Network Working Group
Internet-Draft
Expires: September 8, 2011

S. Barre
C. Paasch
O. Bonaventure
UCLouvain, Belgium

March 7, 2011

MultiPath TCP - Guidelines for implementers
draft-barre-mptcp-impl-00

Abstract

Multipath TCP is a major extension to TCP that allows improving the resource usage in the current Internet by transmitting data over several TCP subflows, while still showing one single regular TCP socket to the application. This document describes our experience in writing a MultiPath TCP implementation in the Linux kernel and discusses implementation guidelines that could be useful for other developers who are planning to add MultiPath TCP to their networking stack.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The MultiPath TCP protocol [1] is a major TCP extension that allows for simultaneous use of multiple paths, while being transparent to the applications, fair to regular TCP flows [2] and deployable in the current Internet. The MPTCP design goals and the protocol architecture that allow reaching them are described in [3]. Besides the protocol architecture, a number of non-trivial design choices need to be made in order to extend an existing TCP implementation to support MultiPath TCP. This document gathers a set of guidelines that should help implementers writing an efficient and modular MPTCP stack. The guidelines are expected to be applicable regardless of the Operating System (although the MPTCP implementation described here is done in Linux [4]). Another goal is to achieve the greatest level of modularity without impacting efficiency, hence allowing other multipath protocols to nicely co-exist in the same stack. In order for the reader to clearly disambiguate "useful hints" from "important requirements", we write the latter in their own paragraphs, starting with the keyword "IMPORTANT". By important requirements, we mean design options that, if not followed, would lead to an under-performing MPTCP stack, maybe even slower than regular TCP.

This draft presents implementation guidelines that are based on the code which has been implemented in our MultiPath TCP aware Linux kernel (the version covered here is 0.6) which is available from http://inl.info.ucl.ac.be/mptcp. We also list configuration guidelines that have proven to be useful in practice. In some cases, we discuss some mechanisms that have not yet been implemented. These mechanisms are clearly listed. During our work in implementing MultiPath TCP, we evaluated other designs. Some of them are not used anymore in our implementation. However, we explain in the appendix the reason why these particular designs have not been considered further.

This document is structured as follows. First we propose an architecture that allows supporting MPTCP in a protocol stack residing in an operating system. Then we consider a range of problems that must be solved by an MPTCP stack (compared to a regular TCP stack). In Section 4, we propose recommendations on how a system administrator could correctly configure an MPTCP-enabled host. Finally, we discuss future work, in particular in the area of MPTCP optimization.

1.1. Terminology

In this document we use the same terminology as in [3] and [1]. In addition, we will use the following implementation-specific terms:

- Meta-socket: A socket structure used to reorder incoming data at the connection level and schedule outgoing data to subflows.

- Master subsocket: The socket structure that is visible from the application. If regular TCP is in use, this is the only active socket structure. If MPTCP is used, this is the socket corresponding to the first subflow.

- Slave subsocket: Any socket created by the kernel to provide an additional subflow. Those sockets are not visible to the application (unless a specific API [5] is used). The meta-socket, master and slave subsocket are explained in more details in Section 2.2.

- Endpoint id: Endpoint identifier. It is the tuple (saddr, sport, daddr, dport) that identifies a particular subflow, hence a particular subsocket.

- Fendpoint id: First Endpoint identifier. It is the endpoint identifier of the Master subsocket.

- Connection id or token: It is a locally unique number, defined in Section 2 of [1], that allows finding a connection during the establishment of new subflows.

2. An architecture for Multipath transport

Section 4 of the MPTCP architecture document [3] describes the functional decomposition of MPTCP. It lists four entities, namely Path Management, Packet Scheduling, Subflow Interface and Congestion Control. These entities can be further grouped based on the layer at which they operate:

- Transport layer: This includes Packet Scheduling, Subflow Interface and Congestion Control, and is grouped under the term "Multipath Transport (MT)". From an implementation point of view, they all will involve modifications to TCP.

- Any layer: Path Management. Path management can be done in the transport layer, as is the case of the built-in path manager (PM) described in [1]. That PM discovers paths through the exchange of TCP options of type ADD_ADDR or the reception of a SYN on a new address pair, and defines a path as an endpoint_id (saddr, sport, daddr, dport). But, more generally, a PM could be any module able to expose multiple paths to MPTCP, located either in kernel or user space, and acting on any OSI layer (e.g. a bonding driver that would expose its multiple links to the Multipath Transport).

Because of the fundamental independence of Path Management compared to the three other entities, we draw a clear line between both, and define a simple interface that allows MPTCP to benefit easily from any appropriately interfaced multipath technology. In this document, we stick to describing how the functional elements of MPTCP are defined, using the built-in Path Manager described in [1], and we leave for future separate documents the description of other path managers. We describe in the first subsection the precise roles of the Multipath Transport and the Path Manager. Then we detail how they are interfaced with each other.

2.1. MPTCP architecture

Although, when using the built-in PM, MPTCP is fully contained in the transport layer, it can still be organized as a Path Manager and a Multipath Transport Layer as shown in Figure 1. The Path Manager announces to the MultiPath Transport what paths can be used through path indices for an MPTCP connection, identified by the fendpoint_id (first endpoint id). The fendpoint_id is the tuple (saddr, sport, daddr, dport) seen by the application and uniquely identifies the MPTCP connection (an alternate way to identify the MPTCP connection being the conn_id, which is a token as described in Section 2 of [1]). The Path Manager maintains the mapping between the path_index and an endpoint_id. The endpoint_id is the tuple (saddr, sport, daddr, dport) that is to be used for the corresponding path index.

Note that the fendpoint_id itself represents a path and is thus a particular endpoint_id. By convention, the fendpoint_id is always represented as path index 1. As explained in [3], Section 5.6, it is not yet clear how an implementation should behave in the event of a failure in the first subflow. We expect, however, that the Master subsocket should be kept in use as an interface with the application, even if no data is transmitted anymore over it. It also allows the fendpoint_id to remain meaningful throughout the life of the connection. This behavior has yet to be tested and refined with Linux MPTCP.

Figure 1 shows an example sequence of MT-PM interactions happening at the beginning of an exchange. When the MT starts a new connection (through an application connect() or accept()), it can request the PM to be updated about possible alternate paths for this new connection. The PM can also spontaneously update the MT at any time (normally when the path set changes). This is step 1 in Figure 1. In the example, 4 paths can be used, hence 3 new ones. Based on the update, the MT can decide whether to establish new subflows, and how many of them. Here, the MT decides to establish one subflow only, and sends a request for endpoint_id to the PM. This is step 2. In step 3, the answer is given: <A2,B2,0,pB2>. The source port is unspecified to allow the MT ensure the unicity of the new endpoint_id, thanks to the new_port() primitive (present in regular TCP as well). Note that messages 1,2,3 need not be real messages and can be function calls instead (as is the case in Linux MPTCP).

```
                               Control plane
   +---------------------------------------------------------------+
   |                     Multipath Transport (MT)                  |
   +----------------------------------------------------|----------+
     ^                  |           ^                   v
     |                  |           |          [Build new subsocket,
     | 1.For fendpt_id  |2.endpt_id |                 with endpt_ids
     |<A1,B1,pA1,pB1>   | for path  | 3.<A2,B2,   <A2,B2,new_port(),pB2]
     |Paths 1->4 can be | index 2 ? |   0,pB2>
     |used.             |           |
     |                  |           |
     |                  |           |
     |                  v           |
   +---------------------------------------------------------------+
   |                         Path Manager (PM)                     |
   +---------------------------------------------------------------+
      /                                     \
     /---------------------------------------\
     | mapping table:                        |
     |   Subflow   <--> endpoint_id          |
     |  path index                           |
     |                                       |
     |    [see table below]                  |
     |                                       |
     +---------------------------------------+
```

Figure 1: Functional separation of MPTCP in the transport layer

The following options, described in [1] , are managed by the Multipath Transport:

- o MULTIPATH CAPABLE (MP_CAPABLE): Tells the peer that we support MPTCP and announces our local token.

