Shaping the Linux kernel MPTCP implementation towards upstream acceptance

Doru-Cristian Gucea, Octavian Purdila

Open Source Technology Center, Intel
Bucharest, Romania
{doru.gucea, octavian.purdila}@intel.com

Abstract

This paper describes the challenges that we face in trying to integrate MultiPath TCP (MPTCP) in the official Linux kernel repository. This is a non-trivial task because the MPTCP implementation is invasive and mixes the MPTCP and TCP code and thus substantially increases the complexity of Linux's kernel TCP/IP implementation. In this paper we present several changes done to the MPTCP implementation that move the MPTCP code in a separate layer with the purpose of managing the complexity of the kernel implementation. To do so a series of problems must be solved: extract the MPTCP specific code from the TCP hot-path, create an MPTCP specific layer on top of TCP, pass MPTCP specific options to/from the TCP layer and have an efficient TCP fallback mechanism.

Keywords

MultiPath TCP, TCP kernel structures, sub-flow, meta socket, master socket.

Introduction

MultiPath TCP (MPTCP) is a transport layer protocol which takes advantage of today's Internet architecture where multiple paths exist between endpoints. The application uses a single TCP like socket with multiple sub-flows being started in kernel-space for the same connection. These sub-flows are implemented as normal TCP connections and are completely transparent to the application.

There are multiple benefits of MPTCP and probably the most common is increased throughput (up to 300% in the data center [1]). By using a coupled congestion control algorithm [2], MPTCP achieves bottleneck fairness while keeping the throughput improvements. It makes sure it does not consume more bandwidth on a single link then a typical TCP connection would and it also moves the traffic away from the congested links as much as possible.

Another typical multi-path use-case is the now increasingly present smart phone which has both a 3G and a Wi-Fi network interface. The only choice offered by TCP is to use a single path which flows either through the 3G or the Wi-Fi interface. If the 3G connection drops and we decide to use the Wi-Fi interface, there is a delay of about 350ms only for scanning for nearby access points, without considering the authentication and association delay. This is a serious problem for Voice over IP (VoIP) applications like Skype where delays over 250ms are unacceptable [3]. MPTCP solves this problem by starting two sub-flows, one for each interface. In this way, by the time the 3G connection drops, there is already a connection ready and the handover time which includes the scanning, authentication and association to the AP is eliminated. Moreover, MPTCP can be set to use a sub-flow as a backup path which is used only if there are no regular paths available. Going back to the previous use-case, there is the possibility to use the 3G interface as a backup one in order to minimize the Internet usage costs.

A generalization of this handover use-case is the situation where we want to use MPTCP to obtain the best possible network experience in the case of significant variations of the 3G and Wi-Fi signal quality. This is natural with MPTCP by using two sub-flows, one for 3G and one for Wi-Fi, because of the coupled congestion control.

MPTCP is helpful even when using just one physical Wi-Fi interface. The Linux kernel has the option of creating virtual network interfaces which are an abstract representation of a computer network interface that may or may not correspond directly to a physical network interface [4]. In the Multi Wi-Fi project [5] a client continuously scans for nearby access points and adds a new virtual network interface every time a new AP is discovered. This gives the possibility for MPTCP to open a new sub-flow for each AP it sees. Besides the fact that there is no need for handover, experiments also showed that the interactions between MPTCP and L2 layer successfully managed the hidden terminal situations [6], mainly because of the coupled congestion control algorithm from MPTCP.

MPTCP can also greatly enhance VM migration between two nodes as it allows the transport network connection to remain open across layer 3 domains. On the first device MPTCP will open a first sub-flow. After the VM is migrated to a second device, MPTCP will open a second sub-flow, with the new network configurations, all being transparent to the applications [7].

MPTCP can also be used to optimize the energy consumption on mobile devices. For example, transferring data using a poor Wi-Fi connection has two important characteristics: the interface stays on for a longer time and the screen also stays on while downloading if the user wants to view the content immediately. Using MPTCP, the 3G interface may be enabled for a short time to assist the first connection if the additional energy consumed by 3G will be less than the one used by the screen while waiting for the slower connection to complete the task [8].

