Design Overview of Multipath TCP version 0.3 for FreeBSD-10

Nigel Williams, Lawrence Stewart, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 130424A
Swinburne University of Technology
Melbourne, Australia
njwilliams@swin.edu.au, lastewart@swin.edu.au, garmitage@swin.edu.au

***Abstract*—This report introduces FreeBSD-MPTCP v0.3, a modification to the FreeBSD-10 kernel that enables support for the IETF's emerging Multipath TCP (MPTCP) specification. We outline the motivation for (and potential benefits of) using MPTCP, and discuss key architectural elements of our design.**

***Index Terms*—CAIA, TCP, Multipath, Kernel, FreeBSD**

I. Introduction

Traditional TCP has two significant challenges – it can only utilise a single network path between source and destination per session, and (aside from the gradual deployment of explicit congestion notification) congestion control relies primarily on packet loss as a congestion indicator. Traditional TCP sessions must be broken and reestablished when endpoints shift their network connectivity from one interface to another (such as when a mobile device moves from 3G to 802.11, and thus changes its active IP address). Being bound to a single path also precludes multihomed devices from using any additional capacity that might exist over alternate paths.

TCP Extensions for Multipath Operation with Multiple Addresses (RFC6824) [1] is an experimental RFC that allows a host to spread a single TCP connection across multiple network addresses. Multipath TCP (MPTCP) is implemented within the kernel and is designed to be backwards compatible with existing TCP socket APIs. Thus it operates transparently from the perspective of the application layer and works with unmodified TCP applications.

As part of CAIA's NewTCP project [2] we have developed and released a prototype implementation of the MPTCP extensions for FreeBSD-10 [3]. In this report we describe the architecture and design decisions behind our version 0.3 implementation. At the time of writing, a Linux reference implementation is also available at [4].

The report is organised as follows: we briefly outline the origins and goals of MPTCP in Section II. In Section III we detail each of the main architectural changes required to support MPTCP in the FreeBSD 10 kernel. The report concludes with Section IV.

II. Background to Multipath TCP (MPTCP)

The IETF's Multipath TCP (MPTCP) working group[1] is focused on an idea that has emerged in various forms over recent years – namely, that a single transport session as seen by the application layer might be striped or otherwise multiplexed across multiple IP layer paths between the session's two endpoints. An over-arching expectation is that TCP-based applications see the traditional TCP API, but gain benefits when their session transparently utilises multiple, potentially divergent network layer paths. These benefits include being able to stripe data over parallel paths for additional speed (where multiple similar paths exist concurrently), or seamlessly maintaining TCP sessions when an individual path fails or as a mobile device's multiple underlying network interfaces come and go. The parts of an MPTCP session flowing over different network paths are known as *subflows*.

A. Benefits for multihomed devices

Contemporary computing devices such as smartphones, notebooks or servers are often multihomed (multiple network interfaces, potentially using different link layer technologies). MPTCP allows existing TCP-based applications to utilise whichever underlying interface (network path) is available at any given time, seamlessly maintaining transport sessions when endpoints shift their network connectivity from one interface to another.

When multiple interfaces are concurrently available, MPTCP enables the distribution of an application's

[1]http://datatracker.ietf.org/wg/mptcp/charter/

traffic across all or some of the available paths in a manner transparent to the application. Networks can gain traffic engineering benefits as TCP connections are steered via multiple paths (for instance away from congested links) using coupled congestion control [5]. Mobile devices such as smartphones and tablets can be provided with persistent connectivity to network services as they transition between different locales and network access media.

B. SCTP is not quite the same as MPTCP

It is worth noting that SCTP (stream control transmission protocol) [6] also supports multiple endpoints per session, and recent CMT work [7] enables concurrent use of multiple paths. However, SCTP presents an entirely new API to applications, and has difficulty traversing NATs and any middleboxes that expect to see only TCP, UDP or ICMP packets 'in the wild'. MPTCP aims to be more transparent than SCTP to applications and network devices.