- o MP_JOIN/MP_AUTH: Initiates a new subflow (Note that MP_AUTH is not yet part of our Linux implementation at the moment)

- o DATA SEQUENCE NUMBER (DSN_MAP): Identifies the position of a set of bytes in the meta-flow.

- o DATA_ACK: Acknowledge data at the connection level (subflow level acknowledgments are contained in the normal TCP header).

- o DATA FIN (DFIN): Terminates a connection.

- o MP_PRIO: Asks the peer to revise the backup status of the subflow on which the option is sent. Although the option is sent by the

Multipath Transport (because this allows using the TCP option space), it may be triggered by the Path Manager. This option is not yet supported by our MPTCP implementation.

- MP_FAIL: Checksum failed at connection-level. Currently the Linux implementation does not implement the checksum in option DSN_MAP, and hence does not implement either the MP_FAIL option.

The Path manager applies a particular technology to give the MT the possibility to use several paths. The built-in MPTCP Path Manager uses multiple IPv4/v6 addresses as its mean to influence the forwarding of packets through the Internet. When the MT starts a new connection, it chooses a token that will be used to identify the connection. This is necessary to allow future subflow-establishment SYNs (that is, containing the MP_JOIN option) to be attached to the correct connection. An example mapping table is given hereafter:

token	path index	Endpoint id
token_1	1	<A1,B1,0,pB1>
token_1	2	<A2,B2,0,pB1>
token_1	3	<A1,B2,0,pB1>
token_1	4	<A2,B1,0,pB1>
token_2	1	<A1,B1,0,pB2>
token_2	2	<A2,B1,0,pB2>

Table 1: Example mapping table for built-in PM

Table 1 shows an example where two MPTCP connections are active. One is identified by token_1, the other one with token_2. As per [1], the tokens must be unique locally. Since the endpoint identifier may change from one subflow to another, the attachment of incoming new subflows (identified by a SYN + MP_JOIN option) to the right connection is achieved thanks to the locally unique token. The built-in path manager currently implements the following options The following options (defined in [1]) are intended to be part of the built-in path manager:

- Add Address (ADD_ADDR): Announces a new address we own

- o Remove Address (REMOVE_ADDR): Withdraws a previously announced address

Those options form the built-in MPTCP Path Manager, based on declaring IP addresses, and carries control information in TCP options. An implementation of Multipath TCP can use any Path Manager, but it must be able to fallback to the default PM in case the other end does not support the custom PM. Alternative Path Managers may be specified in separate documents in the future.

2.2. Structure of the Multipath Transport

The Multipath Transport handles three kinds of sockets. We define them here and use this notation throughout the entire document:

- o Master subsocket: This is the first socket in use when a connection (TCP or MPTCP) starts. It is also the only one in use if we need to fall back to regular TCP. This socket is initiated by the application through the socket() system call. Immediately after a new master subsocket is created, MPTCP capability is enabled by the creation of the meta-socket.

- o Meta-socket: It holds the multipath control block, and acts as the connection level socket. As data source, it holds the main send buffer. As data sink, it holds the connection-level receive queue and out-of-order queue (used for reordering). We represent it as a normal (extended) socket structure in Linux MPTCP because this allows reusing much of the existing TCP code with few modifications. In particular, the regular socket structure already holds pointers to SND.UNA, SND.NXT, SND.WND, RCV.NXT, RCV.WND (as defined in [6]). It also holds all the necessary queues for sending/receiving data.

- o Slave subsocket: Any subflow created by MPTCP, in addition to the first one (the master subsocket is always considered as a subflow even though it may be in failed state at some point in the communication). The slave subsockets are created by the kernel (not visible from the application) The master subsocket and the slave subsockets together form the pool of available subflows that the MPTCP Packet Scheduler (called from the meta-socket) can use to send packets.

2.3. Structure of the Path Manager

In contrast to the multipath transport, which is more complex and divided in sub-entities (namely Packet Scheduler, Subflow Interface and Congestion Control, see Section 2), the Path Manager just maintains the mapping table and updates the Multipath Transport when the mapping table changes. The mapping table has been described above (Table 1). We detail in Table 2 the set of (event,action) pairs that are implemented in the Linux MPTCP built-in path manager. For reference, an earlier architecture for the Path Management is discussed in Appendix A.1. Also, Appendix A.2 proposes a small extension to this current architecture to allow supporting other path managers.

event	action
master_sk bound: This event is triggered upon either a bind(), connect(), or when a new server-side socket becomes established.	Discovers the set of local addresses and stores them in local_addr_table
ADD_ADDR option received or SYN+MP_JOIN received on new address	Updates remote_addr_table correspondingly
local/remote_addr_table updated	Updates mapping_table by adding any new address combinations, or removing the ones that have disappeared. Each address pair is given a path index. Once allocated to an address pair, a path index cannot be reallocated to another one, to ensure consistency of the mapping table.
Mapping_table updated	Sends notification to the Multipath Transport. The notification contains the new set of path indices that the MT is allowed to use. This is shown in Figure 1, msg 1.
Endpoint_id(path_index) request received from MT (Figure 1, msg 2)	Retrieves the endpoint_ids for the corresponding path index from the mapping table and returns them to the MT. One such request/response is illustrated in Figure 1, msg 3. Note that in that msg 3, the local port is set to zero. This is to let the operating system choose a unique local port for the new socket.

Table 2: (event,action) pairs implemented in the built-in PM

3. MPTCP challenges for the OS

MPTCP is a major modification to the MPTCP stack. We have described above an architecture that separates Multipath Transport from Path Management. Path Management can be implemented rather simply. But Multipath Transport involves a set of new challenges, that do not exist in regular TCP. We first describe how an MPTCP client or server can start a new connection, or a new subflow within a connection. Then we propose techniques (a concrete implementation of which is done in Linux MPTCP) to efficiently implement data reception (at the data sink) and data sending (at the data source).

3.1. Charging the application for its CPU cycles

As this document is about implementation, it is important not only to ensure that MPTCP is fast, but also that it is fair to other applications that share the same CPU. Otherwise one could have an extremely fast file transfer, while the rest of the system is just hanging. CPU fairness is ensured by the scheduler of the Operating System when things are implemented in user space. But in the kernel, we can choose to run code in "user context", that is, in a mode where each CPU cycle is charged to a particular application. Or we can (and must in some cases) run code in "interrupt context", that is, interrupting everything else until the task has finished. In Linux (probably a similar thing is true in other systems), the arrival of a new packet on a NIC triggers a hardware interrupt, which in turn schedules a software interrupt that will pull the packet from the NIC and perform the initial processing. The challenge is to stop the processing of the incoming packet in software interrupt as soon as it can be attached to a socket, and wake up the application. With TCP, an additional constraint is that incoming data should be acknowledged as soon as possible, which requires reordering. Van Jacobson has proposed a solution for this [7]: If an application is waiting on a recv() system call, incoming packets can be put into a special queue (called prequeue in Linux) and the application is woken up. Reordering and acknowledgement are then performed in user context. The execution path for outgoing packets is less critical from that point of view, because the vast majority of processing can be done very easily in user context.

