Design Overview of Multipath TCP version 0.4 for FreeBSD-11

Nigel Williams, Lawrence Stewart, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 140822A
Swinburne University of Technology
Melbourne, Australia
njwilliams@swin.edu.au, lastewart@swin.edu.au, garmitage@swin.edu.au

***Abstract*—This report introduces FreeBSD-MPTCP v0.4, a modification to the FreeBSD-11 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-11 [3]. In this report we describe the architecture and design decisions behind our version 0.4 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 11 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) 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 265307 of FreeBSD-11. Table I provides a summary of files modified or added to the FreeBSD-11 kernel.

A broad view of the changes and additions between revision 265307 and the MPTCP-enabled kernel:

1) Creation of the Multipath Control Block (MPCB) and the re-purposing 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 data-structures.
4) Changes to socket send and socket receive buffers to allow concurrent access from multiple subflows and mapping of data.

[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

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

File	Status
sys/netinet/tcp_var.h	Modified
sys/netinet/tcp_subr.c	Modified
sys/netinet/tcp_input.c	Modified
sys/netinet/tcp_output.c	Modified
sys/netinet/tcp_timer.c	Modified
sys/netinet/tcp_reass.c	Modified
sys/netinet/tcp_syncache.c	Modified
sys/netinet/tcp_usrreq.c	Modified
sys/netinet/mptcp_var.h	Added
sys/netinet/mptcp_subr.c	Added
sys/kern/uipc_sockbuf.c	Modified
sys/sys/sockbuf.h	Modified
sys/sys/socket.h	Modified
sys/sys/socketvar.h	Modified

Table I
KERNEL FILES MODIFIED OR ADDED AS PART OF MPTCP IMPLEMENTATION

5) MPTCP option insertion and parsing code for input and output paths.
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. For detail on the overall structure and operation of the FreeBSD TCP/IP stack, see [13].

A. Protocol Control Blocks

The implementation adds a new control block, the MPTCP control block (MPCB), and re-purposes the TCP Control Block (RFC 793 [14]) 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. This also includes variables for data-level accounting and session tokens. 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,* or *default subflow*). The subflow control block is a modified traditional TCP control block found in `netinet/tcp_var.h`. Modifications to the control block include the addition of subflow flags, which are used to propagate subflow state to the MPCB (E.g. during packet scheduling).

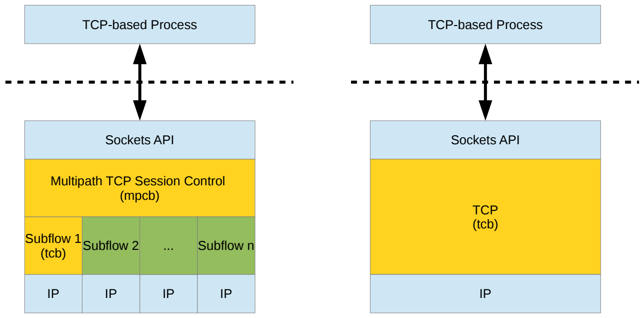

Figure 1. Logical MPTCP stack structure (left) versus traditional TCP (right). User space applications see same socket API.

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 subflows to the MPTCP connection.

B. Asynchronous Task Handlers

Listing 1 Asynchronous tasks: Provide deferred execution for several MPTCP session-related tasks.

```
struct task join_task; /* For enqueuing
    aysnc joins in swi */
struct task data_task; /* For enqueuing
    aysnc subflow sched wakeup */
struct task pcb_create_task; /* For
    enqueueing async sf inp creation */
struct task rexmit_task; /* For enqueuing
    data-level rexmits */
```

When processing a segment, traditional TCP typically follows one of only a few paths through the TCP stack. For example, an incoming packet triggers a hardware interrupt, which causes an interrupt thread to be scheduled that, when executed, handles processing of the packet (including transport-layer processing, generating a response to the incoming packet).

Code executed in this path should be directly relevant to processing the current packet (parsing options, updating sequence numbers, etc). Operations such as copying out data to a process are deferred to other threads.

Maintaining a multipath session requires performing several new operations that may be triggered by incom-

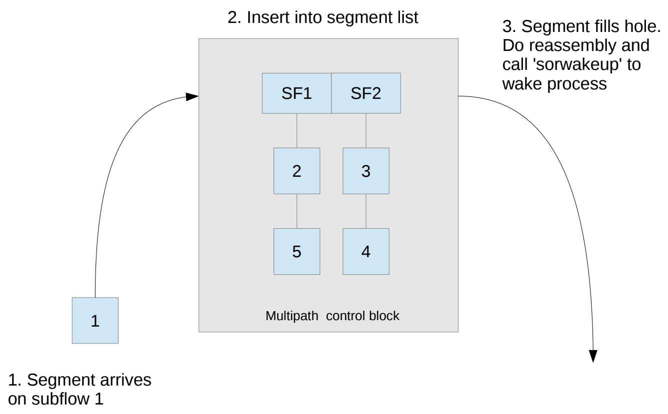

Figure 2. Each subflow maintains a segment receive list. Segments are placed into the list in subflow-sequence order as they arrive (data-level sequence numbers are shown). When a segment arrives in data-sequence order, the lists are locked and data-level re-ordering occurs. The application is then alerted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

ing or outgoing packets. Some of these operations are not immediately related to the current packet therefore can be executed asynchronously. We have thus defined several new handlers for these tasks (Listing 1) which are attached to a software interrupt thread using *taskqueue*[9].

Each of the task variables has an associated handler in `netinet/mptcp_subr.c`, and provide the following functionality:

Join task (`mp_join_task_handler`): Attempt to join addresses the remote host has advertised.

Data task (`mp_datascheduler_task_handler`): Part of packet scheduling. Call output on subflows that are waiting to transmit.

PCB Create Task (`mp_sf_alloc_task_handler`): Allocate PCBs for subflows on an address we are about to advertise.