C. Previous MPTCP implementation and development

Most early MPTCP work was supported by the EU's Trilogy Project[2], with key groups at University College London (UK)[3] and Université catholique de Louvain in Louvain-la-Neuve (Belgium)[4] publishing code, working group documents and research papers. These two groups are responsible for public implementations of MPTCP under Linux userland[5], the Linux kernel[6] and a simulation environment (htsim)[7]. Some background on the design, rationale and uses of MPTCP can be found in papers such as [8]–[11].

D. Some challenges posed by MPTCP

MPTCP poses a number of challenges.

1) Classic TCP application interface: The API is expected to present the single-session socket of conventional TCP, while underneath the kernel is expected to support the learning and use of multiple IP-layer identities for session endpoints. This creates a non-trivial implementation challenge to retrofit such functionality into existing, stable TCP stacks.

[2]http://www.trilogy-project.org/
[3]http://nrg.cs.ucl.ac.uk/mptcp/
[4]http://inl.info.ucl.ac.be/mptcp
[5]http://nrg.cs.ucl.ac.uk/mptcp/mptcp_userland_0.1.tar.gz
[6]https://scm.info.ucl.ac.be/trac/mptcp/
[7]http://nrg.cs.ucl.ac.uk/mptcp/htsim_0.1.tar.gz

2) Interoperability and deployment: Any new implementation must interoperate with the reference implementation. The reference implementation has not yet had to address interoperation, and as such holes and assumptions remain in the protocol documents. An interoperable MPTCP implementation, given FreeBSD's slightly different network stack paradigm relative to Linux, should assist in IETF standardisation efforts. Also, the creation of a BSD-licensed MPTCP implementation benefits both the research and vendor community.

3) Congestion control (CC): Congestion control (CC) must be coordinated across the subflows making up the MPTCP session, to both effectively utilise the total capacity of heterogeneous paths and ensure a multipath session does not receive *"...more than its fair share at a bottleneck link traversed by more than one of its subflows"* [12]. The WG's current proposal for MPTCP CC remains fundamentally a loss-based algorithm that *"...only applies to the increase phase of the congestion avoidance state specifying how the window inflates upon receiving an ACK. The slow start, fast retransmit, and fast recovery algorithms, as well as the multiplicative decrease of the congestion avoidance state are the same as in standard TCP*" (Section 3, [12]). There appears to be wide scope for exploring how and when CC for individual subflows ought to be tied together or decoupled.

III. Changes to FreeBSD's TCP Stack

Our MPTCP implementation has been developed as a kernel patch[8] against revision 248226 of FreeBSD-10.

A broad view of the changes and additions between revision 248226 and the MPTCP-enabled kernel:

1) Creation of the Multipath Control Block (MPCB) and the repurposing of the existing TCP Control Block (TCPCB) to act as a MPTCP subflow control block.
2) Changes to user requests (called from the socket layer) that handle the allocation, setup and deallocation of control blocks.
3) New data segment reassembly routines and datastructures.
4) Changes to socket send and socket receive buffers to allow concurrent access from multiple subflows and mapping of data.
5) MPTCP option insertion and parsing code for input and output paths.

[8]Implementing MPTCP as a loadable kernel module was considered, but deemed impractical due to the number of changes required.

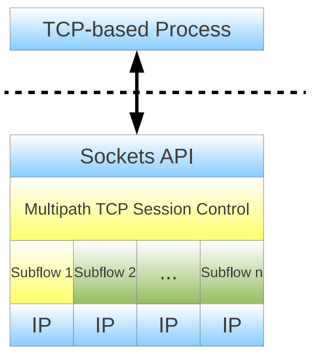


Figure 1. Logical MPTCP stack structure (left) versus traditional TCP (right). User space applications see same socket API.

6) Locking mechanisms to handle additional concurrency introduced by MPTCP.
7) Various MPTCP support functions (authentication, hashing etc).

The changes are covered in more detail in the following subsections.