Current MPTCP implementation

MultiPath TCP is an improvement of the current communication protocols, bringing forth a protocol that is more flexible and adaptive to the existing environment (e.g. available resources), provides an enhanced user experience and doing so is transparent to applications. Hence an implementation at the OS level makes sense and as such there is an MPTCP implementation for the Linux kernel that is maintained in an off-tree open-source repository by the academic community.

Our aim is to bring the off-tree Linux kernel MPTCP implementation in the official tree. We believe that doing so will accelerate MPTCP adoption and that will allow the above mentioned and probably other new and interesting use-cases to enrich the user's experience.

Design of the original MPTCP implementation

The design of the original implementation was to implement the MPTCP functionality directly at the TCP layer. For example, new code is added in the TCP layer that checks whether we are handling a simple TCP connection or an MPTCP one and then takes appropriate action.

The main MPTCP specific abstractions are: meta socket, master socket and sub-flow socket. The meta socket is the socket used for the management of the sub-flows and it is visible to user-space applications. It is implemented as a TCP socket although it does not have a direct mapping to a network level TCP connection.

The sub-flow sockets correspond to a TCP connection at the network level. They are also implemented as TCP sockets but in addition they have MPTCP specific data structures to deal with MPTCP options.

The master socket is a special type of sub-flow socket and corresponds to the first sub-flow opened by the MPTCP connection. In case the MPTCP connection fails at the handshake phase, the implementation falls back to TCP. In this case the master socket is used to keep an active connection with the peer and is also the socket that it is visible to user-space applications.

MPTCP employs a double level sequence number space: one for the meta and one for the sub-flow. The first one, used for the meta, starts at IDSN (Initial Data Sequence Number) which is calculated using a pair of keys exchanged during the handshake for the master-socket. The second one, used for the sub-flow, starts at ISSN (Initial Sub-flow Sequence Number) which is negotiated during the handshake for that specific sub-flow and are basic TCP sequence numbers. In order to map a segment between the meta level and sub-flow level, the Data Sequence Signal option (DSS) is used (Figure 1). This is an option inside the TCP header and it has four fields: Data Sequence Number (DSN) relative to IDSN, Sub-flow Sequence Number (SSN) relative to ISSN, Length and Checksum. DSS option maps each byte from kernel level to application level by saying that Length bytes starting at

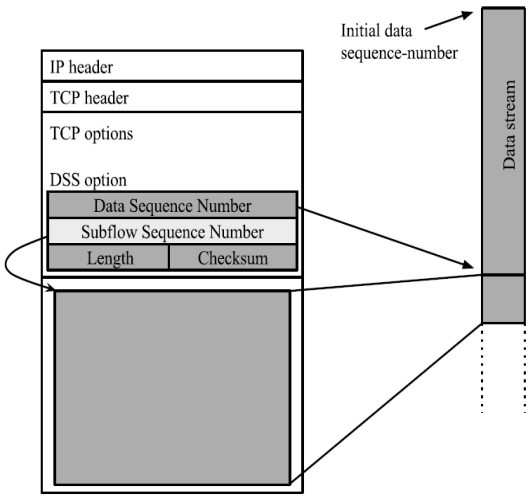

Figure 1: Data Sequence Signal Option [10]

SSN+ISSN from the sub-flow level maps to DSN+ISDN at meta-level.

The implementation is divided in several parts: MPTCP handshake, the handling of MPTCP options, the "routing" of MPTCP packets to meta, master or sub-flow sockets, receiving and sending data between the sub-flow sockets and meta socket, creating new sub-flows, scheduling output data to sub-flows (path manager), congestion control. Some of these parts are done directly at the TCP layer and are invasive (e.g. MPTCP handshake, handling MPTCP packets) while other parts are fairly well isolated from the TCP code (e.g. path manager).

MPTCP handshake

We believe that the existing handshake for MPTCP is complex mainly because the resources are allocated in a late stage of this process. The allocation path for MPTCP resources as done in the original implementation is illustrated in Figure 3, left side.