Retransmit Task (`mp_rexmit_task_handler`): Initiate data-level re-injection of segments after a subflow has failed to deliver data.

The *data task* and *retransmit task* are discussed further in Section III-F and Section III-G respectively.

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

[9]http://www.freebsd.org/cgi/man.cgi?query=taskqueue

In pre-MPTCP FreeBSD, if a segment arrives that is not the next expected segment (sequence number does not equal receive next, `tcp_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 re-purpose 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 is the data-level recieve next, `ds_rcv_nxt`), then data-level reassembly is triggered. In the current implementation, data-level reassembly is triggered from a kernel thread context. A future optimisation will see reassembly deferred to a userspace thread context (specifically that of the reading process).

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.

D. 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

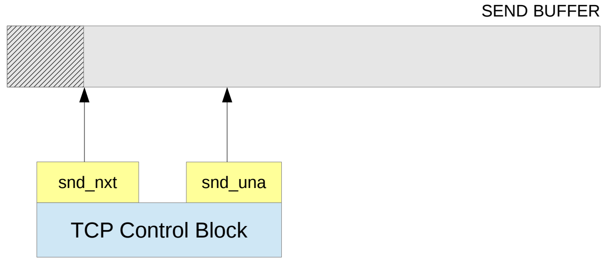

Figure 3. Standard TCP Send Buffer. The lined area represents sent bytes that have been acknowledged by the receiver.

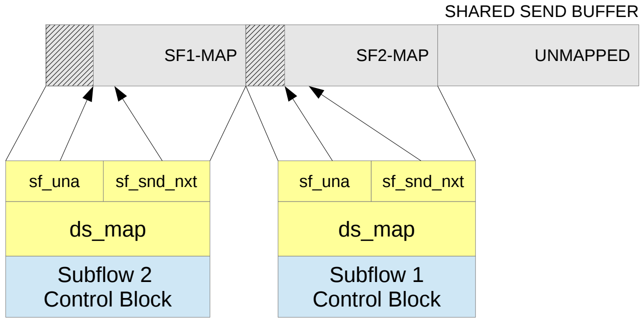

Figure 4. A MPTCP send buffer contains bytes that must be mapped to multiple TCP-subflows. Each subflow is allocated one or more *ds_maps* (DSS-MAP) that define these mappings.

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 (shown in Listing 2), is defined in `netinet/tcp_var.h`, and is used for both send-related and receive-related functions. Maps are stored in the subflow control block lists `t_txmaps` (send buffer maps) and `t_rxmaps` (received maps) respectively. A data-level list, `mp_rxtmitmaps`, is used to queue ds_maps that require retransmission after a data-level timeout. The struct itself contains variables for tracking sequence numbers, memory locations and status. It also includes several list entries (e.g `mp_ds_map_next`) as an instantiated map may belong to different (potentially multiple) lists, depending on the purpose.

On the send side, `ds_maps` track accounting information (bytes sent, acked) related to DSN maps advertised to the peer, and are used to access data in the socket send buffer (via for example `ds_map_offset`, `mbuf_offset`). By mediating socket buffer access through ds_maps in this way, rather than accessing the send buffer directly, lock contention can be reduced when sending data using multiple subflows. On the receive side, *`ds_maps`* are created via incoming DSS options and maintain mappings between subflow and sequence spaces. .

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 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-F)

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 acknowledged 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 acknowledged at the data level 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

Listing 2 ds_map struct

```
struct ds_map {
  TAILQ_ENTRY(ds_map) sf_ds_map_next;
  TAILQ_ENTRY(ds_map) mp_ds_map_next;
  TAILQ_ENTRY(ds_map) mp_dup_map_next;
  TAILQ_ENTRY(ds_map) rxmit_map_next;
  uint64_t ds_map_start; /* starting DSN of mapping */
  uint32_t ds_map_len; /* length of data sequence mapping */
  uint32_t ds_map_offset; /* bytes sent from mapping */
  tcp_seq sf_seq_start; /* starting tcp seq num of mapping */
  uint64_t map_una;  /* bytes sent but unacknowledged in map */
  uint16_t ds_map_csum; /* csum of dss psuedo-header & mapping data */
  struct mbuf* mbuf_start; /* mbuf in which this mappings starts */
  u_int mbuf_offset; /* offset into mbuf where data starts */
  uint16_t flags;    /* status flags */
};
...
/* Status flags for ds_maps */
#define MAPF_IS_SENT 0x0001 /* Sent all data from map */
#define MAPF_IS_ACKED 0x0002 /* All data in map is acknowledged */
#define MAPF_IS_DUP 0x0004 /* Duplicate, already acked at ds-level */
#define MAPF_IS_REXMIT 0x0008 /* Is a rexmit of a previously sent map */
```

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-C, 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.

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).

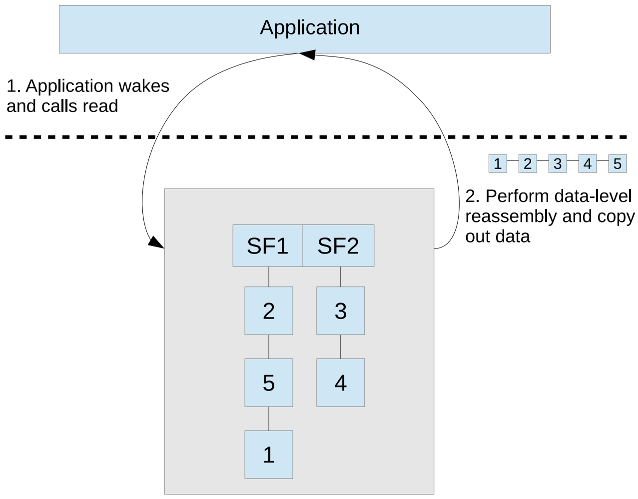

Figure 5. A future release will integrate the multipath reassembly structure into the socket receive buffer. Segments will be read directly from the multi-subflow aware buffer as data-level reassembly occurs.

