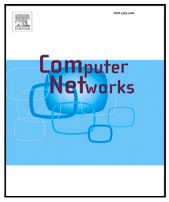
Computer Networks 252 (2024) 110671

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet



Software article

Multipath TCP implementation under FreeBSD-13 for pluggable machine learning models

Shiva Raj Pokhrel [a,*], Jonathan Kua [a], Brenton Fleming [a], Sebnem Ozer [b,1], Jeff Howe [c], Anwar Walid [d]

[a] School of Information Technology, Deakin University, Geelong, VIC 3220, Australia
[b] Charter, United States
[c] Comcast Corporation, United States
[d] Amazon Science, United States

ARTICLE INFO

Keywords:
MPTCP
FreeBSD

ABSTRACT

Multipath TCP (MPTCP) has been implemented and evaluated using the modular approach known as ModCC in FreeBSD. This research builds a foundation for developing an experimental platform and implementing MPTCP protocol stacks in FreeBSD-13 to plug machine learning (ML) models. Several independent and interoperable implementations are essential for the Internet Engineering Task Force (IETF) recognition of emerging ML-based MPTCP algorithms. We implement MPTCP under FreeBSD-13 and make them publicly available as dynamically pluggable user and kernel modules for testing and experimentation by the wider network and the Internet community. In particular, we redesign MPTCP over FreeBSD and implement enhanced ModCC in V13.1. In addition, we develop and implement a new kernel module CC-DRL to handle two system calls: (i) `drl_get_buffer` to marshal data from the kernel to ML models in the user space and (ii) `drl_update_cwnd` to marshal data from the ML models to the kernel space. CC-DRL operates in parallel with ModCC. We present new insights and concise descriptions of MPTCP implementation with modular multipath scheduling and highlight challenges to assist other parallel initiatives to develop compatible MPTCP stacks. The code and implementation are open source to FreeBSD 13.1.

1. Introduction

TCP extensions for multipath operation with multiple IP addresses, the so-called multipath TCP (MPTCP), is standardized by the Internet Engineering Task Force (IETF) in RFC 8684 [1]. Many novel multipath TCP (MPTCP) algorithms have been developed, simulated, and implemented in recent years.[2] Each offers a unique method to determine how congested the paths in the network are and what balancing act is needed to alleviate the problem [2]. Some were reported to have been put into production before receiving a thorough objective review from the Internet Engineering Task Force (IETF) and the Internet Research Task Force (IRTF).[3]

The Internet Congestion Control Group (*iccrg*) is an active IRTF group that is tasked with investigating and evaluating novel schemes for MPTCP. For an IETF's MPTCP standard to advance, it must be implemented effectively by many different groups and assessed in the wild, in different operating systems, preferably using the exact published specification. The IRTF's *iccrg* is tasked by the IETF to assess the dynamic behavior of new TCP and MPTCP developments [3,4].

Of particular relevance to this paper is the MPTCP implementation in FreeBSD-11 [5], which uses a modular congestion control framework known as ModCC. This article builds on [5] to develop a renewed foundation for (i) developing an experimental platform of MPTCP and (ii) implementing the Machine Learning (ML)-based MPTCP (ML-MPTCP) protocol stack in FreeBSD-13. Indeed, having an independent and interoperable implementation is a significant foundation for the IETF recognition of new ML-MPTCP algorithms. Therefore, we implement MPTCP under FreeBSD-13 and make them publicly available as dynamically pluggable user and kernel modules for testing and experimentation by the more comprehensive network and the Internet community.

* Corresponding author.
E-mail address: shiva.pokhrel@deakin.edu.au (S.R. Pokhrel).
[1] Formerly with Comcast Corporation.
[2] https://github.com/multipath-tcp/mptcp.
[3] We can see who has been using MPTCP, https://www.multipath-tcp.org/.

https://doi.org/10.1016/j.comnet.2024.110671
Received 8 August 2023; Received in revised form 4 December 2023; Accepted 21 July 2024
Available online 23 July 2024

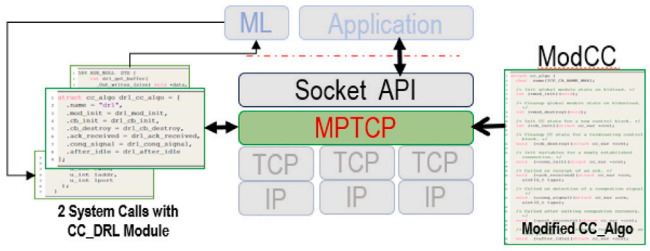


Fig. 1. Proposed novel architecture of multipath TCP implementation under FreeBSD-13 for pluggable ML models. We redesign the MPTCP implementation in FreeBSD and enhance ModCC in the V13.1 TCP stack. We design and develop a new CC-DRL kernel module to handle two system calls, (i) `drl_get_buffer` to marshal data from the kernel to ML models in the user space and (ii) `drl_update_cwnd` to marshal data from the ML models to the kernel space, which operate in parallel with ModCC.

We redesign the MPTCP implementation in FreeBSD and enhance ModCC in the V13.1 TCP stack. Fig. 1 illustrates the details of the proposed new architecture of multipath TCP stack under FreeBSD-13 for pluggable ML models. Furthermore, motivated by the new MPTCP theories based on deep reinforcement learning (DRL) [2,6–9], we designed and developed a new kernel module CC-DRL to handle two system calls, namely, (i) `drl_get_buffer` to gather data from the kernel to ML models in the user space and (ii) `drl_update_cwnd` to gather data from the ML models to the kernel space, which operate in parallel with ModCC. We develop a novel modular multipath scheduling mechanism and present new insights and concise descriptions of challenges in FreeBSD kernel to help other parallel initiatives in developing compatible MPTCP stacks under various operating systems.

2. State of the art

Several ML ideas have been proposed and applied to enable intelligence under MPTCP — authors in [10] employed policy gradient for improving congestion control strategies; [11] used DRL to enhance MPTCP scheduling under heterogeneous links; [12] implemented virtual reality with prioritization using NS-3 simulation.

Five notable MPTCP designs [2,6–9] are using *deep reinforcement learning* (DRL); [6,9] developed their implementation and evaluation are based on NS-3; [7] build the implementation on top of MPTCP v0.92 of the Linux kernel; [2,13] developed on top of the Linux implementation along the lines of [7]. All of these MPTCP designs [2,6,7,9] focused on congestion control and scheduling and developed their implementation in Linux or NS-3. Authors in [8,9] introduced Deep Q Network (DQN) framework to improve MPTCP congestion control [9] and packet scheduling [8] implemented over NS-3 by simulating asymmetric links.

In fact, none of the notable ML-based MPTCP designs mentioned above [2,6–9] developed and evaluated their design on FreeBSD. It should be noted that the implementation of MPTCP under FreeBSD has been an imminent challenge, with no updates available after [5]. There are no known works on modular multipath scheduling in FreeBSD. However, it is essential to have an independent and interoperable FreeBSD MPTCP implementation and modular multipath scheduling to understand and recognize the performance of new ML-based MPTCP over FreeBSD.

2.1. FreeBSD TCP implementations

FreeBSD features a modular congestion control framework, ModCC, initially developed by CAIA/Swinburne University [14,15]. The ModCC framework aimed to advance TCP congestion control research by allowing congestion control algorithms to be implemented as dynamically loadable kernel modules, significantly reducing the time and effort required to develop and evaluate new congestion control implementations. Congestion control in FreeBSD is managed by regulating the number of segments allowed in transit at a given time, referred to as the congestion window. The key attributes of ModCC include: (i) congestion control routines, established as a pointer to a set of functions within the `cc_algo` structure, (ii) private data transport and associated statistics per connection through the `cc_data` pointer, (iii) a structure `cc_var` linking the TCP control block and the congestion control algorithm, and (iv) an extension of the TCP control block with pointers facilitating congestion control algorithms on a per-connection basis. The works by Zou et al. [16] and Kumar et al. [4] utilized the TCP stack, and [17] used MPTCP stack implementations in FreeBSD.