The MPTCP handshake on the client side starts with creating a normal TCP socket. Calling connect() in user-space will trigger the sending of a packet having the SYN and MP_CAPABLE (a specific MPTCP option) flags set, then it blocks waiting for a packet with ACK and MP_CAPABLE flags set. When the packet is received from the IP layer, the interrupt handler, tcp_v4_rcv(), is called and the packet is passed to the transport layer. If tcp_rcv_synsent_state_process() finds the MP_CAPABLE flag in the packet the path is redirected to MPTCP code for allocation of specific resources. The socket that we used until now for the communication with the server becomes the meta socket and the master socket, which is the first sub-flow of the connection, is created as a clone from the meta socket by calling sk_clone_lock(), the basic function from kernel which creates an identical copy of a socket. The connection with the server will be "switched" from the meta socket to the master socket, which means that from

now on the meta socket will be used only for management while the communication with the server will be done on the master socket. This is the reason for creating the master socket as clone, as the meta socket has connection state available (such as ACK and SEQ numbers) to allow communication with the peer. A common question about this mechanism is why not build the master socket around initial connection socket and create the management socket using the sock_create_kern() function thus rendering the clone operation unnecessary. The answer is that we can't change the socket the application talks with, so building the master socket around the initial connection socket would mean that the application talks only with the master socket.

A similar flow happens on the server side as well. The handshake starts with receiving a SYN and MP_CAPABLE flag and that's the time when a minimal representation of a socket, a request_sock is created. Upon the receive of the final ACK and MP_CAPABLE flags a full TCP socket is created as a clone of the TCP listener socket and this is set as the meta socket. The master socket is created as a clone after the meta socket (as is done on the client side), in order to preserve the connection state. The meta socket is returned to the application as a result of an accept() system call.

MPTCP receive path

The kernel code for TCP uses several queues for storing incoming TCP segments: backlog queue, prequeue queue, receive queue and out-of-order queue [9, 12]. The usage of these queues inside kernel is illustrated in Figure 2.

The receive queue contains processed segments, which means that all protocol headers are stripped and data is ready to be copied to the user application. The difference between the receive queue and out-of-order queue is that that in sequence packets are in the first one. The backlog and prequeue queues contain unprocessed segments as they are received from the IP layer. When a new packet is received from the IP layer in tcp_v4_rcv() and the socket is locked it will be put in the backlog queue. Otherwise, if the socket is free and we have an application waiting, the segments are put in the prequeue.

The original MPTCP implementation does not use the backlog queue and the prequeue queues for sub-flows. All the packets in tcp_v4_rcv() are put in the meta socket backlog or prequeue queues. This is achieved by adding a branch in the tcp_v4_rcv() that enqueues MPTCP packets in the queues of the meta socket. Thus, the backlog queue of the meta sockets contains all MPTCP packets from all sub-flows.

This backlog queue is processed at the time the packet arrived if the socket is not used by the application, or every time the application releases the socket. Processing of the backlog implies passing the SKB from the backlog to the receive queue of the socket. However, in the original MPTCP implementation, processing of the meta socket backlog doesn't imply that the SKBs are put directly in the receive queue of the meta socket. Instead, when the meta-

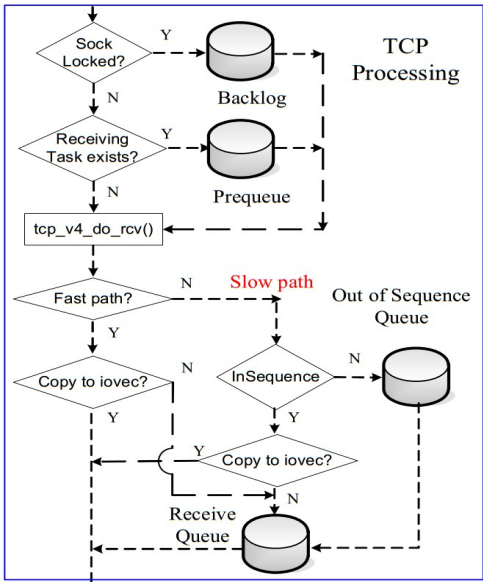

Figure 2: TCP queues [12]

socket backlog is processed, they are first transferred in receive queue of the sub-flow and later they are put in the meta socket receive queue. The packets can be copied to user space only after they had been put in the meta socket receive queue.