E. Receiving DSS Maps and Acknowledgments

As mentioned in Section III-D1, the `ds_map` struct is used within the send and receive paths as well as packet scheduling. The struct allows the receiver to track incoming data-maps, and the sender to track acknowledgement of data at subflow- and data- levels. The following subsections detail the primary uses of `ds_maps` in the send and receive paths.

1) Receiving data mappings: New *ds_maps* are created when a packet that contains a MPTCP DSS (Data-Sequence Signal) option that specifies a DSN-map (Data-Sequence Number) is received. Maps are stored within the subflow-level list `t_rxdsmaps` and are used to derive the DSN of an incoming TCP segment (in cases where a mapping spans multiple segments, the DSN will not be included with the transmitted packet). The processing of the DSS option (Figure 6), is summarised as follows:

1) If an incoming DSN-map is found during option parsing, it is compared to an existing list of mappings in `t_rxdmaps`. While looking for a matching map, any fully-acknowledged maps are discarded.
2) If the incoming data is found to be covered by an existing ds_map entry, the incoming DSN-map is disregarded and the existing map is selected. If the mapping represents new data, a new `ds_map` struct is allocated and inserted into the received map list.
3) The returned map - either newly allocated or existing - is used to calculate the appropriate DSN for the segment. The DSN is then "tagged" (see below) onto the mbuf header of the incoming segment.

The *mbuf_tags*[10] framework is used to attach DSN metadata to the incoming segment. Tags are attached to the mbuf header of the incoming packet, and can hold additional metadata (e.g. VLAN tags, firewall filter tags). A structure, `dsn_tag` (Listing 3) is defined in `netinet/mptcp_var.h` to hold the mbuf tag and the 64-bit DSN.

A dsn_tag is created for each packet, regardless of whether a MPTCP connection is active. For standard TCP connections this means the TCP sequence number of the packet is placed into the dsn_tag. Listing 4 shows use of the tags for active MPTCP connections.

Once a DSN has been associated with a segment, standard input processing continues. The DSN is eventually read during segment reassembly.

2) ACK processing and DS_Maps: The MPTCP layer separates subflow-level sequence space and the socket send buffers. As the same data may be mapped to multiple subflows, data cannot be freed from the send buffer until all references to it have been removed. A single ds_map is stored in both subflow-level and data-level transmit lists, and must be acknowledged at both levels before the data can be cleared from the send buffer.

Although subflow-level acknowledgment does not immediately result in the freeing of send buffer data, the data is considered 'delivered' from the perspective of the subflow. Subflow-level processing of ACKs is shown in Figure 7.

On receiving an ACK, the amount of data acknowledged is calculated and the list of transmitted maps, `t_txdmaps`, is traversed. Maps covered by the acknowledgement are marked as being 'acked' and are dropped from the transmitted maps list. At this point

Listing 3 dsn_tag struct: This structure is used to attach a calculated DSN to an incoming packet.

```
/* mbuf tag defines */
#define PACKET_TAG_DSN 10
#define PACKET_COOKIE_MPTCP 34216894
#define DSN_LEN 8

struct dsn_tag {
   struct m_tag tag;
   uint64_t  dsn;
};
```

Listing 4 Prepending a dsn_tag to a received TCP packet. The tag is used later during reassembly to order packets from multiple subflows. Unrelated code ommitted for brevity.

```
/* Initialise the mtag and containing
    dsn_tag struct */
struct dsn_tag *dtag = (struct dsn_tag *)
   m_tag_alloc(PACKET_COOKIE_MPTCP,
       PACKET_TAG_DSN, DSN_LEN, M_NOWAIT);
struct m_tag *mtag = &dtag->tag;
...
/* update mbuf tag with current data seq
    num */
dtag->dsn = map->ds_map_start +
    (th->th_seq - map->sf_seq_start);
...
/* And prepend the mtag to mbuf, to be
    checked in reass */
m_tag_prepend(m, mtag);
```