A. Protocol Control Blocks

The implementation adds a new control block, the MPTCP control block (MPCB), and repurposes the TCP Control Block (RFC 793 [13]) as a subflow control block. The header file *netinet/mptcp_var.h* has been added to the FreeBSD source tree, and the MPCB structure is defined within.

A MPCB is created each time an application creates a TCP socket. The MPCB maintains all information required for multipath operation and manages the subflows in the connection. It sits logically between the subflow TCP control blocks and the socket layer. This arrangement is compared with traditional TCP in Figure 1.

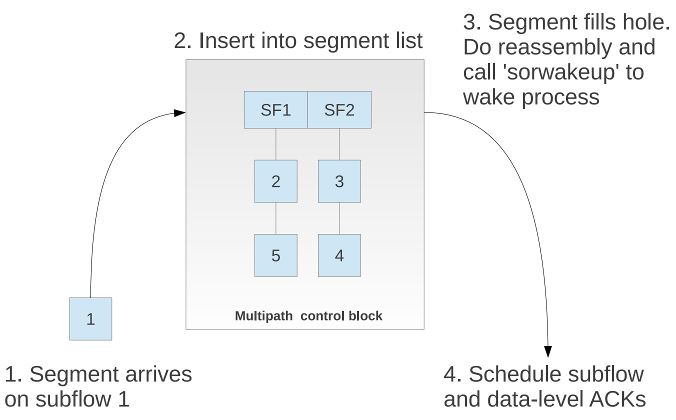
At creation, each MPCB associated with a socket contains at least one subflow (the *master subflow*, or *subflow 1*). The subflow control block is a modified traditional TCP control block found in *netinet/tcp_var.h*.

Protocol control blocks are initialised and attached to sockets via functions in *netinet/tcp_usrreq.c* (user requests). A call to *tcp_connect()* in *netinet/tcp_usrreq.c* results in a call to *mp_newmpcb()*, which allocates and initialises the MPCB.

A series of functions (*tcp_subflow_**) are implemented in *tcp_usrreq.c* and are used to create and attach any additional *slave subflows* to the MPTCP connection.

B. Segment Reassembly

MPTCP adds a data-level sequence space above the sequence space used in standard TCP. This allows segments received on multiple subflows to be



2. Insert into segment list
3. Segment fills hole. Do reassembly and call 'sorwakeup' to wake process
SF1
SF2
2
3
5
4
1
Multipath control block
1. Segment arrives on subflow 1
4. Schedule subflow and data-level ACKs

Figure 2. Each subflow maintains a segment receive list. Segments are placed into the list in subflow-sequence order as they arrive (data-level sequence numbers are shown). When a segment arrives in data-sequence order, the lists are locked and data-level re-ordering occurs. The application is then alerted and can read in the in-order data.

ordered before delivery to the application. Modifications to reassembly are found in *netinet/tcp_reass.c* and in *kern/uipc_socket.c*.

In pre-MPTCP FreeBSD, if a segment arrives that is not the next expected segment (sequence number does not equal RCV.NXT), it is placed into a reassembly queue. Segments are placed into this queue in sequence order until the expected segment arrives. At this point, all in-order segments held in the queue are appended to the socket receive buffer and the process is notified that data can be read in. If a segment arrives that is in-order and the reassembly list is empty, it is appended to the receive buffer immediately.

In our implementation, subflows do not access the socket receive buffer directly, and instead repurpose the traditional reassembly queue for both in-order queuing and out-of-order reassembly. Unknown to subflows, their individual queues form part of a larger multipath-related reassembly data structure, shown in Figure 2.

All incoming segments on a subflow are appended to that subflow's reassembly queue (the *t_segq* member of the TCP control block defined in *netinet/tcp_var.h*) in subflow sequence order. When the head of a subflow's queue is in data sequence order (segment's data level sequence number equals ds_rcv_nxt), then data-level reassembly is triggered (ultimately by a wakeup on the socket which will in turn defer reassembly to the userspace thread context, but due to unresolved bugs we currently trigger from kernel thread context).