The RACK (Recent Acknowledgment) TCP stack [18] is one of the pluggable TCP stacks that FreeBSD supports since version 12. Instead of just switching out the congestion control algorithm, FreeBSD now allows for the dynamic loading and use of a completely other TCP stack, the RACK stack (defined in RFC 8985). Once the RACK stack has been loaded, it can take over for the normal FreeBSD TCP stack in processing TCP traffic, or it can remain inactive. Of particular relevance to this work is the RACK stack (RFC 8985) and the implementation of MPTCP by CAIA [19] under FreeBSD-11, which was released as an experimental patch that added multipath support in the FreeBSD kernel. However, the FreeBSD kernel had evolved considerably between version 11 (V11), released in October 2016, and version 13.1 (V13.1), released in May 2022. This included significant changes in the implementation of the TCP stack, and, as a result, the FreeBSD-11 patch [19] could not be stacked under V13.1. Most of our design aspects and implementation follow along the lines of that of CAIA MPTCP implementation (detailed in [5]), therefore, we include the essential enhancements and important details in this paper.

Our project funded by the COMCAST innovation program has created a FreeBSD MPTCP stack to facilitate advanced research activities in MPTCP. FreeBSD MPTCP stack is an open-source software distribution along with FreeBSD egineering release with all essential supporting materials and details including the novel multipath scheduling framework. Our focus is on designing an extensible system to streamline experimentation with MPTCP congestion control, scheduling, and path management schemes.

2.2. Multipath congestion control

The congestion windows for MPTCP connections vary depending on the subflows involved. Coupling the congestion windows used on each subflow is essential to mandate most traffic onto uncongested networks and promote fairness at network bottlenecks and resource pooling. In MPTCP (RFC6356 and RFC 8684) [1], we see one possible strategy for doing so; while it falls short of ideal resource sharing, it is "safe" because it can be easily implemented on the Internet as it stands right now. For instance, the congestion window (w_i) is updated for each received acknowledgment, where the window increases as $w_{i+1} = w_i + \Delta(w_i)$,

$$\Delta(w_i) = \min\{\delta, \frac{1}{w_i}\} \quad , \quad \delta = \frac{\max_i(w_i/RTT_i^2)}{(\sum_i \frac{w_i}{RTT_i})^2} \tag{1}$$

is what the MPTCP *Linked Increase Algorithm* (*cf.* RFC 6356 and RFC 8684) employs during the congestion avoidance.

We can see that, if $\delta \geq \frac{1}{w_i}$, then $\Delta(w_i) = \frac{1}{w_i}$ in (1), that is, the update of w_i evolves similarly to that of a TCP, but, with $\min(\cdot,\cdot)$ operation, the effective rate of increase of the congestion window (of any path) of an MPTCP subflow is slower than that of a TCP under identical conditions. As a result, it can peacefully coexist with single-path TCP at common bottlenecks without negatively impacting the performance of either protocol. Many congestion controllers are expected to be deployed for MPTCP, each with its own set of goals related to quality of service, dependability, robustness, resource sharing, fairness, and stability.

Modern ML-based MPTCP's design, architecture, and implementation shall always aim to ensure that congestion control functions have all the data they need to make sound decisions, such as which packets are to be dropped and when they shall happen for each subflow. To this end, perhaps the best known ML-based MPTCP to date is the hybrid MPTCP proposed in [2] and implemented in Linux. We will have a quick look at the congestion control details of hybrid MPTCP [2] to understand the principles and architecture of ML-based MPTCP in detail. Their design of Hybrid MPTCP decomposes MPTCP congestion control into two layers. The *first layer* is extended from the MPTCP operation shown in (1) where for every packet loss indication over subflow i, it performs a multiplicative decrease and updates the congestion window w_i' as $w_i' \leftarrow w_i'/2$.

In contrast to the multiplicative decrease for a loss, it uses a continuous increase in the congestion window w_i' for every acknowledgment that has been successfully received using the operation in (1). Based on the DRL agent, the outer loop, which runs as a daemon process, is the *second layer*. It is responsible for guaranteeing the delay while harnessing network variations through its constant monitoring of the network and traffic dynamics, its computation of the effective congestion window, $\hat{w}_i$, and its periodic enforcement.

2.3. Existing MPTCP implementations

2.3.1. NS-2/NS-3 implementation

Simulation trials, frequently employing NS-2 [20] or NS-3 [21–24], have dominated the published literature on MPTCP behavior over the last few years [6,9]. We can test fundamental multipath congestion control concepts in a NS-based multipath network environment without having to learn and understand the ins and outs of the actual operating system [9]. As an additional benefit, simulations can potentially be used to learn minute details in a pragmatic approach to the dynamic behavior of a CC algorithm — simply by simulating (by approximating) network and operating system environments [21–24]. But it is known that such simplifications and approximations are often major drawbacks of NS and without testing new MPTCP algorithms in the production settings, and we cannot know how they will perform in practice.

2.3.2. Linux implementation

The MPTCP implementation in the Linux kernel is hosted on its site by the IP Networking Lab[4] so that it may be used by users, researchers and developers. Sébastien Barré and Gregory Detal, two major figures in the development of the Linux kernel's MPTCP stack implementation, have founded a firm to commercialize it.

In general, we get in touch with Gregory Detal (Tessares) and Sébastien Barré (Praos) when we require commercial grade support, new features, or backport to older kernel releases when building the application on top of Linux MPTCP stack.

Sébastien Barré is credited with creating the first MPTCP implementation and architecture for Linux — based on the shim6 implementation; they began this project in 2009 and worked on the MPTCP implementation. Many more people have since joined the effort, including Christoph Paasch, Fabien Duchêne, and Gregory Detal being the current active developers. In summary, research into new design aspects such as ML-based MPTCP can currently only be undertaken using the Linux-based implementation. Several new ML-based Linux MPTCP implementations [2,7,13] show one way to design a modular system that can grow as needed. While this concept can be generalized, the intrinsic differences between the Linux and FreeBSD kernels make it challenging to make a direct translation. For example, it is possible to poll in Linux so that minimal information is lost, although doing so may have a considerable impact on system speed since there is a trade-off between measurement granularity and polling's impact on the performance.

[4] https://inl.info.ucl.ac.be/.

Table 1
Design and implementation of MPTCP architecture: Modified or added files in FreeBSD V13.1.

Modified files	Added files
sys/conf/files	sys/amd64/conf/MPTCP
sys/conf/kern.pre.mk	sys/netinet/mptcp.h
sys/kern/uipc_socket.c	sys/netinet/mptcp_dtrace_declare.h
sys/netinet/in_pcb.c	sys/netinet/mptcp_dtrace_define.h
sys/netinet/in_pcb.h	sys/netinet/mptcp_handshake.c
sys/netinet/in_proto.c	sys/netinet/mptcp_pcb.c
sys/netinet/tcp.h	sys/netinet/mptcp_pcb.h
sys/netinet/tcp_input.c	sys/netinet/mptcp_reass.c
sys/netinet/tcp_output.c	sys/netinet/mptcp_sched.h
sys/netinet/tcp_reass.c	sys/netinet/mptcp_subr.c
sys/netinet/tcp_subr.c	sys/netinet/mptcp_timer.c
sys/netinet/tcp_syncache.c	sys/netinet/mptcp_timer.h
sys/netinet/tcp_syncache.h	sys/netinet/mptcp_types.h
sys/netinet/tcp_timer.c	sys/netinet/mptcp_usrreq.c
sys/netinet/tcp_timewait.c	sys/netinet/mptcp_var.h
sys/netinet/tcp_usrreq.c	sys/netinet/tcp_usrreq.h
sys/netinet/tcp_var.h	
sys/sys/socketvar.h	

3. Novelty and contributions

3.1. MPTCP redesign and implementation in FreeBSD V13.1

With the considerable evolution of FreeBSD from v.11 to 13.1 (V13.1) and the major changes in its implementation of the TCP stack, we find that the patch [19] cannot be applied directly under V13.1. Instead, the changes designed for FreeBSD-11 are to be redesigned and implemented with the new TCP stack of v.13. Therefore, we perform a greenfield integration and implementation of MPTCP in the V13.1 kernel. First, we analyze the change within the context of V11, identifying the respective reverse engineering needs in the V13.1 kernel, and then accomplish the required design and implementation of MPTCP in V13.1 with minimal possible modifications, where required.

