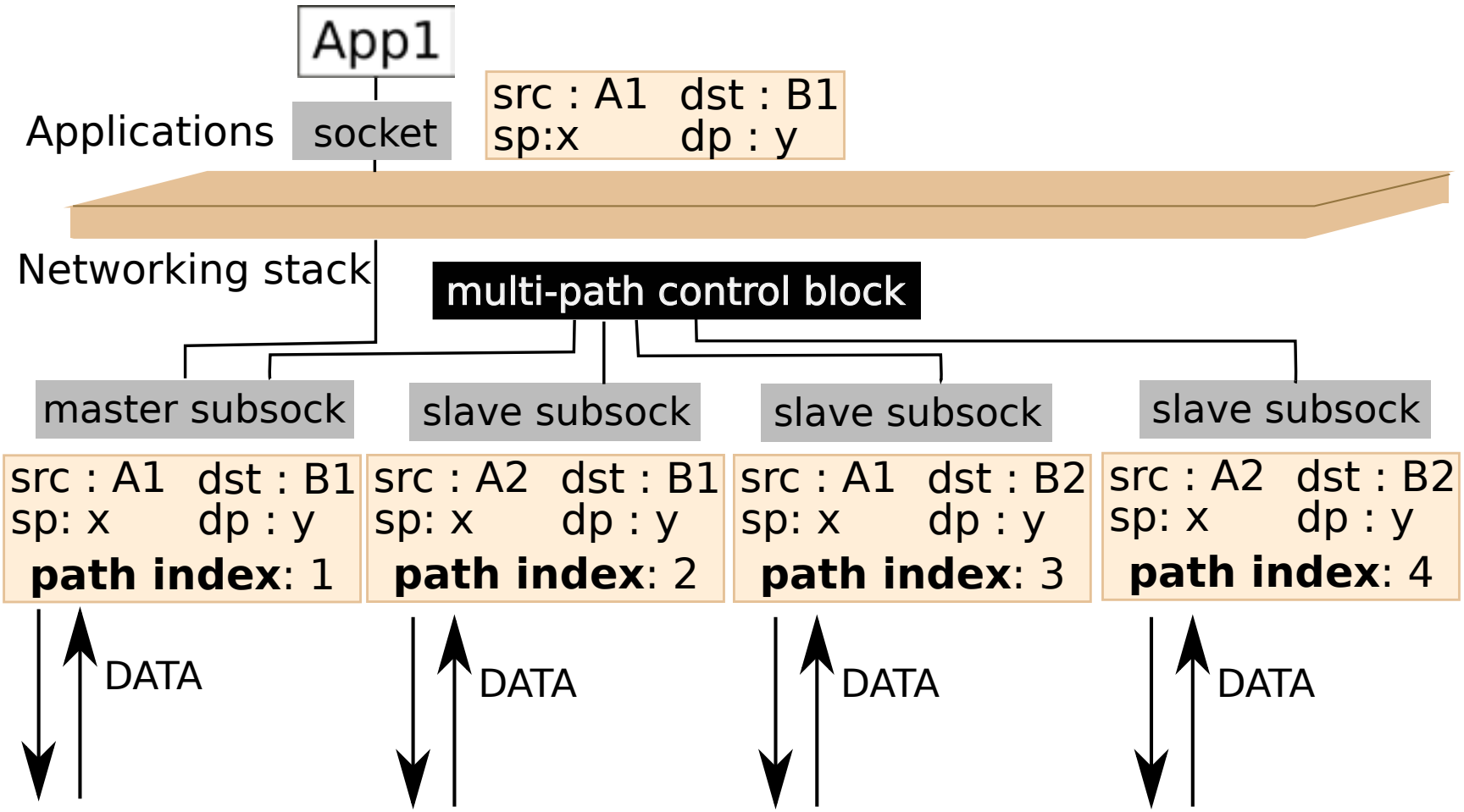
OS implementation discussion

Sébastien Barré and Costin Raiciu

July 24th, 2010

MPTCP implementors workshop - Maastricht

Architecture



Overview of Linux TCP Stack: Packet Sending

- Standard TCP: send syscall
  1. Are there free send buffers? If not, wait.
  2. Copy data from user space and
  3. Create skb(s), add them to send buffer.
  4. Send packets out if allowed by cwnd and rwnd
- Standard TCP: ack reception
  1. Update local data structures: snd.una, cwnd, rcvwnd, etc.
  2. Fast retransmit if necessary
  3. Send packets from send buffer as allowed by cwnd/rwnd