[10] http://www.freebsd.org/cgi/man.cgi?query=mbuf_tags

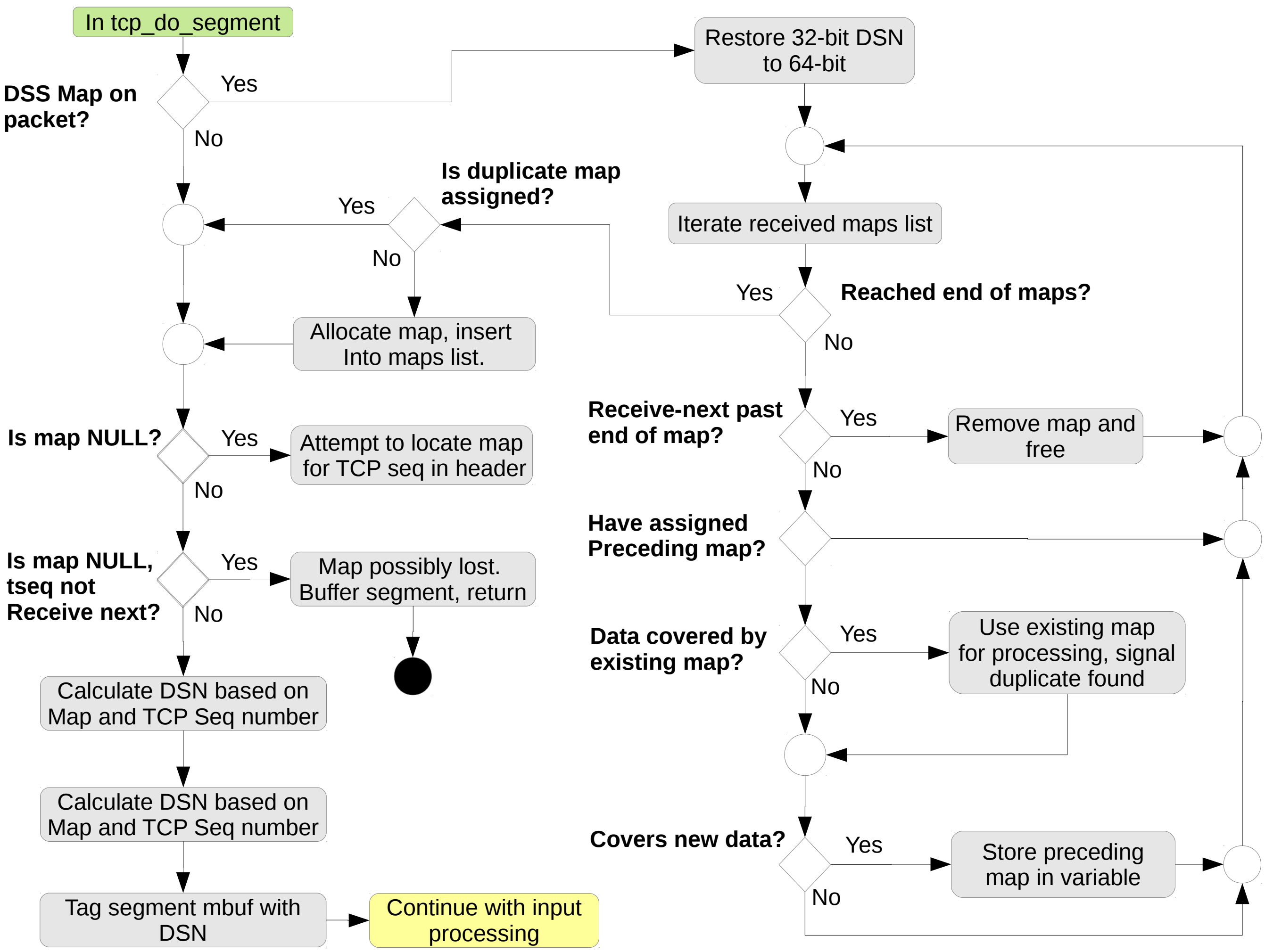

Figure 6. Receiver processing of DSN Maps. A list of ds_maps is used to track incoming packets and tag the mbuf with an appropriate DSN (mapping subflow-level to data-level).

a reference to dropped maps still exists within the data-level transmit list.

If any maps were completed, the `mp_deferred_drop()` function is called (detailed in Section III-E3 below). At this point the data has been successfully delivered, from the perspective of the subflow. It is the MPTCP layers responsibility to facilitate retransmission of data if it is not ultimately acknowledged at the data-level. Data-level acknowledgements (DACKs) are also processed at this time, if present.

3) Deferred drop from send buffer: The function `mp_deferred_drop()` in `netinet/mptcp_subr.c` handles the final accounting of sent data and allows acknowledged data to be dropped from the send buffer. The 'deferred' aspect refers to the fact that the time at which segments are acknowledged is no longer (necessarily) the time at which that data is freed from the send buffer. The process is shown in Figure 8, and broadly described below:

1) Iterate through transmitted maps and store a reference to maps that have been fully acknowledged. The loop is terminated at the end of the list, or if a map is encountered that overlaps the acknowledged region or shares an mbuf with another map that has not yet been acknowledged.
2) If there are bytes to be dropped, the corresponding maps are freed and the bytes are dropped from the socket send buffer. The process is woken up at this time to write new data. If there are no bytes to drop, all outstanding data has been acknowledged