In this document, when discussing CPU fairness, we will use the following terms:

- User context: Execution environment that is under control of the OS scheduler. CPU cycles are charged to the associated application, which allows to ensure fairness with other applications.

o Interrupt context: Execution environment that runs with higher priority than any process. Although it is impossible to completely avoid running code in interrupt context, it is important to minimize the amount of code running in such a context.

o VJ prequeues: This refers to Van Jacobson prequeues, as explained above[7].

3.2. At connection/subflow establishment

As described in [1], the establishment of an MPTCP connection is quite simple, being just a regular three-way exchange with additional options. As shown in Section 2.2 this is done in the master subsocket. Currently Linux MPTCP attaches a meta-socket to a socket as soon as it is created, that is, upon a socket() system call (client side), or when a server side socket enters the ESTABLISHED state. An alternate solution is described in Appendix A.3.

An implementation can choose the best moment, maybe depending on the OS, to instantiate the meta-socket. However, if this meta-socket is needed to accept new subflows (like it is in Linux MPTCP), it should be attached at the latest when the MP_CAPABLE option is received. Otherwise incoming new subflow requests (SYN + MP_JOIN) may be lost, requiring retransmissions by the peer and delaying the subflow establishment.

The establishment of subflows, on the other hand, is more tricky. The problem is that new SYNs (with the MP_JOIN option) must be accepted by a socket (the meta-socket in the proposed design) as if it was in LISTEN state, while its state is actually ESTABLISHED. There is the following in common with a LISTEN socket:

o Temporary structure: Between the reception of the SYN and the final ACK, a mini-socket is used as a temporary structure.

o Queue of connection requests: The meta-socket, like a LISTEN socket, maintains a list of pending connection requests. There are two such lists. One contains mini-sockets, because the final ACK has not yet been received. The second list contains sockets in the ESTABLISHED state that have not yet been accepted. "Accepted" means, for regular TCP, returned to the application as a result of an accept() system call. For MPTCP it means that the new subflow has been integrated in the set of active subflows.

We can list the following differences with a normal LISTEN socket.

- o Socket lookup for a SYN: When a SYN is received, the corresponding LISTEN socket is found by using the endpoint_id. This is not possible with MPTCP, since we can receive a SYN on any endpoint_id. Instead, the token must be used to retrieve the meta-socket to which the SYN must be attached. A new hashtable must be defined, with tokens as keys.

- o Lookup for connection request: In regular TCP, this lookup is quite similar to the previous one (in Linux at least). The 5-tuple is used, first to find the LISTEN socket, next to retrieve the corresponding mini-socket, stored in a private hashtable inside the LISTEN socket. With MPTCP, we cannot do that, because there is no way to retrieve the meta-socket from the final ACK. The 5-tuple can be anything, and the token is only present in the SYN. There is no token in the final ACK. Our Linux MPTCP implementation uses a global hashtable for pending connection requests, where the key is the 5-tuple of the connection request.

An implementation must carefully check the presence of the MP_JOIN option in incoming SYNs before performing the usual socket lookup. If it is present, only the token-based lookup must be done. If this lookup does not return a meta-socket, the SYN must be discarded. Failing to do that could lead to mistakenly attach the incoming SYN to a LISTEN socket instead of attaching it to a meta-socket.

3.3. Subflow management

Further research is needed to define the appropriate heuristics to solve these problems. Initial thoughts are provided in Appendix B.1.

Currently, in a Linux MPTCP client, the Multipath Transport tries to open all subflows advertised by the Path Manager. On the other hand, the server only accepts new subflows, but does not try to establish new ones. The rationale for this is that the client is the connection initiator. New subflows are only established if the initiator requests them. This is subject to change in future releases of our MPTCP implementation.

3.4. At the data sink

There is a symmetry between the behavior of the data source and the data sink. Yet, the specific requirements are different. The data sink is described in this section while the data source is described in the next section.

3.4.1. Receive buffer tuning

The MPTCP required receive buffer is larger than the sum of the buffers required by the individual subflows. The reason for this and proper values for the buffer are explained in [3] Section 5.3. Not following this could result in the MPTCP speed being capped at the bandwidth of the slowest subflow.

An interesting way to dynamically tune the receive buffer according the bandwidth/delay product (BDP) of a path, for regular TCP, is described in [8] and implemented in recent Linux kernels. It uses the COPIED_SEQ sequence variable (sequence number of the next byte to copy to the app buffer) to count, every RTT, the number of bytes received during that RTT. This number of bytes is precisely the BDP. The accuracy of this technique is directly dependent on the accuracy of the RTT estimation. Unfortunately, the data sink does not have a reliable estimate of the SRTT. To solve this, [8] proposes two techniques:

1. Using the timestamp option (quite accurate).

2. Computing the time needed to receive one RCV.WND [6] worth of data. It is less precise and is used only to compute an upper bound on the required receive buffer.

As described in [1], section 3.3.3, the MPTCP advertised receive window is shared by all subflows. Hence, no per-subflow information can be deduced from it, and the second technique from [8] cannot be used. [3] mentions that the allocated connection-level receive buffer should be 2*sum(BW_i)*RTT_max, where BW_i is the bandwidth seen by subflow i and RTT_max is the maximum RTT estimated among all the subflows. This is achieved in Linux MPTCP by slightly modifying the first tuning algorithm from [8], and disabling the second one. The modification consists in counting on each subflow, every RTT_max the number of bytes received during that time on this subflow. Per subflow, this provides its contribution to the total receive buffer of the connection. This computes the contribution of each subflow to the total receive buffer of the connection.

3.4.2. Receive queue management

As advised in [1], Section 3.3.1, "subflow-level processing should be undertaken separately from that at connection-level". This also has the side-effect of allowing much code reuse from the regular TCP stack. A regular TCP stack (in Linux at least) maintains a receive queue (for storing incoming segments until the application asks for them) and an out-of-order queue (to allow reordering). In Linux MPTCP, the subflow-level receive-queue is not used. Incoming segments are reordered at the subflow-level, just as if they were plain TCP data. But once the data is in-order at the subflow level, it can be immediately handed to MPTCP (See Figure 7 of [3]) for connection-level reordering. The role of the subflow-level receive queue is now taken by the MPTCP-level receive queue. In order to maximize the CPU cycles spent in user context (see Section 3.1), VJ prequeues can be used just as in regular TCP (they are not yet supported in Linux MPTCP, though).

An alternate design, where the subflow-level receive queue is kept active and the MPTCP receive queue is not used, is discussed in Appendix A.4.

3.4.3. Scheduling data ACKs

As specified in [1], Section 3.3.2, data ACKs not only help the sender in having a consistent view of what data has been correctly received at the connection level. They are also used as the left edge of the advertised receive window.