Multipath TCP:Packet Sending (conceptual)

- Multipath TCP: send syscall
  1. Are there free send buffers? If not, wait.
  2. Copy data from user space to a connection-level buffer.
  3. If there are subflows available, do immediately send data
  4. Create skb(s), send data
- Multipath TCP: ack reception
  1. Update local data structures: snd.una, cwnd, rcvwnd, etc.
  2. Fast retransmit if necessary
  3. Create skb(s) and send them as allowed by cwnd/rwnd

Multipath TCP:Packet Sending (single MSS)

- Multipath TCP: send syscall
  1. Are there free send buffers? If not, wait.
  2. Copy data from user space and create skb(s) with min(MSS)
  3. Add skbs to a connection-level skb queue.
  4. If there are subflows available, do immediately send data to those subflows
  5. Send packets out, also moving skb(s) into subflow-level send queue.
- Multipath TCP: ack reception
  1. Update local data structures: snd.una, cwnd, rcvwnd, etc.
  2. Fast retransmit if necessary
  3. Take skb(s) from connection-level queue and send them as allowed by cwnd/rwnd

Multipath TCP: Packet Sending (current solution)

- Multipath TCP: send syscall
  1. Are there free send buffers? If not, wait.
  2. Use scheduler to decide which subflow send buffer should receive user data
  3. Copy data from user space and
  4. Create skb(s), add them to send buffer.
  5. Send packets out if allowed by cwnd and rwnd
- Standard TCP: ack reception
  1. Update local data structures: snd.una, cwnd, rcvwnd, etc.
  2. Fast retransmit if necessary
  3. Send packets from send buffer as allowed by cwnd/rwnd
  4. Might have to "take" packets from another subflow's send buffer.