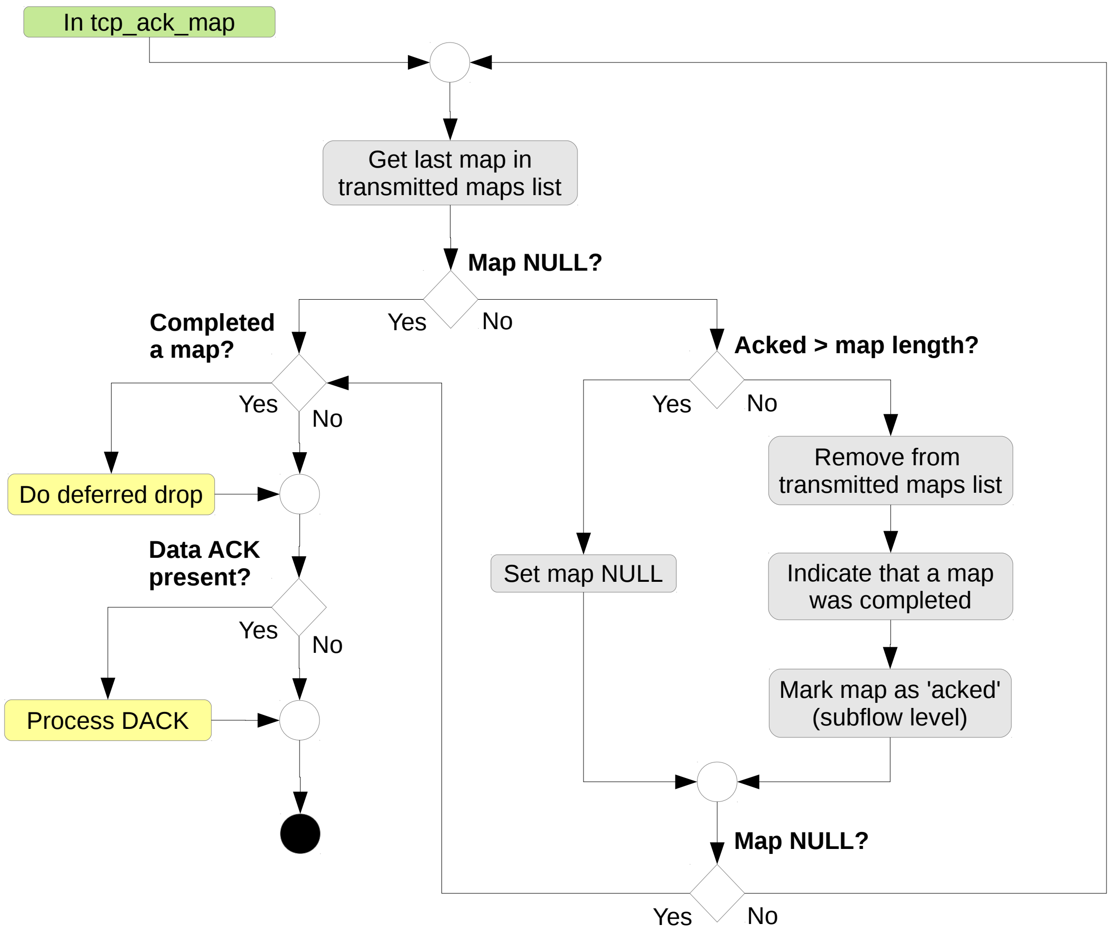

Figure 7. Transmitted maps must be acknowledged at the subflow- and data-levels. However, once acknowledged at the subflow level, the subflow considers the data as being 'delivered'.

and the send buffer is empty, the process is woken so that it may write new data.

F. Packet Scheduling

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 v0.4 patch, and can be found within the `mp_find_dsmap()` function of `netinet/mptcp_subr.c` and `tcp_usr_send()` in `netinet/tcp_usrreq.c`. The current scheduler implementation controls two common pathways through which data segments can be requested for output - calls to `tcp_usr_send()` from the socket, and direct calls to `tcp_output()` from within the TCP stack (for example from `tcp_input()` on receipt of an ACK). The packet scheduler will be modularised in future updates, providing scope for more complex scheduling schemes.

Figure 9 shows these data transmission pathways, and the points at which scheduling decisions are made. To control which subflows are able to send data at a particular time the scheduler uses two subflow flags: `SFF_DATA_WAIT` and `SFF_SEND_WAIT`.

1) SFF_SEND_WAIT: On calls to `tcp_usr_send()`, the list of active subflows is traversed. The first subflow with `SFF_SEND_WAIT` set is selected as the subflow to send data on. The flag is cleared before calling `tcp_output()`.
2) SFF_DATA_WAIT: If a subflow is not allocated a map during a call to `tcp_output()`, the `SFF_DATA_WAIT` flag is set. An asynchronous task, `mp_datascheduler_task_handler` is enqueued when the number of subflows with this flag set is greater than zero. When run, the task will call `tcp_output()` with the waiting subflow.

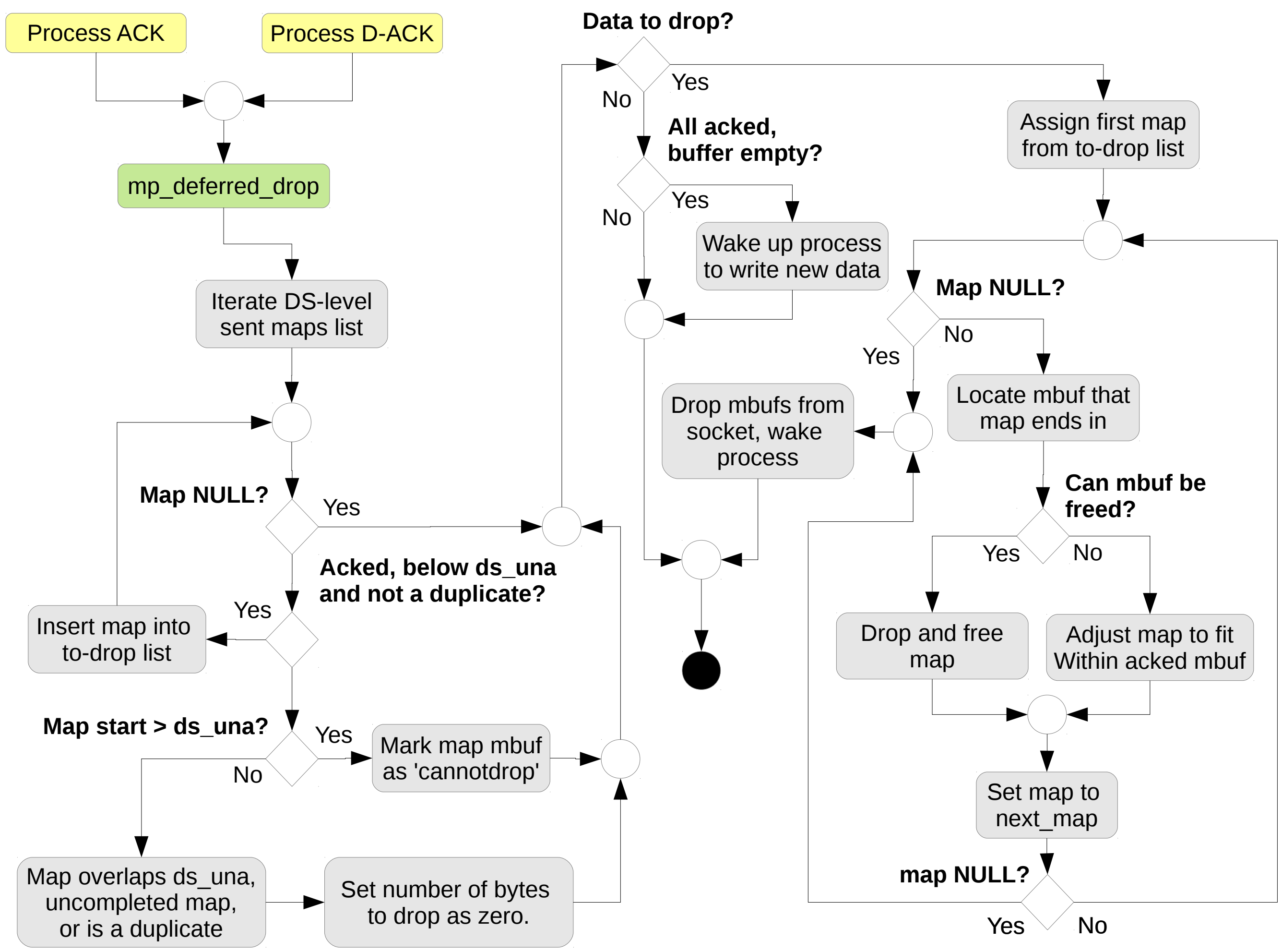

Figure 8. Deferred removal of data from the send buffer. Data bytes are dropped from the send buffer only when acknowledged at the data-level. It is considered deferred as the bytes are not necessarily dropped when acknowledged at the subflow level.