Regarding the kernel code, the processing of the backlog is triggered by release_sock(), when the socket is unlocked:
release_sock →...→sk->sk_backlog_rcv → tcp_v4_do_rcv
By passing the SKB and the socket which received that SKB as parameters to tcp_v4_do_rcv(), we process the packet and move it to the receive queue of the socket. MPTCP modifies the sk_backlog_rcv such that it passes as parameter the sub-flow socket which received that SKB, instead of the meta socket as it would have happened with no modifications. After the packets are enqueued in the receive queue of the sub-flow, the sk_data_ready callback is called. For TCP case this callback is used just for waking up the application and signal it that we have data in the receive queue. MPTCP also has to pass the SKBs from the sub-flow to meta receive queue. The application will wake up and tcp_recvmsg() will copy the data to the userspace buffers.

MPTCP send path

The send path is split into two separate actions: scheduling the data to be sent on one of the sub-flows and the actual sending operation. The first part is mostly done in the path manager which has a modular architecture that allows multiple implementations for MPTCP scheduling modules (e.g. fullmesh, ndiffports, etc.).

The second part deals with splitting the meta socket queued data into segments and adding the MPTCP specific information – Data Sequence Signal (DSS) to them. It also deals with retransmissions, and here things become

complex, as we might need to retransmit a segment on a different sub-flow which might require changing the DSS associated with the segment.
Since the DSS information must be stored in the TCP header as options and as the options are written off in the header long after the segment is created, the DSS must be stored somewhere in the SKB. One option would be to use the TCP SKB control block which is an opaque storage area usable by protocols, and even some drivers, to store private per-packet information [11]. The problem is that this space is limited, so the approach taken is to store it in the data section of the SKB, in the space reserved for the TCP header. This also has drawbacks as each time a TCP packet is copied so that is header is modified (via pskb_copy()) - for example when a packet is prepared to be sent on the wire and its the TCP header must be added, the DSS options must also be copied and that requires changes in the TCP stack.
The MPTCP send path is fairly well isolated from the rest of the TCP stack and it relies on hooks in the tcp_write_xmit function. This function is called when there is enough space in the send window and it sends the packet to the IP layer.

MPTCP implementation alternatives

As it can be seen from the previous section the original MPTCP implementation is probably the most flexible approach and can arguably offer the best performance but at the cost of increased complexity in the TCP stack.
An alternative would be to implement MPTCP in userspace. However, a complete userspace implementation would require implementing a duplicate TCP stack since MPTCP relies on a TCP stack being present. If an MPTCP userspace implementation is to be successful we believe that it needs to be able to use the existing kernel TCP stack. This would require changes to the TCP stack so that we can send and receive MPTCP options like MP_CAPABLE, MP_JOIN, DSS, ADD_ADDRESS, etc. These options could be passed as control messages (also known as ancilary data - data that is not part of the payload) to/from userspace using sendmsg() or recvmsg(). For the other system calls where we need to pass or receive options, such as connect() and accept(), we can probably rely on setsockopt() and getsockopt() to schedule options to be sent or to receive options. However, we also need to pass options during the connection handshake on the server side: get the options from the SYN request and pass options to the SYN,ACK response. With this, and probably other issues to be found in the details of the current MPTCP implementation, an equivalent of a FUSE [14] like solution for networking is probably required if we are to be successful in creating a userspace implementation for MPTCP.
Another alternative is to try to manage the complexity of an MPTCP implementation in the kernel by moving the MPTCP code in a separate layer that sits on top of the TCP layer. Create a new socket family that will implement the MPTCP logic and use plain TCP sockets for sub-flows. The sub-flow sockets can be manipulated with standard kernel level API such as tcp_read_sock(), sock_create_kern(), kernel_sendpage() / kernel_sendmsg() or tcp_write_xmit(). The MPTCP options can be passed through either the TCP SKB control block, through control messages or by passing another parameter to kernel_sendpage(). In this approach one problem is how to handle the fallback to TCP. The simple solution is to make the MPTCP layer act as a middle man but more lightweight solutions that switch the user visible socket from the MPTCP socket to the sub-flow socket at the file level may be possible.