Multipath TCP: Packet Scheduler

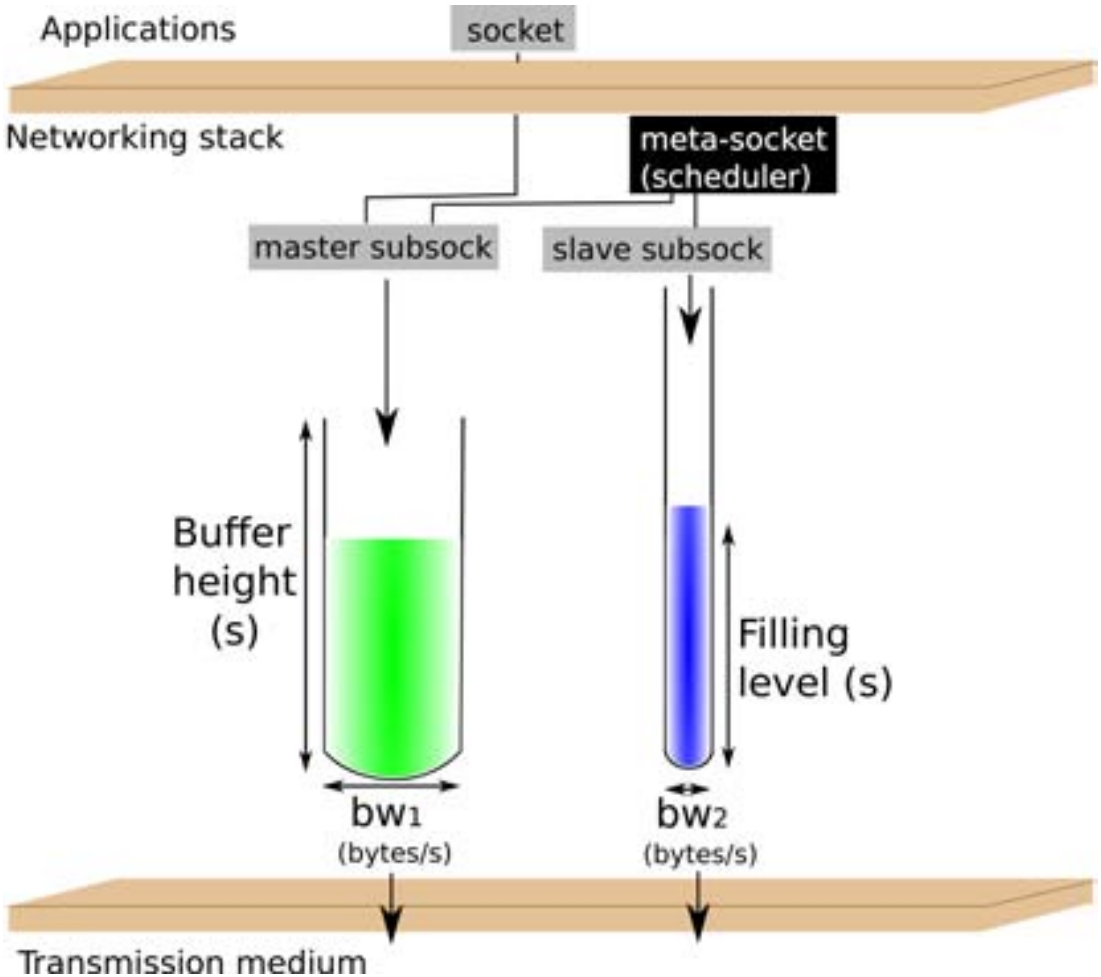


Figure: Send buffer representation

TCP sending buffers

- Must be just big enough to be able to feed a new segment to the network whenever the congestion window is opened.
- Without loss, this means that the sending buffers must have the size of a BDP (bandwidth-delay product), that is, *cwnd*.
- With single loss, and SACK, it must be able to feed another BDP, where the BDP is the amount of bytes transmitted between the fast retransmit and the corresponding acknowledgement.

TCP send buffer tuning

With single flow TCP, $sndbuf = 2 * max(cwnd)$

See Semke,Mahdavi and Mathis, *Automatic TCP Buffer tuning*, CCR, 1998

Multipath TCP: Size of the send buffers

They **must** be of same height to achieve maximum path utilization:

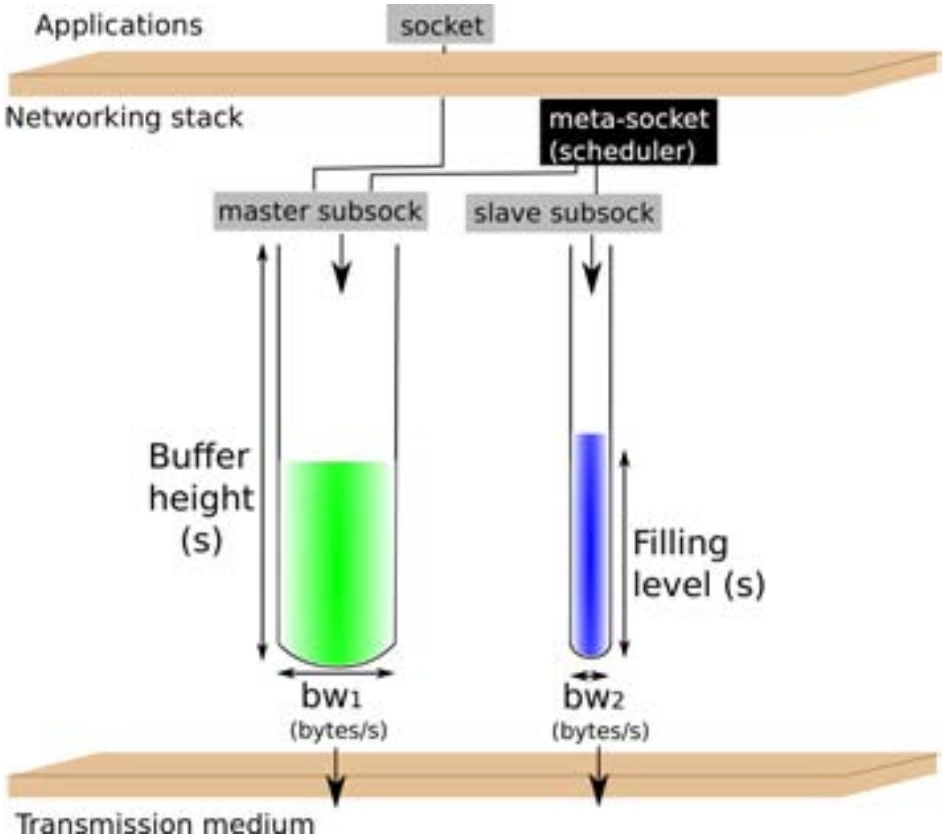


Figure: Send buffer representation

re-scheduling

- When path properties change, a subflow can suddenly be flushed and connection-level reordering happens then.
- As long as a segment has never been transmitted, it can be moved around, although this has a cost.
- Let's do it, when necessary only.

Rescheduling policy

if $\exists$ subflow *i* such that *cwnd closed, sndwnd open* **and** $\exists$ subflow *j* such that *cwnd open, sndwnd closed* **then** reschedule.