Our understanding of the details of the FreeBSD-13.1 engineering release source code and reverse engineering of the MPTCP patch V11 (with important information from [19]) allows us to redesign and develop a framework with a detailed codebase V13.1, generating a new implementation of the MPTCP stack. This redesign of MPTCP in V13.1 is a major departure from V11, and its implementation leverages valuable additions to the V13.1 TCP stack. The code implementation of V13.1 MPTCP is mostly done by the author *Brenton Fleming*.

To enable MPTCP under V13.1, we have added 16 new files and modified 18 existing files, as detailed in Table 1. Several cautious redesigns of the `ModCC` components and their modifications were required to produce a working implementation of MPTCP in V13.1. For tractability and comparison, significant changes to V13.1 (compared to V11) for both new and modified kernel files shown in Table 1 and associated justifications are provided in Section 5.

3.2. System calls in FreeBSD-13.1

We implemented a novel kernel CC module referred to as, `CC-DRL`,[5] to deploy the `ModCC` interface, and a user-space module to process packet transport statistics. To this end, two new system calls are added to the kernel to support the marshaling of relevant data transport statistics between the user and the kernel space. These system calls are implemented as functions in the `CC-DRL` kernel module; all modified or added kernel file names are listed in Table 2.

[5] The MPTCP congestion control using deep reinforcement learning (DRL) is quite well known in the literature, hence, we named this module as `CC-DRL` for simplicity.

Table 2
Plugging ML with system calls: Kernel files modified or added for DRL-CC.

Files	Status
sys/compat/freebsd32/syscalls.master	Modified
sys/conf/files	Modified
sys/kern/init_sysent.c	Modified
sys/kern/syscalls.master	Modified
sys/kern/systrace_args.c	Modified
sys/netinet/cc/cc_drl.h	Added
sys/netinet/cc/cc_drl.c	Added
sys/sys/syscall.h	Modified
sys/sys/syscall.mk	Modified
sys/sys/sysproto.h	Modified

The CC-DRL kernel module handles kernel side operations inline with the ModCC structure. As processing and decision making are being transferred to the ML processes in the user space, the kernel module only needed to implement a reduced subset of ModCC functions as shown in the listing 1 are being transferred to the ML modules in the user space. See Section 6 for details.

```
struct cc_algo drl_cc_algo = {
  .name = "drl",
  .mod_init = drl_mod_init,
  .cb_init = drl_cb_init,
  .cb_destroy = drl_cb_destroy,
  .ack_received = drl_ack_received,
  .cong_signal = drl_cong_signal,
  .after_idle = drl_after_idle
};
```

Listing 1. CC-DRL Implementation

The drl_get_buffer system call is added as the primary mechanism to transfer data between the kernel and user space. Its function copies a buffer containing size elements of type pkt to the calling process. For CC-DRL the behavior of drl_ack_received and drl_cong_signal is essentially the same as the logic blocks typically present in these functions are to be handled by the user process.

The drl_update_cwnd system call, shown in listing 2 facilities updating a congestion window of connections in the user space. The connection is located using the laddr and lport parameters. The congestion window is set to the value provided by the cwnd parameter.

```
583 AUE_NULL  STD {
    int drl_update_cwnd(
      u_int cwnd,
      u_int laddr,
      u_int lport
    );
  }
```

Listing 2. Definition of drl_update_cwnd system call

3.3. Important design and implementation considerations

We have identified and resolved three main design and implementation challenges which are discussed below.

3.3.1. Minimizing system call overheads

A significant challenge for pluggable MPTCP proposed is the additional latency introduced due to system call overheads for marshaling network statistics between FreeBSD user and kernel space and the incurred processing delays due to the complex internal computing involved in the training and prediction of ML. However, the proposed pluggable kernel module CC-DRL and its implementation facilitate an asynchronously interacting interface between the MPTCP stack at the kernel level and the ML in the user space (recall Fig. 1).

When developing such an asynchronous coupling, we always seek to ensure that packet processing in the standard TCP stack is seamless and never gets adversely affected. One main limitation of the proposed approach is the slight temporal lag of the knowledge base between the user and the kernel space in network congestion. The study towards a more tightly coupled design where ML models and MPTCP are fully synchronized requires a different approach and will be investigated in the future. Nevertheless, due to communication bottlenecks between network nodes, the impacts of system calls in such a cross-layer design where the ML model sits in the application layer and interacts with the implementation of the Linux MPTCP stack implementation [2,7], appears to be negligible.

3.3.2. Handling user-kernel events for congestion window per subflow

Another major challenge is handling events per subflow, as the congestion window is per subflow, and we need to develop an ability to update congestion control. To this end, we select the combination of local address and port to uniquely identify a subflow. The corresponding fields (laddr and lport) are added to the pkt structure and as inputs to drl_update_cwnd(). When drl_update_cwnd() is called, ccv_list (see 6.3) is traversed to find the corresponding connection and access its structure cc_data.

3.3.3. Understanding thread safety and ameliorating kernel instability

Thread safety was another major concern in understanding kernel stability in early designs of the MPTCP architecture. Areas of concern that we have addressed in the redesign include (i) kernel-to-user marshaling, and (ii) user-to-kernel marshaling.

For example, we have made it plausible for the kernel thread to attempt to queue a new node via calls to drl_ack_received() or drl_cong_signal() while being drained by drl_get_buffer().

The initial design in [5] attempts to pass the cc_data pointer, which could then be used to access the memory directly. This approach may not be safe and impacts kernel stability as the subflow terminates before all acknowledgments have been handled.

4. Implementation Details of ModCC in FreeBSD

FreeBSD features a modular congestion control framework ModCC, originally developed by CAIA/Swinburne University of Technology [14,15]. The framework aimed to advance TCP congestion control research by enabling congestion control algorithms to be implemented as dynamically loadable kernel modules, significantly reducing the time and effort required to develop and evaluate new congestion control implementations. Key features of ModCC include:

- Congestion control routines are implemented as a pointer to a set of functions defined in the cc_algo structure. Congestion control modules implement unique functionality by overriding relevant functions in the structure.
- Algorithms can define and attach per-connection private data through the cc_data pointer.
- The cc_var structure provides a connection between the TCP control block and the linked congestion control algorithm, allowing bidirectional access to related variables including the TCP control block itself, state flags, and cc_data.
- The TCP control block was extended to include pointers to cc_algo and cc_data, allowing congestion control algorithms to be allocated per connection. Different algorithms can be conditionally allocated where required and even changed mid-connection if needed.

```
struct cc_algo {
  char  name[TCP_CA_NAME_MAX];

  /* Init global module state on kldload. */
  int (*mod_init)(void);

  /* Cleanup global module state on kldunload
    . */
  int (*mod_destroy)(void);

  /* Init CC state for a new control block.
    */
  int (*cb_init)(struct cc_var *ccv);

  /* Cleanup CC state for a terminating
    control block. */
  void  (*cb_destroy)(struct cc_var *ccv);

  /* Init variables for a newly established
    connection. */
  void  (*conn_init)(struct cc_var *ccv);

  /* Called on receipt of an ack. */
  void  (*ack_received)(struct cc_var *ccv,
    uint16_t type);

  /* Called on detection of a congestion
    signal. */
  void  (*cong_signal)(struct cc_var *ccv,
    uint32_t type);

  /* Called after exiting congestion recovery
    . */
  void  (*post_recovery)(struct cc_var *ccv);

  /* Called when data transfer resumes after
    an idle period. */
  void  (*after_idle)(struct cc_var *ccv);

  /* Called for an additional ECN processing
    apart from RFC3168. */
  void  (*ecnpkt_handler)(struct cc_var *ccv)
    ;

  /* Called for {get|set}sockopt() on a TCP
    socket with TCP_CCALGOOPT. */
  int     (*ctl_output)(struct cc_var *,
    struct sockopt *, void *);

  STAILQ_ENTRY (cc_algo) entries;
};
```

Listing 3. Definition of `cc_algo` structure.