MPTCP upstreaming process evolution

Our upstream work is split in two directions: a cleanup part where we separate the MPTCP code from the TCP code and a refactoring part where we build a new layer for MPTCP operations and bring this layer a level up above the TCP layer.

Isolate MPTCP code

With the purpose of separating the MPTCP and TCP layers in mind, we noticed that a lot of the MPTCP code added relied on a few conditions that differentiated the two layers: if the current socket was a meta socket, if the other end was MPTCP-capable or if the current socket is a master socket.
This means that, inside the TCP code, when one of the conditions above was true, the flow would be passed to an MPTCP function. We managed to remove these conditional statements by doing the following: for each condition that could pass the control from TCP to MPTCP functions, create a suite of functions pointers in the tcp_sock kernel data structure, each pointer corresponding to a place where that condition would be checked. Afterwards, whenever that flag/condition changed its value (true → false or false → true), change the function pointer in tcp_sock to its TCP or MPTCP function. More accurate, the tcp_sock_ops structure inside the tcp_sock structure contains a list of function pointers: select_window, init_buffer_space, set_rto, write_xmit, write_wakeup and others. This object-oriented design approach offers the possibility to initialize the function pointers once and remove the conditional statements. For example the call tp→ops→select_initial_window may call either mptcp_select_initial_window or tcp_select_initial_window based on the initial assignment to select_initial_window function pointer. This way, we removed all the conditional statements and replace them with calls to the function pointers placed in tcp_sock. One concern that we initially had was the overhead that might be introduced by the locking scheme: each time we called or changed one of the function pointers, the socket containing it must have had its lock held. This turned out to not be a problem due to the fact that the socket was already locked by the upper levels. This locking scheme is the reason why we added these

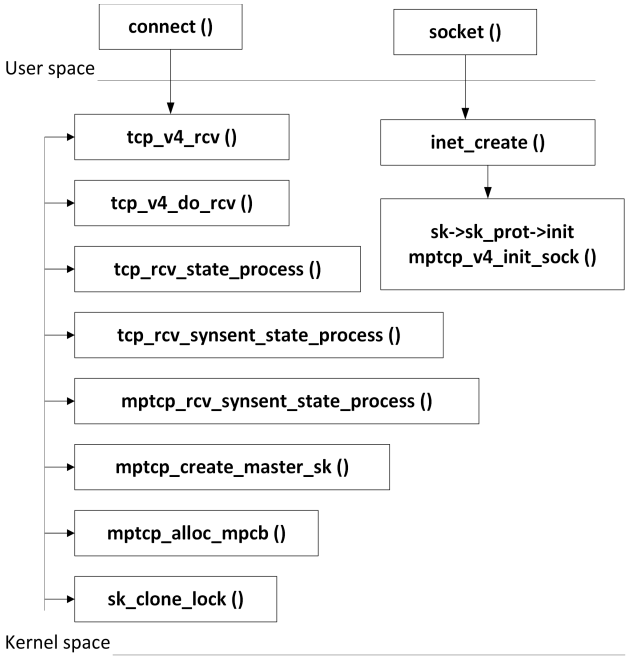

Figure 3: Path for the allocation of the meta and master socket. Before (left side) and after (right side) code refactoring.

function pointers to tcp_sock and not in a separate structure; if we would have grouped them in a separate structure, then the upper stack levels wouldn't have had already taken care of the locking and we would have added an overhead that could be avoided. As far as performance goes, we noticed a slightly, but steadily increased throughput after applying the above mentioned changes.