Subflow selection via SEND_WAIT: Figure 10 illustrates the use of the `SFF_SEND_WAIT` flag. When a process wants to send new data, it may use the `sosend()` Socket I/O function, which results in a call to the `tcp_usr_send()` function in `netinet/tcp_usrreq.c`. On entering `tcp_usr_send()` the default subflow protocol block ('master subflow') is assigned.

At this point the list of subflows (if greater than one) is traversed, checking subflow flags for `SFF_SEND_WAIT`. If not set, the flag is set before iterating to the next subflow. If set, the assigned subflow is switched, the loop terminated, and the flag is cleared before calling `tcp_output()`. If no subflows are found to be waiting for data, the 'master subflow' is used for transmission.

Subflow selection via DATA_WAIT: The `SFF_DATA_WAIT` flag is used in conjunction with an asynchronous task to divide ds_map allocation between the active subflows (Figure 11). When in `tcp_output()`, a subflow will call `find_dsmap()` to obtain a mapping of the send buffer. The process of allocating a map is shown in Figure 12. The current implementation restricts map sizes to 1420 bytes (limiting each map to cover one packet). In cases where no map was returned, the subflow flag is marked `SFF_DATA_WAIT`, and the count of 'waiting' subflows is increased. If a map was returned, then the `SFF_DATA_WAIT` flag is cleared (if set) and the 'waiting' count is decremented.

As map sizes are currently limited to a single-packet size, it is likely that on return from `mp_get_map()`

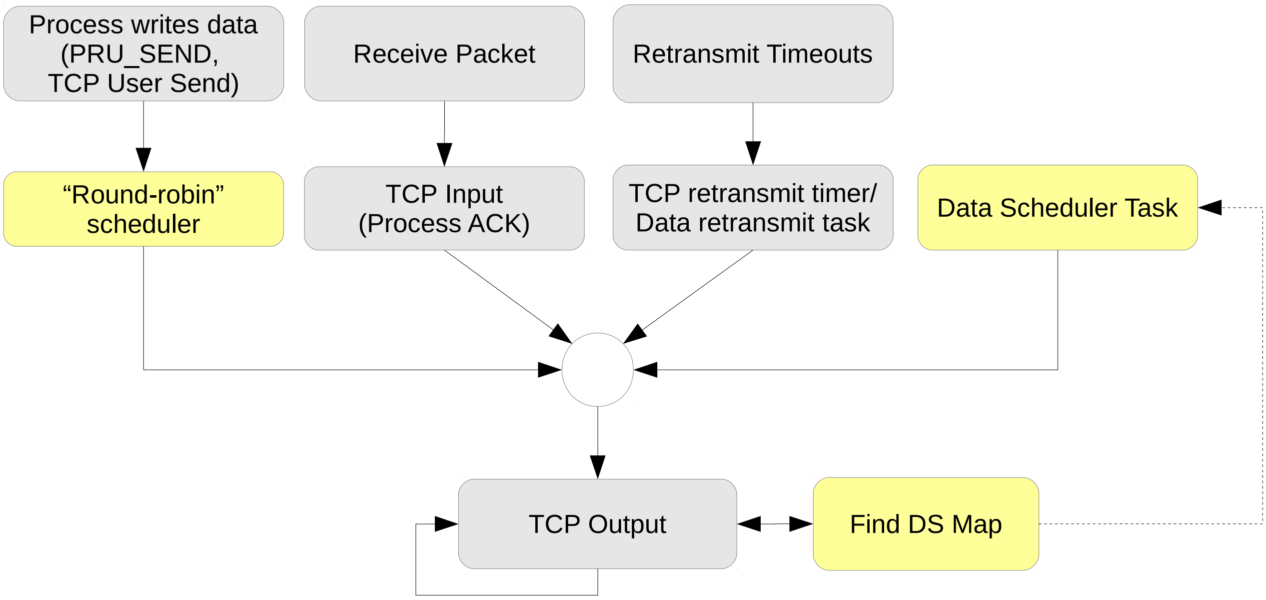

Figure 9. Common pathways to *tcp_output()*, the function through which data segments are sent. Packet scheduling components are shown in orange. Possible entry paths are via the socket (PRU_SEND), on receipt of an ACK or through a retransmit timer. The data scheduler task asynchronously calls into *tcp_output()* when there are subflows waiting to send data. Find DS Map is allocates *ds_maps* to a subflow, and can enqueue the data scheduler task.

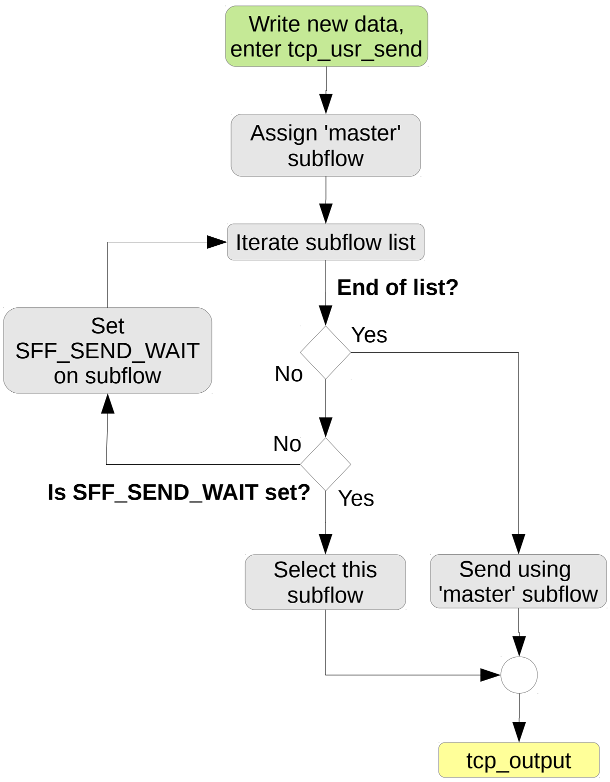

Figure 10. Round-robin scheduling. When a process writes new data to be sent, the scheduler selects a subflow on which to send data.