The `cc_algo` structure is shown in Listing 3. ModCC provides routines to register new algorithms, making them accessible to new TCP connections. The registered algorithms can be viewed using the sysctl interface variable `net.inet.tcp.cc.available`. The default algorithm used for new connections can be viewed or set using the variable `net.inet.tcp.cc.algorithm`.

Congestion control is implemented by controlling the number of segments that can be in transit at any time, known as the congestion window (*cwnd*). The default CC algorithm used by FreeBSD is TCP NewReno which uses the conventional additive increase multiplicative decrease (AIMD) method to dictate the *cwnd* size. *cwnd* increases exponentially to a threshold and then increases by one on each successful transmission. When a segment is lost, *cwnd* is halved. This approach presents several limitations:

- *cwnd* increases slowly resulting in periods of under-utilized bandwidth.
- *cwnd* continues to increase until packet loss occurs.
- *cwnd* is drastically reduced when packet loss occurs.

The proposed design seeks to use machine learning to improve TCP performance by providing a more stable *cwnd* response, which attempts to predict and react before packet loss occurs.

5. Kernel modifications of FreeBSD V13.1

The major changes made to the kernel in FreeBSD 13.1 (since the previous release of the MPTCP patch) and the associated justifications are presented below. See DRL MPTCP for implementation details.

sys/conf/kern.pre.mk: The content of this file has changed considerably and the lines to be modified are no longer present in V13.1. This is a simple change to disable compiler optimizations, so the corresponding change is implemented by setting the *-O0* compile option in the build string.

sys/kern/uipc_socket.c: A new function, *sopoll_mpstream*, is defined in the patch, and it is functionally identical to *sopoll_generic* in V11.0. There are several changes to the *sopoll_generic* function between V11.0 and V13.1. To ensure compatibility with the V13.1 kernel, *sopoll_mpstream* is created as a copy of the V13.1 *sopoll_generic* instead of the existing patch content.

The *soreceive_mpstream* function is omitted from V13.1 as, while defined as a new function in the patch, has also been fully commented out.

sys/netinet/in_pcb.c: A new function, *in_pcballoc_subflow*, is defined in the patch to support allocating Protocol Control Blocks (PCBs) for MPTCP subflows. This function is a duplicate of *in_pcballoc* in V11.0 with two key differences:

- The function returns *inpcb* instead of the success state. If an error occurs, the function returns *null*.
- The *inpcb* is not allocated to the socket.

There are several changes to the *in_pcballoc* function between V11.0 and V13.1. To ensure compatibility with the V13.1 kernel, *in_pcballoc_subflow* is created as a modified copy of the V13.1 *in_pcballoc* instead of the original patch content.

sys/netinet/tcp_usrreq.c The *tcp_attach* function has been removed in V13.1 and all functionality merged into *tcp_usr_attach*, however, the MPTCP code makes specific calls to *tcp_attach* when attaching subflow sockets. To support MPTCP in V13.1's TCP stack, the *tcp_attach* function is reconstructed based on the V13.1 *tcp_usr_attach* function, with an appropriate call placed in *tcp_usr_attach* to replace the now duplicate functionality.

sys/netinet/tcp_usrreq.c A small change is made to the logic of the `tcp_usrclosed` function. In V13.1, the call to `tcp_state_change` has been moved up to the TCPS_LISTEN case, before the fall-through to TCPS_CLOSED. In V11.0, however, this call is made in the TCPS_CLOSED case. To preserve the expected logic for the MPTCP patch, the call to `tcp_state_change` has been replaced with `tcp->t_state = TCPS_CLOSED` (as detailed in the patch) but placed in the TCPS_CLOSED case. This will have no impact on the TCPS_LISTEN case, but needs to be observed for errors as the V13.1 logic for the TCPS_CLOSED case, even for non-MPTCP connections, has slightly changed.

sys/netinet/mptcp_subr.c The *TCP_LINGERTIME* constant and associated socket linger resets have been removed from the FreeBSD kernel.[6] All references have been similarly removed from affected MPTCP functions. The *sbsndptr* function has been retired[7] and replaced with *sbsndptr_noadv*. All references have been updated accordingly.

[6] "kern: net: remove tcp lingertime". https://reviews.freebsd.org/rG4c0bef07be071a1633ebc86a653f9bd59d40796e, 2021.

[7] "Retire sbsndptr() kpi". https://reviews.freebsd.org/rS340591, 2018.

The *ACCEPT_LOCK* global mutex is obsolete and has been removed from the FreeBSD kernel. The corresponding references have been updated accordingly.

An issue is discovered in the *mp_create_subflow_implicit* function causing the kernel to panic when attempting to join subflows. The fault is traced to the call to *solisten_proto*. The *solisten_proto* function configures a socket to a listening state and in V11.0 performed only two actions — set the queue limit to a specified number and set the *SOACCEPT_CONN* flag. The *solisten_proto* function has seen substantial changes between V11.0 and V13.1. The most impacting changes are efficiencies added for resource management, where resources that are not required to perform listening functions are de-allocated or destroyed. Due to limitations in the MPTCP implementation and the way the TCP *syncache* resolves entries, an alternate mechanism is implemented to handle the handshaking and allocation of subflow sockets. This alternate implementation requires MPTCP subflow sockets to be created as a listening socket on receipt of the initial SYN, and then expanded to a full socket on completion of the handshake. In V13.1, configuring the socket as a listening socket using *solisten_proto* destroys resources which are later required when the subflow handshake is complete. To prevent this from happening, the call to *solisten_proto* has been replaced with two statements — set the queue limit and set the *SOACCEPT_CONN* flag. This reverts the subflow socket creation behavior to match the V11.0 implementation without impacting non-MPTCP listen socket creation.

sys/netinet/mptcp_usrreq.c: The `INP_INFO_RLOCK` and related macros are no longer used and have been removed from the latest version of FreeBSD.[8] All instances of `INP_INFO_RLOCK` have been replaced with `INP_INFO_WLOCK`.

sys/netinet/mptcp_timer.c: The *ACCEPT_LOCK* global mutex is obsolete and has been removed from the FreeBSD kernel. References have been updated accordingly.

Calls to *CURVNET_RESTORE* have been removed. There are no preceding calls to set the current VNET. The calling restore here is invalid.

/sys/netinet/tcp_reass.c: The TCP reassembly code has received a substantial rewrite[9] since the development of the original MPTCP patch. The changes in the patch are not applicable to the *tcp_reass* function in the V13.1. The new TCP reassembly function required modified to support MPTCP, ensuring compatibility with both the V13.1 TCP stack code while preserving the expected behavior from the MPTCP implementation. An analysis of the modification made to the function (since V11.0) determined three key functional changes:

- Prevent collapsing *mbuf* chains as sequential TCP packets may not represent contiguous bytes at the data-level.
- Do not present packets to the user (via socket) at the TCP reassembly stage. Packets must be reassembled at the MPTCP layer.
- Queue subflow *mbuf* chains to tcp control block for MPTCP reassembly.

The V13.1 *tcp_reass* algorithm uses a tail queue to store in-sequence segments. Elements in the tail queue contain both the original *mbuf* and some additional metadata. Presentation to the user writes the *mbuf* from in-sequence elements to the socket. As the new TCP reassembly code no longer collapses the in-sequence *mbuf* chains, no further modification is required with respect to this change.