re-scheduling

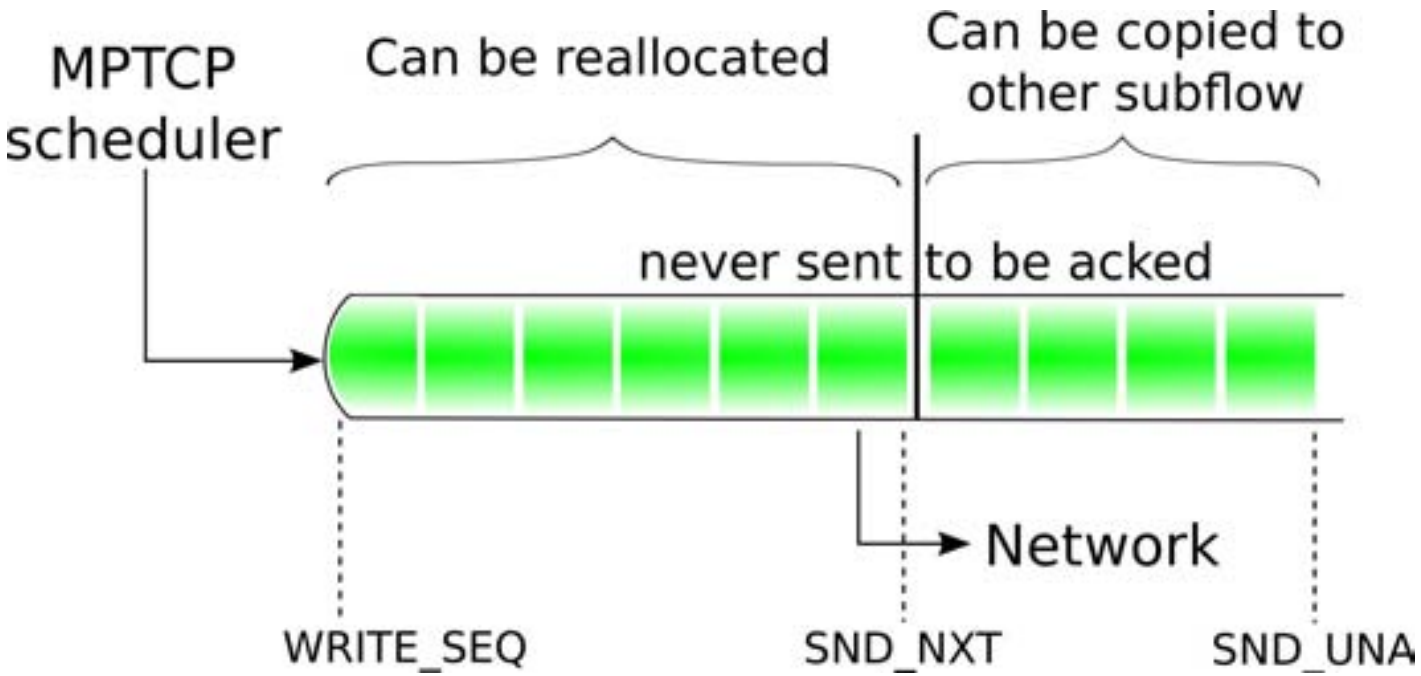


Figure: The subsocket write queue

Estimating the bandwidth

- We could be tempted to use $BW \simeq \frac{cwnd}{SRTT}$. However this has just not the right granularity:
  - cwnd changes at every ack. The experienced throughput is better evaluated with a *mean value* of cwnd, instead of its instantaneous value.
  - We thus estimate the bandwidth each time a cwnd is flushed.
  - Similar approach is used for receiver-side RTT estimation.

Overview of Linux TCP Stack: Packet Receiving

- Standard TCP: packet reception
  1. Is segment in window? If not, send ACK, drop segment
  2. Is this the next expected segment? If yes, add to receive buffer
  3. Segment out of order.Is it the biggest in the ofo queue?
     - Yes: Insert at the end of ofo queue
     - No: Linear walk of ofo queue to find gap.Gap filled? Move segments to receive buffer.
- Standard TCP: recv syscall
  1. Is there available in order data? If not, wait
  2. Copy in-order data from receive buffer to user space.
  3. Free receive buffer

Multipath TCP: Packet Receiving (conceptual)

- Multipath TCP: packet reception
  1. Is segment in subflow window and connection level window? If not, send ACK, drop segment
  2. Is this the next expected segment? If yes, add to connection level receive buffer.
  3. Segment out of order.Is it the biggest in the ofo queue?
     - Yes: Insert at the end of ofo queue
     - No: Linear walk of ofo queue to find gap. Gap filled? Move segments to connection level receive buffer.
- Multipath TCP: recv syscall
  1. Is there available in order data? If not, wait
  2. Copy in-order data from receive buffer to user space.
  3. Free receive buffer

TCP receive buffers

- In the past, receive buffers were fixed to a large value
- But if the application slowly asks for data, that data wastes memory ressources.
- If the BDP is very large and the application is fast, then the receive buffer may become too small.

TCP receive buffer tuning

With single flow TCP, $rcvbuf = 2 * max(BW * estRTT)$

See Fisk, Mike and Feng, *Dynamic Right-sizing in TCP*, 2001

Multipath TCP: More buffers needed

- For each subflow, we may need to buffer data until the slowest subflow has finished a fast retransmit:
  $BW * max_{subflows}(estRTT)$

MPTCP receive buffer tuning

With multipath TCP,
$rcvbuf = 2 * \sum_{subflows} BW * max_{subflows}(estRTT)$

Computing the receive window

- With single path, the receive window represents a portion of the sequence number space.
- With multiple paths, that would create "network-deadlocks". Example:
  - One subflow dies, so we reinject data on another subflow.
  - But that other subflow has currently a zero window, because the receive buffers are full.
  - The receiver waits for those segments to reopen the window.
  - The sender wait that the window is reopened before to send the reinjected segments.

Computing the receive window

- So we *redefine* the receive window as a portion of the **Data Sequence Numbers** space.
- We then announce the same window on all subflows.
- It is computed based on the available aggregate receive buffer.
- data can be (re)transmitted on any subflow, as long as it fits into the most recently received receive window.

