cut in half with each loss. The difference from TCP is that the amount of increase is inversely proportionate to the *total* of all congestion windows in the connection, and proportional to a parameter α which is dynamically adjusted to satisfy Goals 1 and 2. The algorithm is further justified in [16], and implementation details, including suitably efficicient methods for calculating α, are outlined in RFC 6356 [10].

Some researches have called into question the optimality of LIA, leading to the development of an alternative algorithm, the Opportunistic Linked Increases Algorithm (OLIA) [8]. Time will tell which algorithm gains acceptance, but at the moment the LIA algorithm is well studied and well specified, and several reference implementations exist, so I will be using LIA in my design.

4.3.2 Mechanism

A congestion control algorithm specifies a congestion *policy*, but a *mechanism* is also required. In other words, LIA specifies a congestion window for each subflow, but the Transport Layer must guarantee that no more bytes enter the network than the congestion window allows. In TCP, the send buffer is used to enforce congestion control via the sliding window algorithm: the last byte that can be sent is dictated by the minimum of the send window and the congestion window. This approach obviously will not work with a multipath protocol, since a single send buffer is used for the connection, and there must be different congestion windows for each subflow.

MPTCP solves this problem by having a send buffer for each subflow, each with its own instantiation of the sliding window algorithm. In keeping with the "Dumb Subflow, Smart Connection" principle, I believe the best way to enforce congestion control is by a much simpler mechanism:

- Use a single send buffer for the connection.
- Maintain a variable `unacked_bytes` for each subflow.
- `unacked_bytes` increases when bytes are sent over a subflow, and decreases when bytes sent over a subflow are ACKed.

- Bytes are only sent over a subflow if doing so would not make `unacked_bytes` larger than the congestion window.

This design has a number of good characteristics. Besides its simplicity, it is also very flexible: since bytes are not assigned to a subflow unless they can be sent immediately, the protocol can react quickly to changing network conditions. In particular, bytes can be promptly assigned to newly-created subflows, and subflows can be deleted quickly without the need to reassign bytes to a different subflow. Furthermore, if a single packet is lost but later packets are successfully ACKed (corresponding to fast retransmit in TCP), the congestion window will not be held up, since the congestion control mechanism is based on a count of bytes in flight, not on a sliding window. This achieves the same effect as TCP congestion window inflation during fast retransmit, but with less bookkeeping and without the vulnerability to bogus duplicate ACK attacks that characterizes TCP's solution (see RFC 5681 [1]).

4.4 Reliable Delivery and Acknowledgements

It is easy to see that the multipath transport protocol I have been sketching must use selective acknowledgements (SACKs). While a normal ACK indicates the end of the first segment of contiguous data received ("I've got everything up to byte 15"), a SACK states precisely which bytes have been received ("I have bytes 0-15, 17-24, and 30") The reason why SACKs are necessary is that in a multipath protocol there may be multiple packet losses or other network events going on at one time, and the single piece of information provided by an ACK may be insufficient for the protocol to recognize and respond to all of them. For example, suppose two subflows each lose a packet at the same time. Normal ACKs would communicate that the packet with the lower sequence number was lost (through a triple-duplicate ACK), but would provide no information about the other loss until the first packet had been retransmitted and ACKed. A SACK would provide more complete information, allowing the protocol to deduce that one packet was lost on each interface and react accordingly.

In addition to generating SACKs, the protocol must also maintain enough state to properly handle retransmissions, timeouts, and congestion control for each subflow. In order to manage congestion control, whenever a packet is lost it must be possible to figure out which subflow it was originally sent out over, so that the appropriate congestion window can be shrunk. This is only possible if the sender keeps track of which bytes were sent over which subflow, and uses this information to figure out where packet losses originated.