The V13.1 TCP reassembly function contains two relevant return paths requiring modification — one for handling new segment entries and one for inserting a segment into the queue. Both return paths have been modified to remove direct writes to the socket, and instead chain the in-sequence mbufs to the TCP control block for MPTCP reassembly.

/sys/netinet/tcp_input.c: The *TI_WLOCKED* and *TI_UNLOCKED* constants are obsolete and have been removed from the FreeBSD kernel. The references to these constants have being passed as arguments to the *tcp_do_segment* function. As *tcp_do_segment* no longer requires this parameter, these obsolete references have been removed/updated. The *INP_INFO_UNLOCK_ASSERT* macro is obsolete and has been removed from FreeBSD. References have been replaced with *INP_INFO_WUNLOCK_ASSERT*.

An error is discovered within the initial MPTCP patch which caused random termination of MPTCP connections due to detection of a Data Finish (DFIN) flag by the received when the flag is not set in the TCP options field. The fault is traced to random values in the *dtag-dss_flags* field. The field has not been initialized on creation, so instead it retained the value in the previously assigned memory on allocation. A minor modification is made to *tcp_do_segment* to set *dss_flags = 0* following allocation of the dtag for incoming segments.

/sys/netinet/mptcp_usrreq.c: The *INP_INFO_RLOCK* and *INP_INFO_RUNLOCK* macros are obsolete and have been removed from FreeBSD. References have been replaced with INP_INFO_WLOCK and INP_INFO_WUNLOCK respectively.

sys/amd64/conf/MPTCP: The *SCTP* option has been removed from sys/amd64/conf/GENERIC and replaced with of *SCTP_SUPPORT*. As MPTCP requires hashing functions provided by *SCTP* modules, the *SCTP* option has been added to the MPTCP kernel configuration.

The *VIMAGE* option, which enables virtualization of the network stack, was officially added to the GENERIC kernel configuration in October 2017.[10] The MPTCP patch code, released in 2016, is not currently compatible with FreeBSD's virtual network stack and the *VIMAGE* option has been overridden and removed in the MPTCP kernel configuration. This is a significant limitation of the current implementation, and, while overriding the virtual stack is suitable for the scope of initial implementation and evaluation, the MPTCP code must be modernized to support the virtual network stack before formal incorporation into FreeBSD could be considered.

6. New system calls of `ModCC` in FreeBSD-13.1

We implement a novel kernel CC module, referred to as `CC-DRL`, to deploy the `ModCC` interface, and a user-space module to process packet transfer statistics for enabling ML models plugins into the MPTCP. To this end, two new system calls are added to the kernel to support the marshaling of relevant data transport statistics between the user and the kernel space. These system calls are implemented as functions in the `CC-DRL` kernel module. All modified or added kernel file names are listed in Table 2 and are explained below.

6.1. CC-DRL kernel module

The CC-DRL kernel module is responsible for handling kernel-side operations inline with the `ModCC` structure. As processing and decision making are being transferred to the ML processes in the user space, the kernel module only needed to implement a reduced subset of `ModCC` functions (as shown in Listing 4) to connect with the ML modules in the user space.

```
struct cc_algo drl_cc_algo = {
  .name = "drl",
  .mod_init = drl_mod_init,
  .cb_init = drl_cb_init,
  .cb_destroy = drl_cb_destroy,
  .ack_received = drl_ack_received,
  .cong_signal = drl_cong_signal,
  .after_idle = drl_after_idle
};
```

Listing 4. CC-DRL Implementation

[8] "Remove now unused inp info rlock macros". https://reviews.freebsd.org/rS354490, 2019.

[9] "Remove now unused inp info rlock macros". https://reviews.freebsd.org/rS354490, 2019.

[10] "Enable vimage by default". https://reviews.freebsd.org/D12639, 2017.

The details of the functions in Listing 4 are as follows.

drl_mod_init: Allocate and initialise module variables.

drl_cb_init: Allocate and initialise per-connection variables. Register CC instance for *cwnd* updates.

drl_cb_destroy: De-register the CC instance for *cwnd* updates.

drl_ack_received: Handle successful transmission events.

drl_cong_signal: Handle congestion (as indicated by packet loss) events.

drl_after_idle: Reset *cwnd* after connection has been idle.

6.2. Marshaling data from kernel to user space

The `drl_get_buffer` system call, shown in Listing 5 is added as the primary mechanism to transfer data between the kernel and user space. Its function copies a buffer containing `size` elements of type `pkt` to the calling process. The definition of the `pkt` structure is shown in Listing 6.

```
584 AUE_NULL  STD {
    int drl_get_buffer(
      _Out_writes_(size) void *data,
      _Out_ int *size
    );
  }
```

Listing 5. Definition of drl_get_buffer system call

```
struct pkt {
  u_int   cwnd;
  int   smoothed_rtt;
  int   cong_events;

  u_int   laddr;
  u_int   lport;
};
```

Listing 6. Definition of struct pkt

A module-level tail queue, `pkt_queue`, is provided to collect packet events until the next call to `drl_get_buffer()`. To avoid excessive consumption of kernel memory, the queue size is limited by `CC_DRL_MAX_QUEUE`, which has a default value of 10 000. The current size of the queue is tracked by `pkt_queue_size`. The oldest elements in the queue (i.e. those at the head) are dropped once the queue reaches its capacity. The `pkt_queue_mtx` mutex ensures that only a single thread can manipulate the queue at one time.

According to normal congestion control, the respective functions, `ack_received` or `cong_signal` is called by `tcp_input` in response to successful or unsuccessful transmissions. For CC-DRL the behavior of `drl_ack_received` and `drl_cong_signal` is essentially the same as the logic blocks typically present in these functions, are to be handled by the user process. The kernel-side behavior for both functions only performs the following:

- Allocate a new `pkt` structure.
  - Populate the congestion window (`cwnd`) and smoothed round trip time (`smoothed_rtt`) fields from the TCP control block.
  - Populate the local address (`laddr`) and local port (`lport`) fields from the input control block.
  - Populate the congestion event field (`cong_events`); 0 for `drl_ack_received` and 1 for `drl_cong_signal`.
- Attach the `pkt` structure to the `pkt_queue`.
- Update the TCP control block send congestion window value.

6.3. Marshaling data from ML to kernel

The `drl_update_cwnd` system call, shown in listing 7 facilities updating a congestion window of connections in the user space. The connection is located using the `laddr` and `lport` parameters. The congestion window is set to the value provided by the `cwnd` parameter.

```
583 AUE_NULL  STD {
    int drl_update_cwnd(
      u_int cwnd,
      u_int laddr,
      u_int lport
    );
  }
```

Listing 7. Definition of drl_update_cwnd system call

A module-level linked list, `ccv_list`, is provided to hold a reference to the `cc_var` structure for all active connections using CC-DRL. Connections register themselves as part of `drl_cb_init()` and remove themselves as part of `drl_cb_destroy()`. The list allows locating the associated `cc_data` structure based on `laddr` and `lport` parameters passed to the `drl_update_cwnd` system call. The `ccv_list_mtx` mutex ensures only a single thread can manipulate the list at one time, and prevents manipulation (registration/removal) while the list is being accessed by the system call.

6.4. Understanding algorithm latency

A significant challenge for the proposed pluggable MPTCP mechanism is the additional latency introduced due to the overhead of coordinating statistics between the user and the kernel space and the processing delays due to the complex internal computing involved in training and predicting ML. Nevertheless, the proposed CC-DRL architecture and its implementation facilitate an asynchronously-interacting interface between kernel-level MPTCP processing and user-level ML processing. When developing such an asynchronous coupling, we always seek to ensure that packet processing in the standard TCP stack is seamless and never adversely affected. One obvious limitation of such an approach is the kernel of temporal lag and the knowledge base of user space on network congestion.[11]

6.5. Handling events per subflow

The congestion window is specific to each MPTCP subflow, so it is important to provide the ability to update the congestion window for a specific path rather than globally across all subflows. The combination of local address and local port is chosen to uniquely identify a connection. The corresponding fields (`laddr` and `lport`) are added to the `pkt` structure, and as inputs to `drl_update_cwnd()`. When `drl_update_cwnd()` is called, `ccv_list` (see 6.3) is traversed to find the corresponding connection and access its structure `cc_data`. Future work should look at replacing the linked list with a hash table structure to improve look-up times as the number of registered connections grows.