In regular TCP, if a receive buffer becomes full, the receiver announces a receive window. When finally some bytes are given to the application, freeing space in the receive buffer, a duplicate ACK is sent to act as a window upate, so that the sender knows it can transmit again. Likewise, when the MPTCP shared receive buffer becomes full, a zero window is advertised. When some bytes are delivered to the application, a duplicate DATA_ACK must be sent to act as a window update. Such an important DATA_ACK should be sent on all subflows, to maximize the probability that at least one of them reaches the peer. If, however, all DATA_ACKs are lost, there is no other option than relying on the window probes periodically sent by the data source, as in regular TCP.

In theory a DATA_ACK can be sent on any subflow, or even on all subflows, simultaneously. As of version 0.5, Linux MPTCP simply adds the DATA_ACK option to any outgoing segment (regardless of whether it is data or a pure ACK). There is thus no particular DATA_ACK scheduling policy. The only exception is for a window update that follows a zero-window. In this case, the behavior is as described in the previous paragraph.

3.5. At the data source

In this section we mirror the topics of the previous section, in the case of a data sender. The sender does not have the same view of the communication, because one has information that the other can only estimate. Also, the data source sends data and receives acknowledgements, while the data sink does the reverse. This results in a different set of problems to be dealt with by the data source.

3.5.1. Send buffer tuning

As explained in [3], end of Section 5.3, the send buffer should have the same size as the receive buffer. At the sender, we don't have the RTT estimation problem described in Section 3.4.1, because we can reuse the built-in TCP SRTT (smoothed RTT). Moreover, the sender has the congestion window, which is itself an estimate of the BDP, and is used in Linux to tune the send buffer of regular TCP. Unfortunately, we cannot use the congestion window with MPTCP, because the buffer equation does not involve the product BW_i*delay_i for the subflows (which is what the congestion window estimates), but it involves BW_i*delay_max, where delay_max is the maximum observed delay across all subflows. An obvious way to compute the contribution of each subflow to the receive buffer would be: 2*(cwnd_i/SRTT_i)*SRTT_max. However, some care is needed because of the variability of the SRTT (measurements show that, even smoothed, the SRTT is not quite stable). Currently Linux MPTCP estimates the bandwidth periodically by checking the sequence number progress. This however introduces new mechanisms in the kernel, that could probably be avoided. Future experience will tell what is appropriate.

3.5.2. Send queue management

As MultiPath TCP involves the use of several TCP subflows, a scheduler must be added to decide where to send each byte of data. Two possible places for the scheduler have been evaluated for Linux MPTCP. One option is to schedule data as soon as it arrives from the application buffer. This option, consisting in _pushing_ data to subflows as soon as it is available, was implemented in older versions of Linux MPTCP and is now abandoned. We keep a description of it (and why it has been abandoned) in Appendix A.5. Another option is to store all data centrally in the Multipath Transport, inside a shared send buffer (see Figure 2). Scheduling is then done at transmission time, whenever any subflow becomes ready to send more data (usually due to acknowledgements having opened space in the congestion window). In that scenario, the subflows _pull_ segments from the shared send queue whenever they are ready. Note that several subflows can become ready simultaneously, if an acknowledgement advertises a new receive window, that opens more space in the shared send window. For that reason, when a subflow pulls data, the Packet Scheduler is run and other subflows may be fed by the Packet Scheduler in the same time.

```
                              Application
                                   |
                                   v
                                 | * |
    Next segment to send (A)  -> | * |
                                 |---| <- Shared send queue
   Sent, but not DATA-acked(B)-> |_*_|
                                   |
                                   v
                            Packet Scheduler
                                  /  \
                                 /    \
                                |      |
                                v      v
   Sent, but not acked(B)  ->  |_|    |_| <- Subflow level congestion
                                |      |     window
                                v      v
                               NIC    NIC
```

Figure 2: Send queue configuration

This approach, similar to the one proposed in [9], presents several advantages:

- o Each subflow can easily fill its pipe. (As long as there is data to pull from the shared send buffer, and the scheduler is not applying a policy that restricts the subflow).

- o If a subflow fails, it will no longer receive acknowledgements, and hence will naturally stop pulling from the shared send buffer. This removes the need for an explicit "failed state", to ensure that a failed subflow does not receive data (As opposed to e.g. SCTP-CMT, that needs an explicit marking of failed subflows by design, because it uses a single sequence number space [10]).

- o Similarly, when a failed subflow becomes active again, the pending segments of its congestion window are finally acknowledged, allowing it to pull again from the shared send buffer. Note that in such a case, the acknowledged data is normally just dropped by the receiver, because the corresponding segments have been retransmitted on another subflow during the failure time.

Despite the adoption of that approach in Linux MPTCP, there are still two drawbacks:

- o There is one single queue, in the Multipath Transport, from which all subflows pull segments. In Linux, queue processing is optimized for handling segments, not bytes. This implies that the shared send queue must contain pre-built segments, hence requiring the _same_ MSS to be used for all subflows. We note however that today, the most commonly negotiated MSS is around 1380 bytes [4], so this approach sounds reasonable. Should this requirement become too constraining in the future, a more flexible approach could be devised (e.g., supporting a few Maximum Segment Sizes).

o Because the subflows pull data whenever they get new free space in their congestion window, the Packet Scheduler must run at that time. But that time most often corresponds to the reception of an acknowledgement, which happens in interrupt context (see Section 3.1). This is both unfair to other system processes, and slightly inefficient for high speed communications. The problem is that the packet scheduler performs more operations that the usual "copy packet to NIC". One way to solve this problem would be to have a small subflow-specific send queue, which would actually lead to a hybrid architecture between the pull approach (described here) and the push approach (described in Appendix A.5). Doing that would require solving non-trivial problems, though, and requires further study.

As shown, in Figure 2, a segment first enters the shared send queue, then, when reaching the bottom of that queue, it is pulled by some subflow. But to support failures, we need to be able to move segments from one subflow to another, so that the failure is invisible from the application. In Linux MPTCP, the segment data is kept in the Shared send queue (B portion of the queue). When a subflow pulls a segment, it actually only copies the control structure (struct sk_buff) (which Linux calls packet cloning) and increments its reference count. The following event/action table summarizes these operations:

event	action
Segment acknowledged at subflow level	Remove references to the segment from the subflow-level queue
Segment acknowledged at connection level	Remove references to the segment from the connection-level queue

Timeout (subflow-level)	Push the segment to the best running subflow (according to the Packet Scheduler). If no subflow is available, push it to a temporary retransmit queue (not represented in Figure 2) for future pulling by an available subflow. The retransmit queue is parallel to the connection level queue and is read with higher priority.
Ready to put new data on the wire (normally triggered by an incoming ack)	If the retransmit queue is not empty, first pull from there. Otherwise, then take new segment(s) from the connection level send queue (A portion). The pulling operation is a bit special in that it can result in sending a segment over a different subflow than the one which initiated the pull. This is because the Packet Scheduler is run as part of the pull, which can result in selecting any subflow. In most cases, though, the subflow which originated the pull will get fresh data, given it has space for that in the congestion window. Note that the subflows have no A portion in Figure 2, because they immediately send the data they pull.

Table 3: (event,action) pairs implemented in the Multipath Transport queue management

IMPORTANT: A subflow can be stopped from transmitting by the congestion window, but also by the send window (that is, the receive window announced by the peer). Given that the receive window has a connection level meaning, a DATA_ACK arriving on one subflow could unblock another subflow. Implementations should be aware of this to avoid stalling part of the subflows in such situations. In the case of Linux MPTCP, that follows the above architecture, this is ensured by running the Packet Scheduler at each pull operation. This is not completely optimal, though, and may be revised when more experience is gained.