4.4.1 The Segment Table

The segment table is the data structure used to perform this bookkeeping. Essentially, it is a linked list of unacked byte ranges (the segments), recording for each segment which subflow it was sent over, and a duplicate ack counter used for fast retransmission. The segments of the segment table do not necessarily correspond to a packet; in fact, for efficiency they should generally be larger than a single packet. It is maintained as follows:

- Whenever data is sent over a subflow, a new segment is created recording that fact.
- Whenever a SACK is received, it is "merged" with the segment table:
  - If a segment table entry's byte range is subsumed by a byte range in the SACK, the entry is removed.
  - If the first part of a segment table entry's byte range is covered by a byte range in the SACK, the segment table entry is truncated so they no longer overlap.
  - If a byte range in the SACK begins within a segment table entry's byte range, and the SACK was received on the subflow the segment was sent out over, the duplicate ack counter is incremented.

This is mostly straightforward: the segment table contains only bytes that have not yet been acknowledged, meaning that it does not grow indefinitely over time. Once a byte is acknowledged, the table promptly forgets about it.

The last item above deserves some additional explanation. Since bytes within a segment are sent sequentially over a single subflow, in the absence of losses and reordering each successive SACK received over a given subflow that mentions a byte range within a certain segment should advance the left edge of that segment. If an ACK is received that fails to do this, it signifies reordering or loss over that subflow, and so the duplicate ACK counter for that particular segment is incremented. When the counter reaches three, fast retransmit is triggered, resending the segment. Resending the segment is simply a form of opportunistic retransmission, ensuring quick recovery in the case of multiple packet losses.

Notice that there is no requirement that retransmission occur on the same subflow: this an area of flexibility afforded to the protocol, allowing for smart optimizations. For example, if multiple losses are experienced in succession, retransmission could be done over another subflow, since the current subflow may be unreliable or dead.

Finally, since each SACK contains information about the entire connection, there is no requirement that acknowledgements be sent over the same subflow the data was received on. However, acknowledgements sent over a different flow must be marked as such (by setting a bit in the header), as they cannot be used in duplicate ACK calculations. It is easy to see why: receiving a SACK on flow 1 that mentions a segment sent out over flow 1 is normally an indication that new data was received on flow 1, but this is not the case if the SACK is actually in response to data received on flow 2.

Timeouts are also handled by the segment table: each segment is timestamped when it is created, and the timestamp

is updated whenever new bytes are acknowledged in a segment. Periodically, the segment table is sweeped, and any segment with a timestamp older than the timeout period is retransmitted.

5 Implementation

I have made strides towards creating a working implementation of the design discussed above. There are several reasons for implementing experimental protocols like this one: to fix ideas, to uncover unforseen gaps in a design, and to gain a better understanding of how the design can be improved going forward.

The Serval project already has a robust implementation, a 30,000 line linux kernel module implementing the whole of the Serval network stack. In Summer 2012 I successfully implemented multipath support in the SAL, and this semester I worked to implement the multipath transport protocol described here on top of this foundation.

I began by creating a prototype of the protocol "in a vacuum": an implementation of congestion control, acknowledgements, and the segment table, essentially performing all duties of the Transport Layer, simply not connected to the actual network stack. In this way I was able to refine and debug many aspects of my segment table implementation. Since then, I have been working on integrating this implementation into the Serval prototype. Unfortunately, I ran into serious integration issues working my code into the kernel, and so while I have made significant progress in my implementation, I have not yet been able to produce a working prototype. This is, of course, an instance of a well known principle of software engineering, Hofstadter's law: "It always takes longer than you expect, even when you take into account Hofstadter's Law" [5].

Though my implementation efforts have not yet been successful, my design has benefited greatly from the insights afforded by these efforts, and I plan to continue to work on this project and deliver a working prototype in due time.

6 Further Work

Besides its simplicity, the main benefit of the design I describe here over MPTCP is the increased flexibility it provides the Transport Layer. Packets are not committed to a flow until the moment they are sent, meaning that the protocol can adapt to current conditions immediately, without worrying about remapping or reassigning packets, and acknowledgements and retransmitted packets can be sent over arbitrary flows. This begs the question: What does this flexibility buy us? What gains in efficiency or loss recovery can be made by novel policies made possible by my design? I hope once a working implementation is complete, experimental work can proceed to answer these and other questions.