[11] The study of tightly coupled design where ML models and MPTCP are fully synchronized requires a different approach and will be investigated in the future.

Table 3
Kernel files affected by modular scheduler framework.

File	Status
sys/conf/files	Modified
netinet/mp_sched/mp_sched.c	Added
netinet/mp_sched/mp_sched.h	Added
netinet/mp_sched/mp_sched_module.c	Added
netinet/mp_sched/mp_sched_module.c	Added
netinet/mptcp_subr.c	Modified
netinet/mptcp_var.h	Modified
netinet/tcp_subr.c	Modified

6.6. Kernel instability

Thread safety is a concern for kernel stability in early designs of the architecture. Areas of concern that are addressed by our design include:

Kernel to user transfers: It is possible for a kernel thread to attempt to queue a new node via calls to `drl_ack_received()` or `drl_cong_signal()` while it is being drained by `drl_get_buffer()`. The `pkt_queue_mtx` is added to ensure only a single thread could manipulate the queue at any time. To prevent excessive kernel delays during calls to `drl_get_buffer()`, all nodes are initially moved to a temporary queue and the `pkt_queue_mtx` is unlocked, allowing event collection to continue on the main `pkt_queue`, while the system call continues to process the temporary queue.

User to kernel transfers: As discussed in Section 6.5, a mechanism is required to allow the congestion window to be updated on a per-connection basis. Initial designs of this feature investigated the option of passing the `cc_data` pointer, which could then be used to access the memory directly. This approach is unsafe as the TCP connection may terminate before all acknowledgments have been handled by the user DRL process. The resulting calls to `drl_update_cwnd()` may then attempt to access and modify memory which is no longer allocated to the TCP control block. The `ccv_list` approach allows the call to be safely discarded if the corresponding connection has been removed for any reason.

The current MPTCP implementation does not appropriately handle packet loss on sub-flows. Packet losses, either due to congestion or random losses, will typically cause a panic in the receiving host's kernel. Future investigation need to be conducted to address this issue.

6.7. Modular scheduling framework

FreeBSD's MPTCP implementation currently relies on a fixed round-robin scheduler algorithm embedded in the multipath output routine. To enhance flexibility for experimenting with new multipath scheduler algorithms, we introduce a novel scheduler framework. This framework facilitates the development and testing of dynamic loadable kernel modules for schedulers, leveraging FreeBSD's lightweight 'ModCC' [14] designed for dynamically loadable congestion control algorithms. Key features of the framework include the replacement of scheduler routines with pointers to a set of functions, the ability for schedulers to allocate algorithm-specific per-connection memory, association of schedulers and their allocated memory with the Multipath Control Block (MPCB) on a per-connection basis, dynamic loading and unloading of schedulers, and configuration of schedulers through the system control (sysctl) interface. Table 3 provides a summary of the kernel source files affected by the proposed scheduling framework.[12]

Configuration of the scheduling framework is achieved by the introduction of two new system control variables:

(i) *net.inet.tcp.mptcp.scheduler.available*;

(ii) *net.inet.tcp.mptcp.scheduler.algorithm*

The control variables serve two purposes: (i) provides a read-only list of scheduler modules registered with the framework, formatted as a comma-separated variable; (ii) governs the presently selected scheduler algorithm. When assigned a value, the variable sets the default scheduler algorithm for new multipath connections. The specified scheduler must be among the registered modules. When invoked without a value, the variable returns the current default scheduler algorithm. If the currently selected default algorithm is unloaded, the list of registered algorithms is updated, and the default reverts to round-robin. Any active multipath connections using the unloaded module will also revert to round-robin.

For consistency, we suggest that any newly introduced scheduler module, which provides configuration options through the system control facility, follows a standardized approach. Specifically, this involves creating a new group under the *net.inet.tcp.mptcp.scheduler* system control group, with the group name matching the algorithm's name. All configuration options specific to the algorithm should be neatly listed within this newly created group. This approach ensures a uniform structure for exposing and managing configuration options across different scheduler modules.

The fundamental framework structure, "mp_sched_algo", is defined in *netinet/mp_sched/mp_sched.h* and is shown in Listing 8. An additional structure, "mp_sched_var", which stores connection and algorithm specific memory is shown in Listing 9. A scheduler algorithm is required to implement an instance of the mp_sched_algo structure and is uniquely identified by the "name" member. The algorithm is not required to implement all of the function members, but can instead choose to implement those relevant to its operation. At a minimum, a scheduler should define "get_subflow" which is used by the MPCB to select a subflow for transmission.

```
1  struct mp_sched_algo {
2      char  name[MPTCP_SA_NAME_MAX];
3
4      /* Init global module state on kldload.
       */
5      int (*mod_init)(void);
6
7      /* Cleanup global module state on
       kldunload. */
8      int (*mod_destroy)(void);
9
10     /* Init MP scheduler state for a new
       control block. */
11     int (*cb_init)(struct mp_sched_var *
       mpschedv);
12
13     /* Cleanup MP scheduler state for a
       terminating control block. */
14     void  (*cb_destroy)(struct mp_sched_var
       *mpschedv);
15
16     /* Request sub-flow selection for
       transmission */
17     struct sf_handle *  (*get_subflow)(
       struct mp_sched_var *mpschedv);
18
19     /* Called on receipt of a data ack. */
20     void  (*dack_received)(struct
       mp_sched_var *mpschedv);
21
22     STAILQ_ENTRY (mp_sched_algo) entries;
23 };
```

Listing 8. Structure of mp_sched_algo

[12] See Modular Multipath Scheduling for details.

Table 4
Kernel files affected by DQN multipath scheduling.

File	Status
sys/conf/files	Modified
netinet/mp_sched/mp_sched_dqn.c	Added
netinet/mp_sched/mp_sched_dqn.h	Added
sys/compat/freebsd32/syscalls.master	Modified
sys/conf/files	Modified
sys/kern/init_sysent.c	Modified
sys/kern/syscalls.master	Modified
sys/kern/systrace_args.c	Modified
sys/sys/syscall.h	Modified
sys/sys/syscall.mk	Modified
sys/sys/sysproto.h	Modified

```
struct mp_sched_var {
    /* Per-connection private scheduler
    algorithm data. */
    void    *mp_sched_data;

    /* Pointer to multipath control block */
    struct mpcb *mp;
};
```

Listing 9. Structure of mp_sched_var

The MPTCP control block (MPCB) struct defined in *netinet/mptcp_var.h* has two new members. The "struct mp_sched_algo *mp_sched_algo" member stores a pointer to the multipath scheduler algorithm structure. The "struct mp_sched_var *mpschedv" is used to attach a pointer to be MPCB and connection specific memory for use by scheduler algorithm. The "mpschedv" has been added as a member of "struct mpcb_mem" and is initialized alongside the MPCB. Scheduler hooks have been inserted into relevant functions defined in *netinet/mptcp_subr.c*. The "mp_newmpcb()" function assigns a pointer to the MPCBs "mpschev", assigns the current default scheduler algorithm to "mp_sched_algo" and calls "cb_init" to initialize the scheduler. The "mp_discard" function cleans up the scheduler prior to discarding the MPCB by calling "cb_discard" and clearing the pointers assigned to "mp_sched_algo" and "mpschedv". The "mp_output" function is responsible for invoking the scheduler by calling "get_subflow". The subflow returned by the function is then used for transmission.