3.5.3. Scheduling data

As several subflows may be used to transmit data, MPTCP must select a subflow to send each data. First, we need to know which subflows are available for sending data. The mechanism that controls this is the congestion controller, which maintains a per-subflow congestion window. The aim of a Multipath congestion controller is to move data away from congested links, and ensure fairness when there is a shared bottleneck. The handling of the congestion window is explained in

Section 3.5.3.1. Given a set of available subflows (according to the congestion window), one of these has to be selected by the Packet Scheduler. The role of the Packet Scheduler is to implement a particular policy, as will be explained in Section 3.5.3.2.

3.5.3.1. The congestion controller

The Coupled Congestion Control provided in Linux MPTCP implements the algorithm defined in [2]. Operating System kernels (Linux at least) do not support floating-point numbers for efficiency reasons. [2] makes an extensive use of them, which must be worked around. Linux MPTCP solves that by performing fixed-point operations using a minimum number of fractions and performs scaling when divisions are necessary.

Linux already includes a work-around for floating point operations in the Reno congestion avoidance implementation. Upon reception of an ack, the congestion window (counted in segments, not in bytes as proposed in [2] does) should be updated as cwnd+=1/cwnd. Instead, Linux increments the separate variable snd_cwnd_cnt, until snd_cwnd_cnt>=cwnd. When this happens, snd_cwnd_cnt is reset, and cwnd is incremented. Linux MPTCP reuses this to update the window in the CCC (Coupled Congestion Control) congestion avoidance phase: snd_cwnd_cnt is incremented as previously explained, and cwnd is incremented when snd_cwnd_cnt >= max(tot_cwnd / alpha, cwnd) (see [2]). Note that the bytes_acked variable, present in [2], is not included here because Linux MPTCP does not currently support ABC [11], but instead considers acknowledgements in MSS units. Linux uses for ABC, in Reno, the bytes_acked variable instead of snd_cwnd_cnt. For Reno, cwnd is incremented by one if bytes_acked>=cwnd*MSS. Hence, in the case of a CCC with ABC, one would increment cwnd when bytes_acked>=max(tot_cwnd*MSS / alpha, cwnd*MSS).

Unfortunately, the alpha parameter mentioned above involves many fractions. The current implementation of MPTCP uses a rewritten version of the alpha formula from [2]:

```
                          cwnd_max * scale_num
alpha = tot_cwnd * ----------------------------------
                  /     rtt_max * cwnd_i * scale_den \ 2
                  | sum ------------------------------|
                  \  i              rtt_i            /
```

This computation assumes that the MSS is shared by all subflows, which is true under the architecture described in Section 3.5.2 but implies that implementations choosing to support several MSS cannot use the above simplified equation. The variables cwnd_max and rtt_max in the above equation are NOT resp. the maximum congestion window and RTT across all subflows. Instead, they are the values of subflow i such that cwnd_i / rtt_i^2 is maximum. This corresponds to the numerator of the equation provided in [2].

scale_num and scale_den have to be selected in such a way that scale_num > scale_den^2. A good choice is to use scale_num=2^32 (using 64 bits arithmetic) and scale_den=2^10. In that case the final alpha value is scaled by 2^12, which gives a reasonable precision. Due to the scaling, it is necessary to also scale later in the formula that decides whether an increase of the congestion window is necessary or not: snd_cwnd_cnt >= max((tot_cwnd<<12) / alpha,cwnd).

3.5.3.2. The Packet Scheduler

Whenever the Congestion Controller (described above) allows new data for at least one subflow, the Packet Scheduler is run. When only one subflow is available the Packet Scheduler just decides which packet to pick from the A section of the shared send buffer (see Figure 2). Currently Linux MPTCP picks the bottom most segment. If more than one subflow is available, there are three decisions to take:

- Which of the subflows to feed with fresh data: As the only Packet Scheduler currently supported in Linux MPTCP aims at filling all pipes, it always feeds data to all subflows as long as there is data to send.

- In what order to feed selected subflows: when several subflows become available simultaneously, they are fed by order of time-distance to the client. We define the time-distance as the time needed for the packet to reach the peer if given to a particular subflow. This time depends on the RTT, bandwidth and queue size (in bytes), as follows: time_distance_i = queue_size_i/bw_i+RTT_i. Given that with the architecture described in Section 3.5.2, the subflow-specific queue size cannot exceed a congestion window, the time_distance becomes time_distance_i~=RTT_i. This scheduling policy favors fast subflows for application-limited communications (where all subflows need not be used). However, for network-limited communications, this scheduling policy has little effect because all subflows will be used at some point, even the slow ones, to try minimizing the connection-level completion time.

- How much data to allocate to a single subflow: this question concerns the granularity of the allocation. Using big allocation units allows for better support of TCP Segmentation Offload (TSO). TSO allows the system to aggregate several times the MSS into one single segment, sparing memory and CPU cycles, by leaving the fragmentation task to the NIC. However, this is only possible if the large single segment is made of contiguous data, at the subflow level and the connection level (see also important note below).

IMPORTANT: When scheduling data to subflows, an implementation must be careful that if two segments are contiguous at the subflow-level, but non-contiguous at the connection level, they cannot be aggregated into one. As Linux (and probably other systems) merges segments when it is under memory pressure, it could easily decide to merge non-contiguous MPTCP segments, simply because they look contiguous from the subflow viewpoint. This must be avoided, because the DATA_SEQ mapping option would loose its meaning in such a case, leading to all possible kinds of misbehaviors.

3.6. At connection/subflow termination

In Linux MPTCP, subflows are terminated only when the whole connection terminates, because the heuristic for terminating subflows (without closing the connection) is not yet mature, as explained in Section 3.3.

At connection termination, an implementation must ensure that all subflows plus the meta-socket are cleanly removed. The obvious choice to propagate the close() system call on all subflows does not work. The problem is that a close() on a subflow appends a FIN at the end of the send queue. If we transpose this to the meta-socket, we would append a DATA_FIN on the shared send queue (see Section 3.5.2). That operation results in the shared send queue not accepting any more data from the application, which is correct. It also results in the subflow-specific queues not accepting any more data from the shared send queue. The shared send queue may however still be full of segments, which will never be sent because all gates are closed.

IMPORTANT: Upon a close() system call, an implementation must refrain from sending a FIN on all subflows, unless the implementation uses an architecture with no connection-level send queue (like the one described in Appendix A.5). Even in that case, it makes sense to keep all subflows open until the last byte is sent, to allow retransmission on any path, should any one of them fail.

Currently, upon a close() system call, Linux MPTCP appends a DATA_FIN to the connection-level send queue. Only when that DATA_FIN reaches the bottom of the send queue is the regular FIN sent on all subflows.

DISCUSSION: In the Linux MPTCP behavior described above, a connection could still stall near its end if one path fails while transmitting its last congestion window of data (because the maximum size of the subflow-specific send queue is cwnd). This can be avoided by waiting just a bit more before to trigger the subflow-FIN: Instead of sending the FIN together with the DATA_FIN, send the DATA_FIN alone and wait for the corresponding DATA_ACK to trigger a FIN on all subflows. This however augments by one RTT the duration of the overall connection termination.

4. Configuring the OS for MPTCP

Previous sections concentrated on implementations. In this section, we try to gather guidelines that help getting the full potential from MPCTP through appropriate system configuration. Currently those guidelines apply especially to Linux, but the principles can be applied to other systems.