7 Conclusion

The Internet is an archetypical example of the "Worse is Better" approach to software design: its versatility and ubiquity are due in large part to the simplicity of its basic design, and the conscious decisions made by its designers to not provide certain guarantees or services to users of the network. However, the concept of resource pooling, as elaborated by Wischik et al. [14], is so important and so central to providing reliable performance that it has been implemented over and over again by various stakeholders in the Internet ecosystem, often in ways that adversely affect the network at large or impose extra costs and redundant design. A multipath transport protocol provides a way to implement resource pooling in a consistent, elegant way that will benefit users, administrators, and designers across the Internet. Although MPTCP is an excellent step towards a multipath transport protocol, its design is constrained by the need to accomodate middleboxes and a network architecture not designed for multipath. A simpler and more flexible protocol can be created by making use of the versatile network abstraction provided by Serval.

My protocol design is based around guiding principle of "Dumb Subflows, Smart Connection": as much state and complexity should be pushed to the connection level, to avoid the redundant computations and lack of flexibility that come from putting too much complexity at the subflow level. Compared to MPTCP, this allows a simpler congestion control mechanism and more flexibility in retransmissions and acknowledgements, as well as the ability to quickly adjust to changing network conditions. Connection state is managed through a segment table which tracks byte ranges sent out over the various subflows, and is continually updated based on the SACKs received. A proof of concept implementation has shown that the segment table data structure and selective acknowledgment system described above works, and progress has been made towards a working implementation within the Linux kernel.

Acknowledgments

Many thanks to Mike Freedman and Erik Nordström. Professor Freedman was patient and helpful through the whole process, and I would be in a sorry state indeed were it not for Erik's aid in understanding the Serval code base.

References

[1] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP congestion control. RFC 5681 (Draft Standard), Sept. 2009.

[2] ARYE, M., NORDSTROM, E., KIEFER, R., REXFORD, J., AND FREEDMAN, M. J. A formally-verified migration protocol for mobile, multi-homed hosts. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on* (2012), IEEE, pp. 1–12.

[3] FORD, B., AND IYENGAR, J. Breaking up the transport logjam. In *ACM HotNets, October* (2008).

[4] GABREL, R. The rise of "Worse is Better". http://www.jwz.org/doc/worse-is-better.html, 1989. Accessed: 2013-05-06.

[5] HOFSTADTER, D. R. *Gödel, Escher, Bach*. Klett-Cotta Stuttgart, 1985.

[6] IETF. Multipath TCP working group charter. charter-ietf-mptcp-03, June 2012.

[7] KELLY, F., AND VOICE, T. Stability of end-to-end algorithms for joint routing and rate control. *ACM SIGCOMM Computer Communication Review 35*, 2 (2005), 5–12.

[8] KHALILI, R., GAST, N., POPOVIC, M., UPADHYAY, U., AND LE BOUDEC, J.-Y. Mptcp is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 1–12.

[9] NORDSTRÖM, E., SHUE, D., GOPALAN, P., KIEFER, R., ARYE, M., KO, S. Y., REXFORD, J., AND FREEDMAN, M. J. Serval: An end-host stack for service-centric networking. *Proc. 9th USENIX NSDI* (2012).

[10] RAICIU, C., HANDLEY, M., AND WISCHIK, D. Coupled congestion control for multipath transport protocols. RFC 6356 (Experimental), Oct. 2011.

[11] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? Designing and implementing a deployable Multipath TCP. In *NSDI* (2012), vol. 12, pp. 29–29.

[12] RAICIU, C., WISCHIK, D., AND HANDLEY, M. Practical congestion control for multipath transport protocols. *University College London, London/United Kingdom, Tech. Rep* (2009).

[13] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS) 2*, 4 (1984), 277–288.

[14] WISCHIK, D., HANDLEY, M., AND BRAUN, M. B. The resource pooling principle. *ACM SIGCOMM Computer Communication Review 38*, 5 (2008), 47–52.

[15] WISCHIK, D., HANDLEY, M., AND RAICIU, C. Control of Multipath TCP and optimization of multipath routing in the Internet. In *Network Control and Optimization*. Springer, 2009, pp. 204–218.

[16] WISCHIK, D., RAICIU, C., GREENHALGH, A., AND HANDLEY, M. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (2011), USENIX Association, pp. 8–8.

This paper represents my own work in accordance with University regulations.