A new multipath scheduler module, *dqn*, was added using the modular scheduler framework described above. The module contains all of the data structures and sub-routines for the Kernel Side Scheduler Algorithm and Data Transfer Mechanism. Four new system calls were added to facilitate user space interaction with the module. The implementation of these system calls are also contained within the module source files. Table 4 provides a summary of the kernel source files.

7. Performance evaluation

Our FreeBSD V13.1 MPTCP implementation and modular multipath scheduling have been tested in a variety of scenarios to demonstrate the correctness of the implementation protocol and evaluate performance. Each scenario was evaluated using the transfer of a single file of 10 MB. The test environment comprised four virtual machines running on Windows 10: two virtual hosts and two virtual routers. The topology of the test environment is shown in Fig. 2, where each path bandwidth is fixed at 8 Mbps.

For the topology shown in Fig. 2, we can reproduce all of the results using V11 as shown in [15]. For tractability [5], we repeat the same experimental topology and we have found that our results of FreeBSD V13.1 are very similar to those of V11 [5]. In particular, we have classified our evaluation into three categories to quantify and understand the (i) compatibility and interoperability of MPTCP V11 and V13.1 FreeBSD implementation; (ii) single-path performance of our V13.1 FreeBSD MPTCP stack; (iii) throughput performance of V13.1 FreeBSD MPTCP stack when compared with V13.1 engineering release TCP stack.

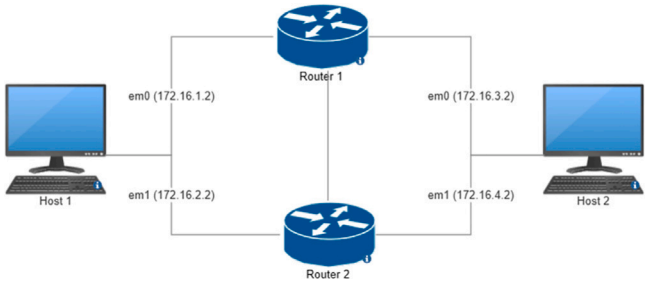


Fig. 2. Experimental topology of our experimental setup comprised 4 VirtualBox machines running on Windows 10 — two virtual hosts with two virtual routers. 10 MB file transfer across both paths fixed in Routers 1 and 2 at 8 Mbps bandwidths.

Before we explain and demonstrate our findings for each category in the following, we provide a clear and detailed account of how we tackled the design and implementation challenges as follows.

(i) Minimizing System Call Overheads. To mitigate system call overheads, we introduced the CC-DRL pluggable kernel module, enabling asynchronous interaction between the MPTCP stack and ML models in user space. This design aimed to reduce latency caused by system call overheads. In our evaluation, we conducted performance assessments, comparing scenarios with and without the CC-DRL module. Preliminary results indicate improvements in latency reduction and enhanced resource utilization.

(ii) Handling User-Kernel Events for Congestion Window per Subflow. Addressing the challenge of handling events for the congestion window per subflow involved uniquely identifying subflows using a combination of local address and port. These identifiers were incorporated into the pkt structure, enabling efficient congestion control updates in the `drl_update_cwnd()` function. Evaluation included rigorous testing of multiple subflow scenarios, monitoring proper identification and congestion control updates through packet traces, logs, and standard diagnostic tools.

(iii) Understanding Thread Safety and Ameliorating Kernel Instability. To ensure thread safety and alleviate kernel instability, we implemented careful design considerations. Specifically, we allowed the kernel thread to queue a new node while being drained by `drl_get_buffer`, enhancing thread safety. We avoided unsafe practices, such as passing `cc_data` pointers directly in kernel thread attempts. The evaluation involved stress testing the system with various concurrency scenarios, ensuring that changes did not lead to race conditions or other thread safety issues.

Concerning the handling of User-Kernel Events for the Congestion Window per subflow, where subflows can be easily identified via a map, this simplifies the challenge. However, several original aspects distinguish it from the V11 implementation.

(i) Asynchronous Interaction. Coordinating the asynchronous interaction between user space and kernel space, especially for subflow-based events like Windows and RTT updates, adds complexity. Efficiently handling events triggered by user space actions, such as updating congestion control parameters based on ML predictions, requires careful actuation via updates (different from V11).

(ii) Data Marshaling and Representation. Efficiently marshaling and representing MPTCP subflow-related data between user and kernel space is crucial. This involves translating subflow-indexed parameters and events into a format suitable for ML models and vice versa.

(iii) Real-time Response. In the context of MPTCP internal dynamics implementation, real-time response is crucial. Handling user-kernel events per subflow promptly, especially when adjusting MPTCP and

subflow parameters dynamically based on ML predictions, requires stringent timing considerations.

(iv) Concurrency and Scalability. MPTCP connections often involve multiple concurrent subflows. Ensuring the scalability of ML-based MPTCP handling in FreeBSD is challenging, particularly when dealing with a large number of concurrent connections and asynchronous interaction using threads for events.

(v) Dynamic Adaptation. MPTCP subflow parameters may need dynamic adaptation based on network conditions and application requirements. Implementing adaptive strategies with ML introduces challenges in dynamically adjusting model predictions and adapting to varying network scenarios.

(vi) Fault Tolerance and Reliability. Guaranteeing fault tolerance and reliability is critical. The system should be resilient to errors, unexpected events, and disruptions to prevent degradation in TCP performance and maintain network communication reliability.

(vii) Interoperability and Compatibility. Ensuring ML-based MPTCP implementation interoperability with existing MPTCP and TCP implementations and compatibility across different FreeBSD versions is an ongoing challenge.

In addition to these challenges, successfully addressing the concurrent execution and synchronization challenges mentioned earlier is crucial to prevent issues like race conditions and data inconsistencies. Ensuring thread safety requires designing and implementing system components and libraries to avoid non-thread-safe operations. Scalability concerns need to be addressed to achieve optimal performance as the number of threads increases. Efficient kernel resource management is essential, involving tasks such as balancing memory and CPU usage among multiple threads. Unfair allocation of resources or increased contention can lead to performance degradation. Addressing these challenges requires a comprehensive understanding of both ML and kernel-level networking principles, along with careful consideration of FreeBSD-specific requirements and constraints.

7.1. Compatibility and interoperability (V11 vs. V13.1)

Looking ahead, using the topology shown in Fig. 2, we conducted two separate experiments for the joint backward compatibility testing of MPTCP under FreeBSD-11 and FreeBSD-13.1 as follows.

Experiment I. Host 1 running the V13.1 MPTCP kernel to Host 2 running the V11.0 MPTCP kernel. Listing 10 shows the initial MPTCP handshake and subflow join. The evolution of the throughputs of each of the subflows and the total MPTCP throughput is shown in Fig. 3.