The next step in separating the layers was to create a new level of indirection by grouping the suite of functions pointers based on the type of socket: functions for a default TCP socket, functions for a sub-flow socket and, finally, functions for the meta socket. A normal TCP socket is created using the default group of functions. If this socket becomes MPTCP capable, the functions for sub-flow type will be assigned to it and all future code will call specific sub-flow functions. Also, a switch between these functions is done in case of a fallback.

In order to isolate the MPTCP code even more, we created MPTCP specific request sockets as well as connection request operations. This allowed us to reduce request_sock to its original size. It also allowed us to remove the MPTCP code from the TCP stack part that deals with accepting new connections (tcp_v4_conn_request() and tcp_v6_conn_request()) since we can now parse the MPTCP options and do the MPTCP checks in the MPTCP specific connection request operations after which we can call the TCP specific connection request operations. As a generalization of these changes we also removed code duplication between IPv4 and IPv6 connection request functions. This independent changes were positively

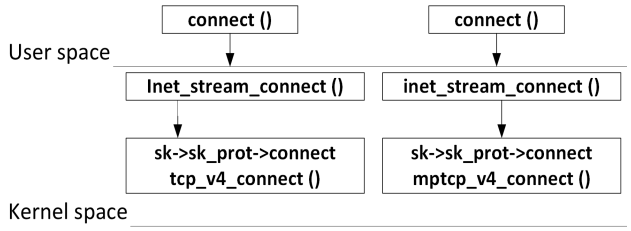

Figure 4: Connect path. Before (left side) and after (right side) code refactoring.

received by the Linux kernel networking community and they were merged upstream.

Another change that we did was to simplify the DSS handling: save part of DSS options (and compute the rest dynamically when writing the TCP header) in the TCP SKB control block when queuing a packet to a sub-flow. This allowed us to remove the code that explicitly copies the DSS block each time an SKB is copied. In order to fit in the control block we had to use a double union: once with the inet_skb_parm / inet6_skb_parm union – these two are only used on the receive path, and once with MPTCP 's path_mask - the mask is not used after queuing the packet to the sub-flow.

Dedicated MPTCP layer

We started implementation of a new protocol, IPPROTO_MPTCP that represent the MPTCP protocol. This way, we can implement MPTCP specific operations cleanly and separately from the TCP layer. Following this design, meta socket is created as soon as the socket is created. This is a special IPPROTO_MPTCP socket. The meta socket is visible to the application level and the MPTCP code will create sub-flows sockets that will simply be IPPROTO_TCP sockets, just like in a regular TCP connection as described in RFC 6824. However, these sub-flows are different from a regular TCP sockets by the extra MPTCP options that it encloses. These are the only changes to the TCP sockets that we cannot separate from the TCP layer. The rest of the MPTCP implementation is moved an upper level that will communicate with the TCP layer. In this way we have a clear separation between the TCP operations represented by the functions from tcp_prot structure and MPTCP operations from mptcp_prot structure. We have total freedom to write our own operations for MPTCP without worrying of mixing code.

With this approach, the application needs to specifically open an MPTCP connection. However, it should be possible to redirect TCP socket creation calls to MPTCP socket creation calls in inet_create(), based on, for example, sysfs settings. It also means that fallback to TCP must be handled inside the MPTCP layer itself by forwarding calls from the MPTCP layer to the TCP layer.

The IPPROTO_* constants generally match what is put on the wire inside the IP header. In the first phase we chose number 7 for the new protocol and that it did not

match well with this convention, since this number was already used by another IANA protocol, CBN. Based on community feedback we later switched to define it using a bit mask approach, IPPROTO_TCP | TCPEXT_MPTCP.
Regarding the kernel data structures, the new layer operations are implemented in the mptcp_prot structure and are specific to the meta socket. The sk_prot field from the sock structure of the meta socket points to this data structure. Using this approach we preserve the object-oriented design of the Linux kernel.