4.1. Source address based routing

As already pointed out by [12], the default behavior of most operating systems is not appropriate for the use of multiple interfaces. Most operating systems are typically configured to use at most one IP address at a time. It is more and more common to maintain several links in up state (e.g. using the wired interface as main link, but maintaining a ready-to-use wireless link in the background, to facilitate fallback when the wired link fails). But MPTCP is not about that. MPTCP is about _simultaneously_ using several interfaces (when available). It is expected that one of the mostly used MPTCP configurations will be through two or more NICs, each being assigned a different address. Another possible configuration would be to assign several IP addresses to the same interface, in which case the path diverges later in the network, based on the particular address that is used in the packet.

Usually an operating system has a single default route, with a single source IP address. If the host has several IP addresses and we want to do MultiPath TCP, it is necessary to configure source address based routing. This means that based on the source address, selected by the MultiPath TCP-module in the operating system, the routing-decision is based on a different routing table. Each of these routing tables defines a default route to the Internet. This is different from defining several default routes in the same routing table (which is also supported in Linux), because in that case only the first one is used. Any additional default route is considered as a fallback route, used only in case the main one fails.

It is easier to understand the necessary configuration by means of an example. Let a host have two interfaces,I1 and I2, both connected to the public Internet and being assigned addresses resp. A1 and A2. Such a host needs 3 routing tables. One of them is the classical default routing table, present in all systems. The default routing table is used to find a route based on the destination address only, when a segment is issued with the undetermined source address. The undetermined source address is typically used by applications that initiate a TCP connect() system call, specifying the destination address but letting the system choose the source address. In that case, after the default routing table has been consulted, an address is assigned to the socket by the system by applying [13]. The additional routing tables are used when the source address is specified. If the source address has no impact on the route that should be chosen, then the default routing table is sufficient. But this is a particular case (e.g., a host connected to one network only, but using two addresses to exploit ECMP paths later in the network). In most cases, a source address is attached to a specific interface, or at least a specific gateway. Both of those cases require defining a separate routing table, one per (gateway, outgoing interface) pair. To select the proper routing table based on the source address, an additional indirection level must be configured. It is called "policy routing" in Linux and is illustrated at the bottom of Figure 3.

```
+-----------------------------------------------------+
|                   Default Table                     |
+-----------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP1 Dev: I1            |
| Dst: 0.0.0.0/0  Via: Gateway-IP2 Dev: I2            |
| Dst: Gateway1-Subnet Dev: I1 Src: A1  Scope: Link   |
| Dst: Gateway2-Subnet Dev: I2 Src: A2  Scope: Link   |
+-----------------------------------------------------+

+-----------------------------------------------------+
|                      Table 1                        |
+-----------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP1 Dev: I1            |
| Dst: Gateway1-Subnet Dev: I1 Src: A1 Scope: Link    |
+-----------------------------------------------------+

+-----------------------------------------------------+
|                      Table 2                        |
+-----------------------------------------------------+
| Dst: 0.0.0.0/0  Via: Gateway-IP2 Dev: I2            |
| Dst: Gateway2-Subnet Dev: I2 Src: A2 Scope: Link    |
+-----------------------------------------------------+

+-----------------------------------------------------+
|                   Policy Table                      |
+-----------------------------------------------------+
|   If src == A1 , Table 1                            |
|   If src == A2 , Table 2                            |
+-----------------------------------------------------+
```

Figure 3: Routing table configuration for MultiPath TCP

If only the default routing table were used, only the first default route would be used, regardless of the source address. For example, a packet with source address A2, would leave the host through interface I1, which is incorrect.

4.2. Buffer configuration

[3], Section 5.3 describes in details the new, higher buffer requirements of MPTCP. Section 3.4.1 and Section 3.5.1 describe how the MPTCP buffers can be tuned dynamically. However, it is important to note that even the best tuning is capped by a maximum configured at the system level. When using MultiPath TCP, the maximum receive and send buffer should be configured to a higher value than for regular TCP. There is no universal guideline on what value is best there. Instead the most appropriate action, for an administrator, is probably to roughly estimate the maximum bandwidth and delay that can be observed on a particular connectivity setup, and apply the equation from [3], Section 5.3 to find a reasonable tradeoff. This exercise could lead an administrator to decide to disable MPTCP on some interfaces, because it allows consuming less memory while still achieving reasonable performance.

5. Future work

A lot of work has yet to be done, and there is much space for improvements. In this section we try to assemble a list of future improvements that would complete this guidelines.

- Today's host processors have more and more CPU cores. Given Multipath TCP tries to exploit another form of parallelism, there is a challenge in finding how those they can work together optimally. An important question is how to work with hardware that behaves intelligently with TCP (e.g. flow to core affinity). This problem is discussed in more details in [14].

- An evaluation of Linux MPTCP exists [4]. But many optimizations are still possible and should be evaluated. Examples of them VJ prequeues (Section 3.1), MPTCP fast path (that is, a translation of the existing TCP fast path to MPTCP) or DMA support. VJ prequeues, described in Section 3.1, are intended to defer segment processing until the application is awoken, when possible.

- Currently, support for TCP Segmentation Offload remains a challenge because it plays with the Maximum Segment Size. Linux MPTCP currently works with a single MSS across all subflows (see Section 3.5.2). Adding TSO support to MPTCP is certainly possible, but requires further work (Section 3.5.2). Also, support for Large Receive Offload has not been investigated yet.

- There are ongoing discussions on heuristics that would be used to decide when to start new subflows. Those discussions are summarized in Appendix B.1, but none of the proposed heuristics have been evaluated yet.

6. Acknowledgements

Sebastien Barre, Christoph Paasch and Olivier Bonaventure are supported by Trilogy (http://www.trilogy-project.org), a research project (ICT-216372) partially funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.

The authors gratefully acknowledge Costin Raiciu, who wrote a userland implementation of MPTCP and provided insight on implementation matters during several fruitful debates. Discussions with Janardhan Iyengar also helped understanding the specificities of MPTCP compared to SCTP-CMT.

The authors would also like to thank the following people for useful discussions on the mailing list and/or reviews: Alan Ford, Bob Briscoe, Mark Handley, Michael Scharf.

7. References

[1] Ford, A., Raiciu, C., and M. Handley, "TCP Extensions for Multipath Operation with Multiple Addresses", draft-ietf-mptcp-multiaddressed-02 (work in progress), October 2010.

[2] Raiciu, C., Handley, M., and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols", draft-ietf-mptcp-congestion-01 (work in progress), January 2011.

[3] Ford, A., Raiciu, C., Handley, M., Barre, S., and J. Iyengar, "Architectural Guidelines for Multipath TCP Development", draft-ietf-mptcp-architecture-05 (work in progress), January 2011.

[4] Barre, S., Paasch, C., and O. Bonaventure, "Multipath TCP: From Theory to Practice", IFIP Networking,Valencia , May 2011, <http://inl.info.ucl.ac.be/publications/mptcp-nw>.

[5] Scharf, M. and A. Ford, "MPTCP Application Interface Considerations", draft-ietf-mptcp-api-00 (work in progress), November 2010.

[6] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[7] Jacobson, V., "Re: query about tcp header on tcp-ip", Sep 1993, <ftp://ftp.ee.lbl.gov/email/vanj.93sep07.txt>.