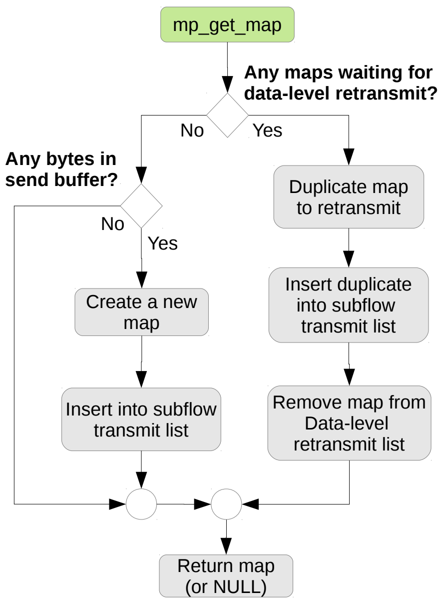

Figure 12. Allocating a ds_map in *mp_get_map()*. First check for maps that require retransmission. Otherwise, if unsent bytes are in the send buffer, a new map is allocated, inserted into the transmission list and returned.

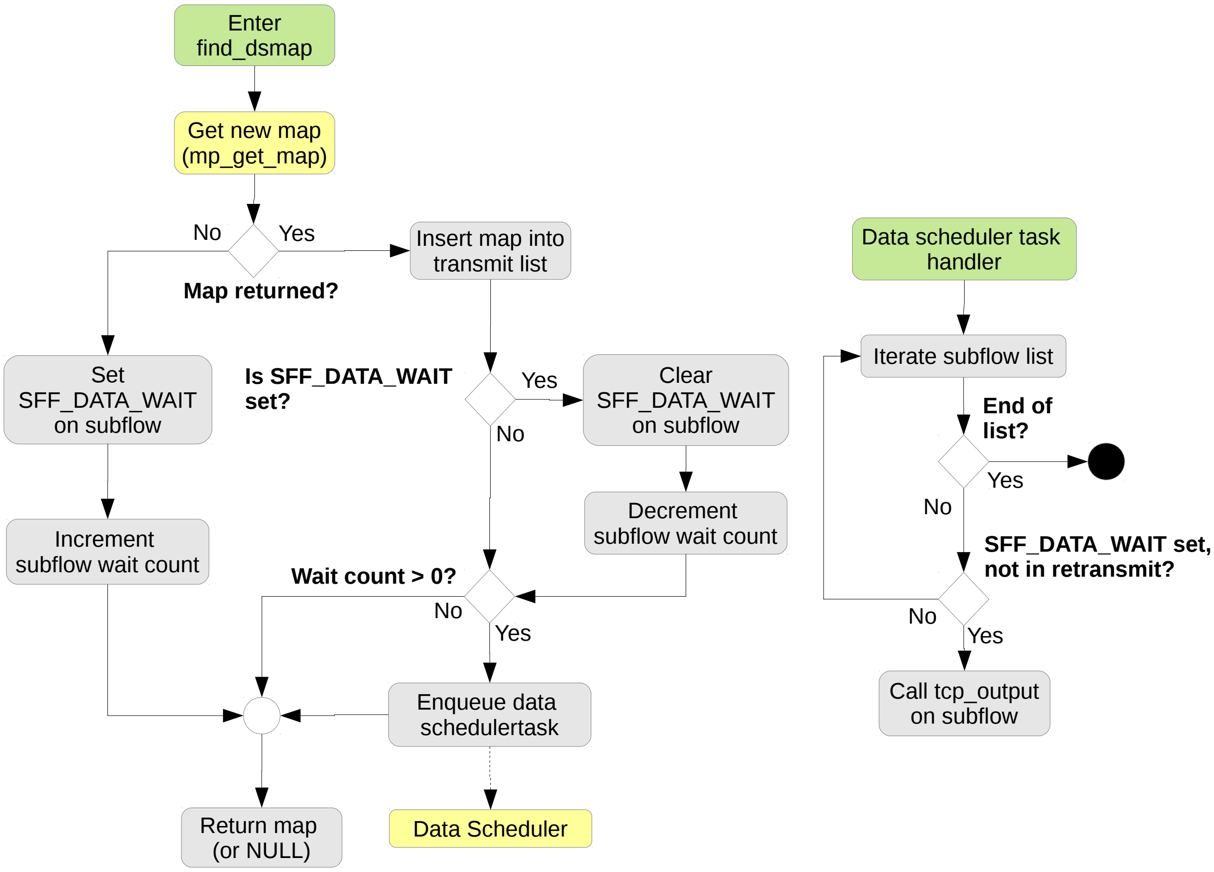

Figure 11. DATA_WAIT subflow selection. Enqueing the data scheduler (left) and the data scheduler (right). Rather than send data segments back-to-back on the same subflow, the scheduler spreads data across the available subflows.

unmapped data remains in the send buffer. Therefore a check is made for any 'waiting' subflows that might be used to send data, in which case a data scheduler asynchronous task is enqueued. When executed, the data scheduler task will call `tcp_output()` on the first subflow with `SFF_DATA_WAIT` set.

G. Data-level retransmission

Data-level retransmission of segments has been included in the v0.4 patch (Figure 13). The current implementation triggers data-level retransmissions based on a count of subflow-level retransmits. In future updates the retransmission strategy will be modularised.