Early allocation of MPTCP resources

Starting with an MPTCP connection by default and allocating specific resources for it from the beginning gives us the possibility to make a clear separation between the master socket and the meta socket and remove the need for cloning the meta socket. Using the new approach we can create the MPTCP data structures by the time the socket() function is called in user space as it is illustrated in figure 3, right side. After the application calls socket(), requesting for IPRPROTO_MPTCP the control is passed to inet_create() in kernel space which calls sk → sk_prot → init(). Depending on the protocol used, the sk_prot field from socket points to a specific set of functions. mptcp_v4_init_sock() is an MPTCP specific function where we allocate the master socket, an IPPROTO_TCP socket, and the meta socket, an IPPROTO_MPTCP socket. These sockets are created using the sock_create_kern() function from kernel without the need to replicate an active connection because we don't have one yet.

Simplified connect path

We also modified the connect path as can be seen in figure 4, right side. The application calls connect() in user-space using the meta socket. The difference now is that the sk_prot field is initialized to the mptcp_prot structure and instead of calling tcp_v4_connect() we call mptcp_v4_connect which is MPTCP specific. We can access the master socket because the function receives the meta socket as a parameter and this management socket encapsulates all the sub-flows. After extracting the master socket we can use it in order to establish a connection with the server using a regular TCP function: tcp_v4_connect().

New receive path

When we started the work for the receive path we noticed that there is some MPTCP code in the tcp_v4_rcv() related to the handling of queues. Of course that we would like to remove it, but that will cause MPTCP packets to go in the prequeue and backlog of the sub-flows. We would then need a way to move the data from the sub-flow sockets to the meta socket. Our work for the receive path is still in progress but the idea is illustrated in Listing 1. The pseudo code has two main parts: mptcp_recvmsg() which is a pure MPTCP function specific to the meta socket, and the sk_data_ready callback, a modified version for waking up the application. When the application tries to read data, a call to mptcp_recvmsg() will be triggered and the application will wait for data using a wait-queue. It will be woken up by sk_data_ready() when there is data in the receive queue of the sub-flow. We also use a 32 bit mask for marking the sub-flow which has ready data. Every bit from this mask identifies one of the 32 sub-flows (this is the maximum number of sub-flows in the current implementation).

```
mptcp_recvmsg (on meta socket)
    wait_event(wq, ready_sub-flows)
    for all ready sub-flows
        lock sock(sub-flow)
        tcp_read_sock(sub-flow) – recv actor
            clone SKB and add to meta socket
        clear sub-flow ready bit
        release_sock(sub-flow)
    tcp_recvmsg (meta socket, O_NONBLOCK)
```

```
sk_data_ready (on sub-flow sockets)
    scan the receive queue and update DSS mapping
    mark sub-flow ready and wake-up meta socket
```

Listing 1: Redesigned receive path

The processing of the backlog queue for the sub-flows will be done in the release_sock() function as is done in normal TCP. Note that the prequeue is not going to be used for MPTCP as sub-flows are not being directly visible to userspace.
In order to pass the data from the sub-flow level to the meta level we are using tcp_read_sock(), a standard kernel API – or at least used by other important users such as NFS, CIFS, etc. tcp_read_sock() iterates through the sub-flow receive queue, extracts an SKB, calls a helper function (recv_actor) on that SKB then unlinks and frees the SKB. The helper function is received as a parameter for tcp_read_sock and this allowed us to fill it with code which enqueues a clone of the SKB from the sub-flow receive queue to meta receive queue. Unlinking the packet is done after the call to the helper function so our code is not allowed to remove it. That's the reason for using a clone in the helper function: an SKB can't be in more than one queue at a time.
One issue that we had in integrating tcp_read_sock() was that not all the SKBs from the receive queue had the DSS mapping so we couldn't enqueue the packet in the meta receive queue. A DSS option for a given packet may be present in subsequent packets due to middle-boxes that coalesce packets and drop the MPTCP options [13]. Because of that we have to scan the receive queue and detect the DSS mapping before unblocking the meta socket's mptcp_recvmsg(). We can do that in the mptcp_sk_data_ready callback function.
For passing data to user space we use tcp_recvmsg() in order to avoid duplicating the code for copying data to user space.