[8] Fisk, M. and W. Feng, "Dynamic right-sizing in TCP", Los Alamos Computer Science Institute Symposium , 2001, <http://woozle.org/~mfisk/papers/tcpwindow-lacsi.pdf>.

[9] Hsieh, H. and R. Sivakumar, "pTCP: An End-to-End Transport Layer Protocol for Striped Connections", ICNP , 2002, <http://www.ece.gatech.edu/research/GNAN/archive/2002/icnp02h.html>.

[10] Becke, M., Dreibholz, T., Iyengar, J., Natarajan, P., and M. Tuexen, "Load Sharing for the Stream Control Transmission Protocol (SCTP)", draft-tuexen-tsvwg-sctp-multipath-01 (work in progress), December 2010.

[11] Allman, M., "TCP Congestion Control with Appropriate Byte Counting (ABC)", RFC 3465, February 2003.

[12] Blanchet, M. and P. Seite, "Multiple Interfaces and

Provisioning Domains Problem Statement", draft-ietf-mif-problem-statement-09 (work in progress), October 2010.

[13] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2003.

[14] Watson, R., "Protocol stacks and multicore scalability", Presentation at Maastricht MPTCP workshop , Jul 2010, <http://www.informatics.sussex.ac.uk/research/projects/ngn/slides/msn10talks/watson-stack.pdf>.

Appendix A. Design alternatives

In this appendix, we describe alternate designs that have been considered previously, and abandoned for various reasons (detailed as well). We keep them here for the archive and possible discussion. We also describe some potential designs that have not been explored yet but could reveal to be better in the future, in which case that would be moved to the draft body.

A.1. Another way to consider Path Management

In a previous implementation of MPTCP, it was proposed that the multipath transport had an even more abstract view of the paths in use than what is described in Section 2. In that design, the sub-sockets all shared the same tuple (saddr,sport,daddr,dport), and was disambiguated only by the path index. The advantage is that the Multipath Transport needs only to worry about how to efficiently spread data among multiple paths, without any knowledge about the addresses or ports used by each particular subflow.

That design was particularly well suited for using Shim6 as a Path Manager, because Shim6 is already designed to work in the network layer and rewrite addresses. The first version of the Linux MPTCP implementation was using Shim6 as path manager. It looks also well suited to path managers that don't use addresses (e.g. path managers that write a label in the packet header, later interpreted by the network). Finally, it removes the need for the token in the multipath transport (connection identification is done naturally with the tuple, shared by all subflows). The token hence becomes specific to the built-in path manager, and can be just ignored with other path managers (the context tag plays a similar role in shim6, nothing is needed if the path manager just sets labels to the packets).

However, this cleaner separation between Multipath Transport and Path Management suffers from three drawbacks:

- It requires a heavy modification to the existing stacks, because it modifies the current way to identify sockets in the stack. They are currently unambiguously identified with the usual 5-tuple. This architecture would require extending the 5-tuple with the path index, given all subflows would share the same 5-tuple.

- Although correctly implemented stacks could handle that new endpoint identifier (5-tuple+path index), having several flows with same 5-tuple could confuse middleboxes.

- When the path manager involves using several addresses, forcing the same 5-tuple for all subflows at the Multipath Transport level implies that the Path Manager needs to rewrite the address fields of each packet. That rewriting operation is simply avoided if the sockets are bound to the addresses actually used to send the packets. Hence, this alternate design would involve avoidable costs for path managers that belong to the "multi-address" category.

A.2. Implementing alternate Path Managers

In Section 2, the Path Manager is defined as an entity that maintains a (path_index<->endpoint_id) mapping. This is enough in the case of the built-in path manager, because the segments are associated to a path within the socket itself, thanks to its endpoint_id. However, it is expected that most other path managers will need to apply a particular action, on a per-packet basis, to associate them with a path. Example actions could be writing a number in a field of the segment or choosing a different gateway than the default one in the routing table. In an earlier version of Linux MPTCP, based on a Shim6 Path Manager, the action was used and consisted in rewriting the addresses of the packets.

To reflect the need for a per-packet action, the PM mapping table (an example of which is given in Table 1) only needs to be extended with an action field. As an example of this, we show hereafter an example mapping table for a Path Manager based on writing the path index into a field of the packets.

token	path index	Endpoint id	Action (Write x in DSCP)
token_1	1	<A1,B1,0,pB1>	1
token_1	2	<A1,B1,0,pB1>	2
token_1	3	<A1,B1,0,pB1>	3
token_1	4	<A1,B1,0,pB1>	4
token_2	1	<A1,B1,0,pB2>	1
token_2	2	<A1,B1,0,pB2>	2

Table 4: Example mapping table for a label-based PM

A.3. When to instantiate a new meta-socket ?

The meta-socket is responsible only for MPTCP-related operations. This includes connection-level reordering for incoming data, scheduling for outgoing data, and subflow management. A natural choice then would be to instantiate a new meta-socket only when the peer has told us that it supports MPTCP. In the server it is naturally the case since the master subsocket is created upon the reception of a SYN+MP_CAPABLE. The client, however, instantiates its master subsocket when the application issues a socket() system call, but needs to wait until the SYN+ACK to know whether its peer supports MPTCP. Yet, it must already provide its token in the SYN.

Linux MPTCP currently instantiates its client-side meta-socket when the master-socket is created (just like the server-side). The drawback of this is that if after socket(), the application subsequently issues a listen(), we have built a useless meta-socket. The same happens if the peer SYN+ACK does not carry the MP_CAPABLE option. To avoid that, one may want to instantiate the meta-socket upon reception of an MP_CAPABLE option. But this implies that the token (sent in the SYN), must be stored in some temporary place or in the master subsocket until the meta-socket is built.

A.4. Forcing more processing in user context

The implementation architecture proposed in this draft uses the following queue configuration:

- Subflow level: out-of-order queue. Used for subflow-level reordering.

- Connection level: out-of-order queue. Used for connection-level reordering.

- Connection level: receive queue. Used for storing the ordered data until the application asks for it through a recvmsg() system call or similar.

In a previous version of Linux MPTCP, another queue configuration has been examined:

- Subflow level: out-of-order queue. Used for subflow-level reordering.

- Subflow level: receive queue. Used for storing the data until the application asks for it through a recvmsg() system call or similar.

o Connection level: out-of-order queue. Used for connection-level reordering.

In this alternate architecture, the connection-level data is lazily reordered as the application asks for it. The main goal for this was to ensure that as many CPU cycles as possible were spent in user context (See Section 3.1). VJ prequeues allow forcing user context processing when the application is waiting on a recv() system call. Otherwise the subflow-level reordering must be done in interrupt context. This remains true with MPTCP because the subflow-level implementation is left unmodified when possible. With MPTCP, the question is: "Where do we perform connection-level reordering ?". This alternate architecture answer is: "Do it _always_ in user context". This was the strength of that architecture. Technically, the task of each subflow was to reorder its own segments and put them in their own receive queue, until the application asks for data. When the application wants to eat more data, MPTCP searches all subflow-level receive queue for the next bytes to receive, and reorder them as appropriate by using its own reordering queue. As soon as the number of requested bytes are handed to the application buffer, the MPTCP reordering task finishes.