Waking up the application correctly

- Many applications will die if you wake them up from select(), then do not give anything to eat upon read().
- stdTCP wakes up when in-order data arrives.
- MPCTP must wake up when in-order DSNs arrives.
- → need to either
  - move segments to the connection-level receive queue as soon as they are received.
  - Defer segment reception until the application needs it, but track the arrival of next DSN.

Unblocking subflows

- stdTCP checks if any data needs to be sent whenever an ack is received.
- due to the shared receive window, MPTCP must do that check, on **all** subflows.
- A received ack can open the sndwnd of another subflow.

Receiving the JOIN option

- Each MPCTP master socket must somehow behave as if it were in LISTEN state.
- Implies much code taken from listening sockets, but tailored to mptcp needs.
- We currently manage a list of pending minisocks attached to the mpcb.
- We must change the lowest level TCP receive function, and check there the presence of the join option. That could be critical for OS insertion.

Connection-level acks

- We need to maintain something similar to a sack-block list.
- But probably need for this can be removed if DATA ack option is mandatory.

Efficient data structures

- ofo queue will often be much more out-of-order than stdTCP.
- If simple list is used, we may need to traverse it very often, and it can be long.
- This is done with socket locked, so can delay the acknowledgement of further received segments.
- Probably some structure enabling binary search should be used.

What is there in the current implementation

- Fall back to legacy TCP.
- Can traverse middleboxes that rewrite the ISN. (subflow seq is relative)
- Supports TSO. (thanks to proper mapping management)
- receive window relative to DSNs.
  - Quite heavy change in implementation !
  - example: ack received on one subflow can trigger data sending on another subflow.
- options: MPC (mod), JOIN, DSN (mod), ADD_ADDR (without port option).
- Scheduler tries to fill all pipes.

TODO (to support draft)

- IPv6 support. Shim6 is supported, but cannot be used together with v4.
- Connection-level FIN. Currently communication terminates when FIN exchange happens on last subflow.
- random generation of token. Currently incremented variable.
- SYN does not occupy DSN seqnum space, but FIN does.
- Use largest receive window across subflows. Currently we use the most recently advertised receive window.

TODO - options

- MPC option:
  - IDSN always 0.
  - No echoing in the SYN/ACK
- DSN is 4 bytes long, not 8, and not randomized, no CRC.
- Data Ack: currently subflow ack translated into data ack by receiver.
- Data FIN: currently FIN on all subflows plays the role of DATA FIN.
- RemoveAddr
- MPFail

TODO - OS features

- DMA support
- TCP fast path - translated to MPCTP
- SMP optimizations.
- SYN cookies

Amount of changes to current stack

file	+	-
net/ipv4/mtcp.c	2163	0
net/ipv4/mtcp_pm.c	1207	0
include/net/mtcp.h	371	0
include/net/mtcp_pm.h	80	0
total	**3821**	**0**
net/ipv4/tcp.c	930	45
net/ipv4/tcp_input.c	860	70
net/ipv4/tcp_output.c	590	87
inlude/net/tcp.h	176	13
net/ipv4/tcp_ipv4.c	71	15
include/linux/tcp.h	69	23
net/ipv4/tcp_minisocks.c	35	5
include/net/request_sock.h	16	0
total	**2747**	**258**

Summary

- **Conceptually**, there are few simple changes needed to implement multipath TCP
- In practice, though, we must deal with stacks optimized for single-path in many ways.
- Our prototype uses a lot of existing code but it can be improved to use even more.
- A lot of challenges ahead to achieve performance:
  - Data structures and algorithms tailored for MPTCP (e.g. global receive buffer).
  - Flow to core affinity issues.
  - The ability to use hardware accelerations (TSO, LRO).

For more information:

- To check the status of this implementation, and future (working) demos of it:
  - `http://inl.info.ucl.ac.be/mptcp`