Data-level reassembly involves traversing each subflow segment list and appending in-sequence (data-level) segments to the socket receive buffer. This occurs in the *mp_do_reass()* function of *netinet/tcp_reass.c*. During this time a write lock is used to exclude subflows from manipulating their reassembly queues.

Subflow and data-level reassembly have been split this way to reduce lock contention between subflows and the multipath layer. It also allows data-reassembly to be deferred to the application's thread context during a read on the socket, rather than performed by a kernel fast-path thread.

At completion of data-level reassembly, a data-level ACK is scheduled on whichever subflow next sends a regular TCP ACK packet.

C. Send and Receive Socket Buffers

In FreeBSD's implementation of standard TCP, segments are sent and received over a single (address,port) tuple, and socket buffers exist exclusively for each TCP session. MPTCP sessions have *1+n* (where n denotes additional addresses) subflows that must access the same send and receive buffers. The following sections describe the changes to the socket buffers and the addition of the *ds_map*. .

1) The ds_map *struct:* The *ds_map* struct is defined in *netinet/tcp_var.h* and used for both send-related and receive-related functions. Send and receive related maps are stored in the subflow control block lists *t_txmaps* (send buffer maps) and *t_rxmaps* (receive buffer maps) respectively.

On the send side, *ds_maps* track accounting information related to DSN maps advertised to the peer, and are used to mediate access between subflows and the send socket buffer. By mediating socket buffer access in this way, lock contention can be avoided when sending data from a *ds_map*. On the receive side, *ds_maps* track accounting information related to received DSN maps and associated payload data from the peer.

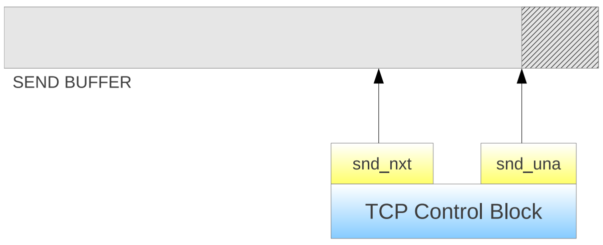


Figure 3. Standard TCP Send Buffer. The lined area represents sent bytes that have been acknowledged by the receiver.

2) Socket Send Buffer: Figure 3 illustrates how in standard TCP, each session has exclusive access to its own send buffer. The variables *snd_nxt* and *snd_una* are used respectively to track which bytes in the send buffer are to be sent next, and which bytes were the last acknowledged by the receiver.

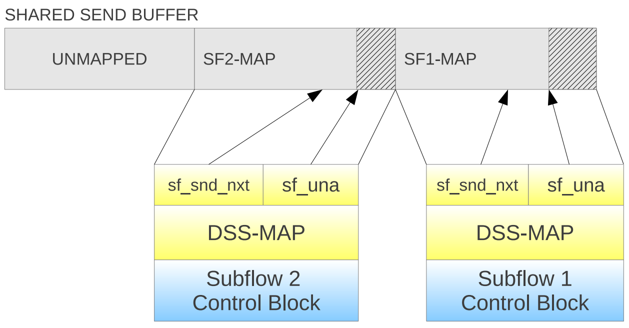


Figure 4. A MPTCP send buffer contains bytes that must be mapped to multiple TCP-subflows. Each subflow is allocated one or more *ds_maps* (DSS-MAP) that define these mappings.

Figure 4 illustrates how in the multipath kernel, data from the sending application is still stored in a single send socket buffer. However access to this buffer is moderated by the packet scheduler in *mp_get_map(),* implemented in *netinet/mptcp_subr.c* (see Section III-D)

The packet scheduler is run when a subflow attempts to send data via *tcp_output()* without owning a *ds_map* that references unsent data.