The chart on the left of Figure 13 shows the steps leading to data-level retransmit. Each subflow maintains a retransmission timer that is started on packet transmission and stopped on acknowledgement. If left to expire (called a retransmit timeout, or RTO), the function `tcp_timer_rexmt()` in `netinet/tcp_timer.c` is called, and the subflow will attempt to retransmit from the last bytes acknowledged. The length of the timeout is based in part on the RTT of the path. A count is kept each time an RTO occurs, up to `TCP_MAXRXTSHIFT` (defined as 12 in FreeBSD), at which point the connection is dropped with a RST.

We define a threshold of: `TCP_MAXRXTSHIFT / 4` (or 12/4, giving 3 timeouts) as the point at which data-level retransmission will occur. A check has been placed into `tcp_timer_rexmt()` that tests whether the count of RTOs has met this threshold. If met, a reference to each ds_map that has not been acknowledged at the data-level is placed into `mp_rxtmitmaps` (a list of maps that require data-level re-injection). Finally, an asynchronous task is enqueued (Figure 13, right) that, when executed, locates the first subflow that is not in subflow-level retransmit and calls `tcp_output()`. The packet scheduler will ensure that ds_maps in `mp_rxtmitmaps` are sent before any existing `ds_maps` in the subflow transmit list (*`t_txdsmaps`*).

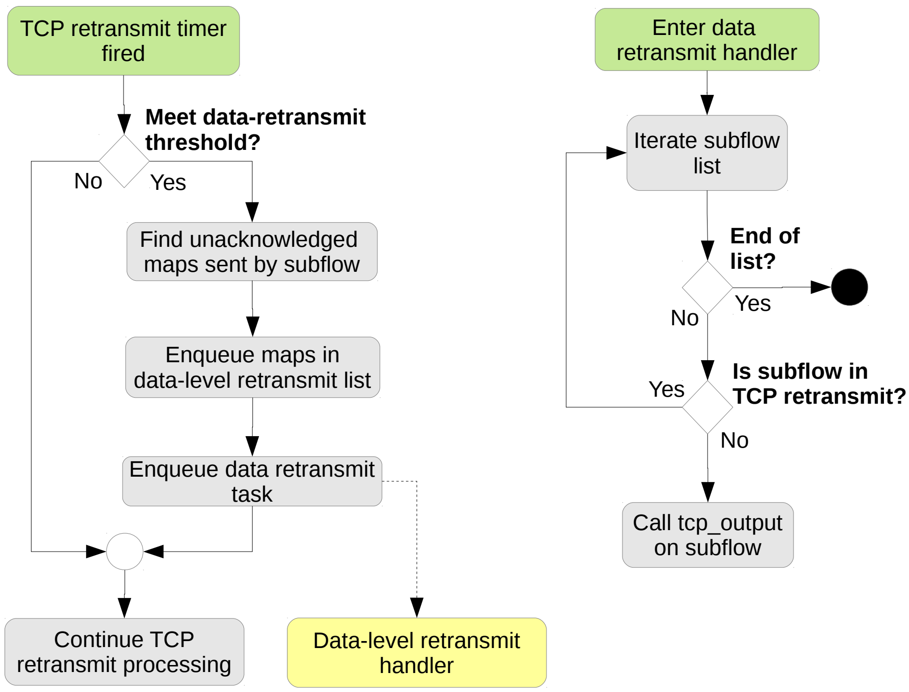

Figure 13. Data-level retransmit logic (left) and task handler (right). Retransmission is keyed off the count of TCP (subflow-level) retransmit timeouts.

It should be noted that subflows retain standard TCP retransmit behaviour independent of data-level retransmits. Subflows will therefore continue to attempt retransmission until the maximum retransmit count is met. On occasions where a subflow recovers from retransmit timeout after data-level retransmission, the receiver will acknowledge the data at the subflow level and discard the duplicate segments.

H. Multipath Session Management

The current implementation contains basic mechanisms for joining subflows and subflow/connection termination, detailed below. Path management will be expanded and modulularised in future updates.

1) Adding subflows: An address can be manually specified for use in a connection between a multi-homed host and single-homed host. This is done using the sysctl[11] utility. Added addresses are available to all MPTCP connections on the host, and will be advertised by all MPTCP connections that reach the established stage.

Subflow joining behaviour is static, and a host will attempt to send an MP_JOIN to any addresses that are received via the ADD_ADDR option[12]. The asynchronous tasks `mp_join_task_handler` and `mp_sf_alloc_task_handler` currently provide this functionality. Both will be integrated with a Path Manager module in a future release.

2) Removing Subflows/Connection Close: The implementation supports removal of subflows from an active MPTCP connection only via TCP reset (RST) due to excessive retransmit timeouts. In these cases, a subflow that has failed will proceed through the standard TCP timeout procedure (as implemented in FreeBSD-11) before closing. Any remaining active subflows will continue to send and receive data. There is currently no other means by which to actively terminate a single subflow on a connection.

On application close of the socket all subflows are shut down simultaneously. The last subflow to be closed will cause the MPCB to be discarded. Subflows on the same host are able to take separate paths (active close, passive close) through the TCP shutdown states.

3) Session Termination: Not documented in this report are modifications to the TCP shutdown code paths.

[11] sysctl net.inet.tcp.mptcp.mp_addresses

[12] One caveat exists: If a host is the active opener (client) in the connection and has already advertised an address, it will not attempt to join any addresses that it receives via advertisement.

Currently the code has been extended in-place with additional checks to ensure that socket is not marked as closed while at least one subflow is still active. These modifications should however be considered temporary and will be replaced with a cleaner solution in a future update.

IV. Conclusions and Future Work

This report describes FreeBSD-MPTCP v0.4, 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] G. Wright, W. Stevens, *TCP/IP Illustrated, Volume 2, The Implementation*. Addison Wesley, 2004.

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