Unfortunately, there are two major drawbacks about doing it that way:

o The socket API supports the SO_RCVLOWAT option, which allows an application to ask not being woken up until n bytes have been received. Counting those bytes requires reordering at least n bytes at the connection level in interrupt context.

o The DATA_ACK [1] should report the latest byte received in order at the connection level. In this architecture, the best we can do is report the latest byte that has been copied to the application buffers, which would slightly change the DATA_ACK semantic described in section 3.3.2 of [1]. This change could confuse peers that try to derive information from the received DATA_ACK.

A.5. Buffering data on a per-subflow basis

In previous versions of Linux MPTCP, the configuration of the send queues was as shown in Figure 4.

```
                              Application
                                   |
                                   v
                            Packet Scheduler
                                 /  \
                                /    \
                               |      |
                               v      v
                             | * |  |   |
  Next segment to send (A) -> | * |  | * |
                             |---|  |---|  <- Separate send queue
  Sent, but not acked (B)  -> |_*_|  |_*_|
                               |      |
                               v      v
                              NIC    NIC
```

Figure 4: Send queue configuration

In contrast to the architecture presented in Section 3.5.2, there is no shared send queue. The Packet Scheduler is run each time data is produced by the application. Compared to Figure 4, the advantages and drawbacks are basically reversed. Here are the advantages:

- This architecture supports subflow-specific Maximum Segment Sizes, because the subflow is selected before the segment is built.

- The segments are stored in their final form in the subflow-specific send queues, and there is no need to run the Packet Scheduler at transmission time. The result is more fairness with other applications (because the Packet Scheduler runs in user context only), and faster data transmission when acknowledgements open the congestion window (because segments are buffered in their final form and no call to the Packet Scheduler is needed.

The drawback, which motivated the architecture change in Linux MPTCP is the complexity of the data allocation (hence the Packet Scheduler), and the computing cost involved. Given that there is no shared send buffer, the send buffer auto-tuning must be divided into its subflow contributions. This buffer size can be easily derived from Section 3.5.1. However, when scheduling in advance a full send buffer of data, we may be allocating a segment hundreds of milliseconds before it actually goes to the wire. The task of the Packet Scheduler is then complicated because it must _predict_ the path properties. If the prediction is incorrect, two subflows may try to put on the wire segments that are very distant in terms of DATA_SEQ numbers. This can eventually result in stalling some subflows, because the DATA_SEQ gap between two subflows exceeds the receive window announced by the receiver. The Packet Scheduler can relatively easily compute a correct allocation of segments if the path properties do not vary (just because it is easy to predict a constant value), but the implementation was very sensitive to variations in delay or bandwidth. The previous implementation of Linux MPTCP solved this allocation problem by verifying, upon each failed transmission attempt, if it was blocked by the receive window due to a gap in DATA_SEQ with other subflows. If this was the case, a full reallocation of segments was conducted. However, the cost of such a reallocation is very high, because it involves reconsidering the allocation of any single segment, and do this for all the subflows. Worse, this costly reallocation sometimes needed to happen in interrupt context, which removed one of the advantages of this architecture.

Yet, under the assumption that the subflow-specific queue size is small, the above drawback almost disappears. For this reason the abandoned design described here could be used to feed a future hybrid architecture, as explained in Section 3.5.2. For the sake of comparison with Table 3, we provide hereafter the action/table implemented by this architecture.

event	action
Segment acknowledged at subflow level	Remove references to it from the subflow-level queue
Segment acknowledged at connection level	No queue-related action.
Timeout (subflow-level)	Push the segment to the best subflow (according to the Packet Scheduler). In contrast with the solution of Section 3.5.2, there is no need for a connection-level retransmit queue, because there is no requirement to be available immediately for a subflow to accept new data.
Ready to put new data on the wire (normally triggered by an incoming ack)	Just send the next segment from the A portion of the subflow-specific send queue, if any. Note that the "IMPORTANT" note from Section 3.5.2 still applies with this architecture.

Table 5: (event,action) pairs implemented in a queue management based

on separate send queues

Appendix B. Ongoing discussions on implementation improvements

This appendix collects information on features that have been currently implemented nowhere, but can still be useful as hints for implementers to test. Feedback from implementers will help converging on those topics and propose solid guidelines for future versions of this memo.

B.1. Heuristics for subflow management

Some heuristic should determine when it would be beneficial to add a new subflow. Linux MPTCP has no such heuristic at the moment, but the topic has been discussed on the MPTCP mailing list, so this section summarizes the input from many individuals. MPTCP is not useful for very short flows, so three questions appear:

- How long is a "too short flow"
- How to predict that a flow will be short ?
- When to decide to add/remove subflows ?

To answer the third question, it has been proposed to use hints from the application. On the other hand the experience shows that socket options are quite often poorly or not used, which motivates the parallel use of a good default heuristic. This default heuristic may be influenced in particular by the particular set of options that are enabled for MPTCP (e.g. an administrator can decide that some security mechanisms for subflow initiation are not needed in his environment, and disable them, which would change the cost of establishing new subflows). The following elements have been proposed to feed the heuristic, none of them tested yet:

- Check the size of the write operations from the applications. Initiate a new subflow if the write size exceeds some threshold. This information can be taken only as a hint because applications could send big chunks of data split in many small writes. A particular case of checking the size of write operations is when the application uses the sendfile() system call. In that situation MPTCP can know very precisely how many bytes will be transferred.
- Check if the flow is network limited or application limited. Initiate a new subflow only if it is network limited.
- It may be useful to establish new subflows even for application-limited communications, to provide failure survivability. A way to do that would be to initiate a new subflow (if not done before

by another trigger) after some time has elapsed, regardless of whether the communication is network or application limited.

- o Wait until slow start is done before to establish a new subflow. Measurements with Linux MPTCP suggest that slow start could be a reasonable tool for determining when it is worth starting a new subflow (without increasing the overall completion time). More analysis is needed in that area, however. Also, this should be taken as a hint only if the slow start is actually progressing (otherwise a stalled subflow could prevent the establishment of another one, precisely when a new one would be useful).

- o Use information from the application-layer protocol. Some of them (e.g. HTTP) carry flow length information in their headers, which can be used to decide how many subflows are useful.

- o Allow the administrator to configure subflow policies on a per-port basis. The host stack could learn as well for what ports MPTCP turns out to be useful.

- o Check the underlying medium of each potential subflow. For example, if the initial subflow is initiated over 3G, and WiFi is available, it probably makes sense to immediately negotiate an additional subflow over WiFi.

It is not only useful to determine when to start new subflows, one should also sometimes decide to abandon some of its subflows. An MPTCP implementation should be able to determine when removing a subflow would increase the aggregate bandwidth. This can happen, for example, when the subflow has a significantly higher delay compared to other subflows, and the maximum buffer size allowed by the administrator has been reached (Linux MPTCP currently has no such heuristic yet).

Authors' Addresses

Sebastien Barre
Universite catholique de Louvain
Place Ste Barbe, 2
Louvain-la-Neuve 1348
BE

Email: sebastien.barre@uclouvain.be
URI: http://inl.info.ucl.ac.be/sbarre

Christoph Paasch
Universite catholique de Louvain
Place Ste Barbe, 2
Louvain-la-Neuve 1348
BE

Email: christoph.paasch@uclouvain.be
URI: http://inl.info.ucl.ac.be/cpaasch

Olivier Bonaventure
Universite catholique de Louvain
Place Ste Barbe, 2
Louvain-la-Neuve 1348
BE

URI: http://inl.info.ucl.ac.be/obo