When invoked, the scheduler must decide whether the subflow should be allocated any data. If granted, allocations are returned as a *ds_map* that contains an offset into the send buffer and the length of data to be sent. Otherwise, a NULL map is returned, and the send buffer appears 'empty' to the subflow. The *ds_map* effectively acts as a unique socket buffer from the perspective of the subflow (i.e. subflows are not aware of what other subflows are sending). The scheduler is not invoked again until the allocated map has been completely sent.

This scheme allows subflows to make forward progress with variable overheads that depend on how frequently the scheduler is invoked i.e. larger maps reduce overheads.

As a result of sharing the underlying send socket buffer via *ds_maps* to avoid data copies, releasing bytes becomes more complex. Firstly, data-level ACKs rather than subflow-level ACKs mark the multipath-level stream bytes which have safely arrived, and therefore control the advancement of ds_snd_una. Secondly, *ds_maps* can potentially overlap any portion of their socket buffer mapping with each other (e.g. data-level retransmit), and therefore the underlying socket buffer bytes (encapsulated in chained mbufs) can only be dropped when both ds_snd_una has acknowledged them and all maps which reference the bytes have been deleted.

To potentially defer the dropping of bytes from the socket buffer without adversely impacting application throughput requires that socket buffer occupancy be accounted for logically rather than actually. To this end, the socket buffer variable *sb_cc* of an MPTCP socket send buffer refers to the logical number of bytes held in the buffer without data-level acknowledgment, and a new variable *sb_actual* has been introduced to track the actual number of bytes in the buffer.

3) Socket Receive Buffer: In pre-MPTCP FreeBSD, in-order segments were copied directly into the receive buffer, at which time the process was alerted that data was available to read. The remaining space in the receive buffer was used to advertise a receive window to the sender.

As described in Section III-B, each subflow now holds all received segments in a segment list, even if they are in subflow sequence order. The segment lists are then linked by their list heads to create a larger data-level reassembly data structure. When a segment arrives that is in data sequence order, data-level reassembly is triggered and segments are copied into the receive buffer. As the size of the reassembly list is effectively unbounded, we currently advertise a maximum receive window (*TCP_MAXWIN * scaling factor*) on all subflows.

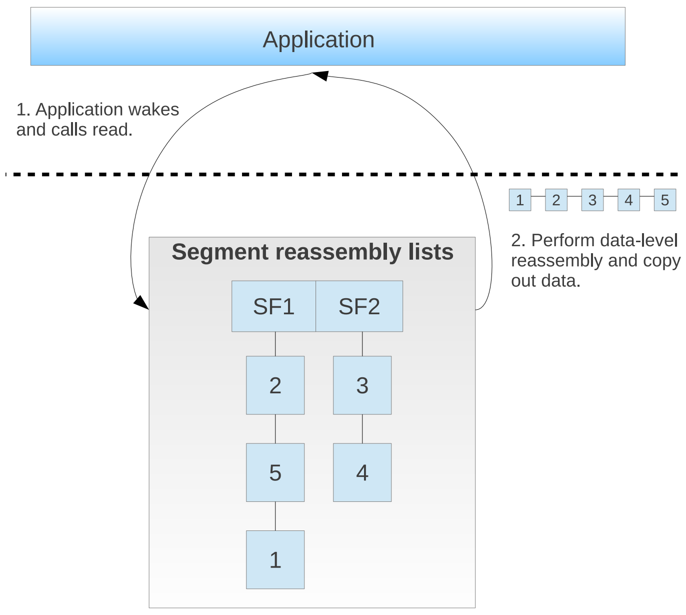

Application
1. Application wakes and calls read.
1
2
3
4
5
Segment reassembly lists
2. Perform data-level reassembly and copy out data.
SF1
SF2
2
3
5
4
1

Figure 5. A future release will integrate the multipath reassembly structure into the socket receive buffer. Segments will be read directly from the multi-subflow aware buffer as data-level reassembly occurs.

We plan to integrate the multipath reassembly structure into the socket receive buffer in a future release. Coupled together with deferred reassembly, an application's thread context would be responsible for performing data-level reassembly on the multi-subflow aware buffer after being woken up by a subflow that received the next expected data-level segment (see Figure 5).