Conclusions and future work

The work that we've done until know demonstrates that upstreaming a network protocol is not an easy task. The work that prepares MPTCP for upstreaming started with separating the TCP layer from the MPTCP layer, continued with the creation of a new abstraction layer above TCP where we can define our own MPTCP specific operations and reached a point where we redesign the receive path. We managed to make the code clear by removing conditional statements using function pointers, rework the DSS handling to remove extract MPTCP code from the TCP layer, rework the connection hand-shake by using MPTCP specific request and connect operations that allowed us to isolate MPTCP code and remove duplicated TCP code from MPTCP code, and exposed an MPTCP specific API to the application that allows us to make a clear distinction between TCP operations and MPTCP operations.

As future work, with the changes that expose the meta sockets separatly from the TCP layer, we can rework the send path and avoid the hook in the TCP layer. Instead of doing the scheduling in tcp_write_xmit() at the TCP level we can do it in sendmsg() at the meta socket level.

We are also looking to see if we can improve the performance in the case MPTCP fallbacks to TCP by eliminating completely the MPTCP layer in this case. This could be done by switching the user visible socket from the MPTCP socket to the TCP socket at the struct socket level. One side effect of the switch is that we might have the switching done while a process is waiting for an event on the old socket (e.g. create the socket, do a non-blocking connect and then do a recvmsg). We can avoid this issue by waking up the socket and restarting the system call. This should be feasible to implement since all socket blocking operations seems to use the sk_wq waiting queue which makes it fairly simple to wake-up the socket for a wide range of blocking conditions.

Acknowledgements

The authors would like to acknowledge Andrei Maruseac, Mihai Andrei, Cristina Ciocan, Irina Tirdea, George Milescu and Gregory Detal who have contributed with patches and ideas to the work described in this paper. We especially like to thank Christoph Paasch for the invaluable feedback and the relentless patch reviews.

References

1. Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, Mark Handley, *Improving Datacenter Performance and Robustness with Multipath TCP,* in proceedings of SIGCOMM 2011

2. Costin Raiciu, M. Handly, D. Wischik, *RFC6356 Coupled Congestion Control for Multipath Transport Protocols*

3. Ishwar Ramani, Stefan Savage, SyncScan: Practical Fast Handoff for 802.11 Infrastructure Networks, INFOCOM05

4. *Virtual Network interface*, Wikipedia, accesed on January 20 2015, http://en.wikipedia.org/wiki/Virtual_network_interface

5. Andrei Croitoru, Dragos Niculescu and Costin Raiciu, *Towards Wifi Mobility without Fast Handover,* Usenix NSDI 2015

6. Mauro Borgo, Andrea Zanella, Paola Bisaglia, Simone Merlin: Analysis of the hidden terminal effect in multi-rate IEEE 802.11b networks

7. Fikirte Abebe Teka, Seamless Live Virtual Machine Migration for Cloudlet Users with Multipath TCP, Master Thesis

8. Costin Raiciu, Dragos Niculescu, Marcelo Bagnulo, Mark Handley, Opportunistic Mobility with Multipath TCP

9. Sammer Seth, M. Ajaykumar Venkatesulu, TCP/IP Arhitecture, Design and Implementation in Linux.

10. Christoph Paasch, Phd. Thesis: Improving Multipath TCP

11. How skbs work: http://vger.kernel.org/~davem/skb.html, accesed on January 20 2015

12. W. Wu, M. Crawford, The performance analysis of Linux networking – packet receiving

13. Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, Mark Handley: How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP

14. FUSE Filesystem in Userspace: http://fuse.sourceforge.net/, accesed on January 20 2015

Authors Biographies

Doru-Cristian Gucea graduated University Politehnica of Bucharest and is working at Intel as a Software Engineer in the Open Source Technology Center group. Octavian Purdila is a Software Architect at Intel in the Open Source Technology Center group. He is also teaching OS internals at the University Politehnica of Bucharest.