D. Packet Scheduler

The packet scheduler is responsible for determining which subflows are able to send data from the socket send buffer, and how much data they can send. A basic packet scheduler is implemented in the v0.3 patch, and can be found in the *mp_get_map()* function of *netinet/mptcp_subr.c*.

The current algorithm checks the number of unmapped bytes (bytes not yet allocated to an existing *ds_map* struct) in the buffer and the total occupancy of the buffer. If the buffer is not full, the application is free to write more data to the socket so we assign the current contents of the buffer to the requesting subflow on the basis more data will be written soon. If the buffer is full (*sb_cc* is equal to the buffer's maximum capacity *sb_hiwat*), the application is stalled waiting for buffer space to become available and subflows are competing for the unmapped data in the buffer. In this case, we allocate an amount of data equal to the unmapped bytes divided by the number of active subflows, with a floor value set to the MSS of the requesting subflow. This provides some protection against a subflow being starved of data to transmit.

The scheduler returns an appropriate *ds_map* struct, and since the length allocated can exceed oneMSS, this mapping forms the basis of a multi-packet MPTCP DSS-map.

The packet scheduler will be modularised and extended with congestion control hooks in future updates, providing scope for more complex scheduling of maps.

IV. Conclusions and Future Work

This report introduced FreeBSD-MPTCP v0.3, a modification of the FreeBSD kernel enabling Multipath TCP [1] support. We outlined the motivation behind and potential benefits of using multipath TCP, and discussed key architectural elements of our design.

We expect to update and improve our MPTCP implementation in the future, and documentation will be updated as this occurs. We also plan on releasing a detailed design document that will provide more in-depth detail about the implementation. Code profiling and analysis of on-wire performance are also planned.

Our aim is to use this implementation as a basis for further research into MPTCP congestion control, as noted in Section II-D3.

Acknowledgements

This project has been made possible in part by a gift from the Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation.

References

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Internet Engineering Task Force, 12 January 2013. [Online]. Available: http://tools.ietf.org/html/rfc6824

[2] G. Armitage and L. Stewart. (2013) Newtcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/

[3] G. Armitage and N. Williams. (2013) Multipath tcp project website. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/mptcp/

[4] O. Bonaventure. (2013) Multipath tcp linux kernel implementation. [Online]. Available: http://multipath-tcp.org/pmwiki.php

[5] D. Wischik, C. Raiciu, A. Greenhalgh and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, 2012.

[6] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, "Stream Control Transmission Protocol," RFC 2960, Internet Engineering Task Force, October 2000. [Online]. Available: http://tools.ietf.org/html/rfc2960

[7] P. Amer, M. Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. Stewart, M. Tuexen, "Load sharing for the stream control transmission protocol (SCTP)," Internet Draft, Internet Engineering Task Force, September 2012. [Online]. Available: http://tools.ietf.org/html/html/draft-tuexen-tsvwg-sctp-multipath-05

[8] A. Ford, C. Raiciu, M. Handley, S. Barré, and J.Iyengar, "Architectural Guidelines for Multipath TCP Development," RFC 6182, Internet Engineering Task Force, March 2011. [Online]. Available: http://tools.ietf.org/html/rfc6182

[9] C. Raiciu, C. Paasch, S. Barré, A. Ford, M. Honda, F. Duchène, O. Bonaventure and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," in *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12)*, San Jose, California, 2012.

[10] S. Barré, C. Paasch, and O. Bonaventure, "Multipath tcp: From theory to practice," in *IFIP Networking, Valencia*, May 2011.

[11] C. Raiciu, S. Barré, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM 2011, Toronto, Canada*, August 2011.

[12] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," RFC 6356, Internet Engineering Task Force, October 2011. [Online]. Available: http://tools.ietf.org/html/rfc6356

[13] J. Postel, "Transmission Control Protocol," RFC 793, Internet Engineering Task Force, September 1981. [Online]. Available: http://tools.ietf.org/html/rfc793