Experiment II. Host 1 running the V11.0 MPTCP kernel to Host 2 running the V13.1 MPTCP kernel. Listing 11 shows the initial MPTCP handshake and subflow join. The evolution of the throughputs of each of the subflows and the total MPTCP is shown in Fig. 4. It is important to note that while flow 1 continues in Fig. 4 until connection termination, no packet transfer occurs in the 5 s–6 s interval.

```
172.16.1.2 -> 172.16.3.2   MPTCP   [SYN]
       Seq=0 Win=65535 Len=0 MSS=1460 WS=64
    SACK_PERM=1 TSval=741765698 TSecr=0
172.16.3.2 -> 172.16.1.2   MPTCP   [SYN, ACK]
       Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
    WS=64 SACK_PERM=1 TSval=1518632725 TSecr
    =741765698
172.16.1.2 -> 172.16.3.2   MPTCP   [ACK]
       Seq=1 Ack=1 Win=65728 Len=0 TSval
    =741765698 TSecr=1518632725
172.16.2.2 -> 172.16.3.2   MPTCP   [SYN]
       Seq=0 Win=65535 Len=0 MSS=1460 WS=64
    SACK_PERM=1 TSval=19366780 TSecr=0
172.16.3.2 -> 172.16.2.2   MPTCP   [SYN, ACK]
       Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
    WS=64 TSval=122131 TSecr=19366780
```

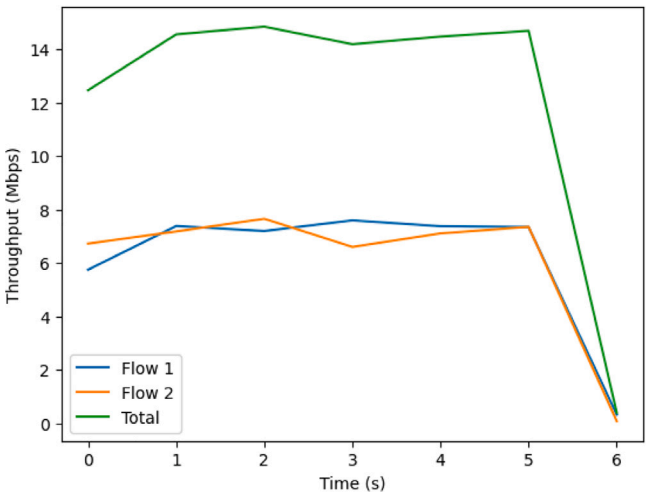


Fig. 3. File transfer from MPTCP FreeBSD-13.1 to MPTCP FreeBSD-11.0: Temporal evolution of throughputs.

```
172.16.2.2 -> 172.16.3.2   MPTCP   [ACK]
       Seq=1 Ack=1 Win=65728 Len=0 TSval
    =19366780 TSecr=122131
```

Listing 10. Handshake and subflow join for File Transfer from FreeBSD13.1 MPTCP to FreeBSD11

```
172.16.1.2 -> 172.16.3.2   MPTCP   [SYN]
       Seq=0 Win=65535 Len=0 MSS=1460 WS=64
    SACK_PERM=1 TSval=148005 TSecr=0
172.16.3.2 -> 172.16.1.2   MPTCP   [SYN, ACK]
       Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
    WS=64 SACK_PERM=1 TSval=3873287288 TSecr
    =148005
172.16.1.2 -> 172.16.3.2   MPTCP   [ACK]
       Seq=1 Ack=1 Win=66560 Len=0 TSval
    =148007 TSecr=3873287288
172.16.2.2 -> 172.16.3.2   MPTCP   [SYN]
       Seq=0 Win=65535 Len=0 MSS=1460 WS=64
    SACK_PERM=1 TSval=148026 TSecr=0
172.16.3.2 -> 172.16.2.2   MPTCP   [SYN, ACK]
       Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
    WS=64 TSval=93442 TSecr=148026
172.16.2.2 -> 172.16.3.2   MPTCP   [ACK]
       Seq=1 Ack=1 Win=66560 Len=0 TSval
    =148028 TSecr=93442
```

Listing 11. Handshake and subflow join for File Transfer from FreeBSD11.0 MPTCP to FreeBSD13.1

Our first main observation from **Experiment I** and **Experiment II**. is as follows.

Observation 1. Both the MPTCP send and receive operations show that the MPTCP handshake between the initial connection and the joining subflow was successful. The performance of each subflow is comparable, the implementations of V11 and V13.1 are compatible with each other, and the total MPTCP throughput is greater than the 8 Mbps cap set in each router.

7.2. Single path operation of MPTCP FreeBSD-13.1

For tractability, we now test to evaluate and demonstrate how the implementation correctly reverts to the standard TCP operation when communicating using only single path. We have the following experiment.

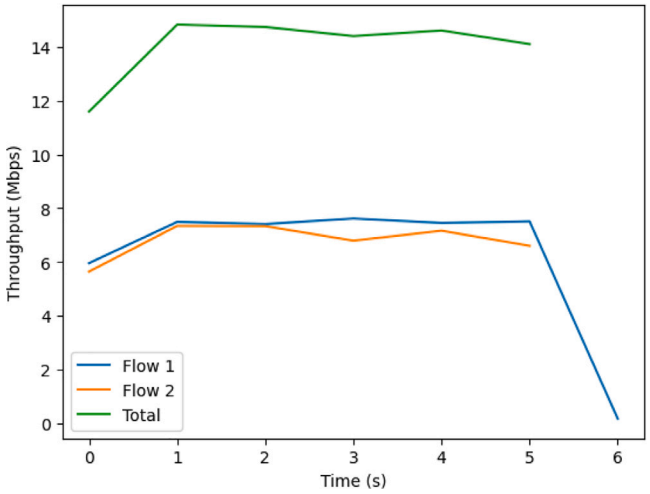


Fig. 4. File transfer from MPTCP FreeBSD-11 to MPTCP FreeBSD-13.1: Temporal evolution of throughputs.

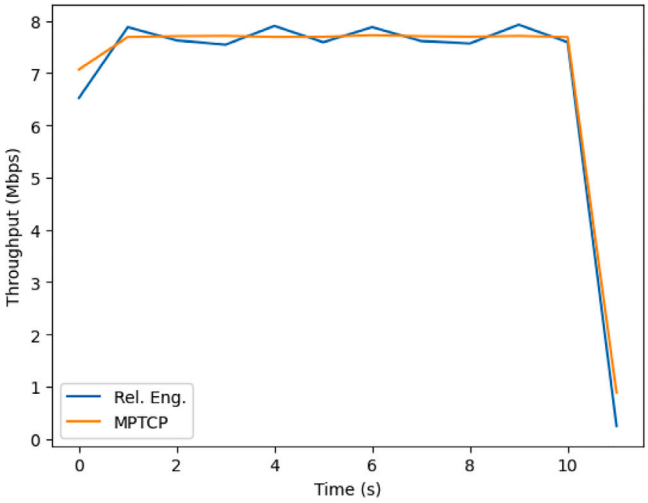


Fig. 5. FreeBSD13.1 MPTCP (**this project**) vs. Release engineering throughput.

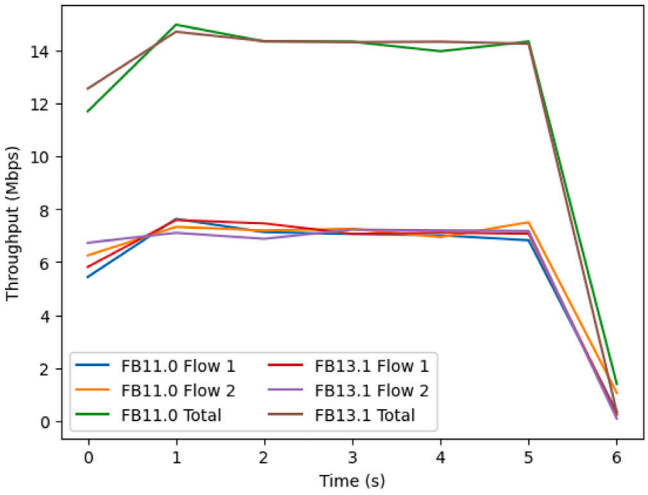


Fig. 6. FreeBSD13.1 vs. FreeBSD11.0 MPTCP throughput.

Experiment III. With the same topology as in Fig. 2, Host 1 uses the V13.1 MPTCP kernel, while Host 2 uses the standard V13.1 engineering release kernel (without MPTCP). We run the same experiment of transferring a single file of 10 MB where 8 Mbps cap is set in each router for the paths. Our observation of the establishment of the connection and handshake is shown in the listing 12. Observe in Listing 12 that host 1 sends MPTCP-enabled SYN, host 2 responds with TCP-only SYN/ACK, and then host 1 responds with TCP-only ACK. We have observed the same behavior when there is only one path available between Host 1 and Host 2.

```
172.16.1.2 -> 172.16.3.2   MPTCP    [SYN]
      Seq=0 Win=65535 Len=0 MSS=1460 WS=64
   SACK_PERM=1 TSval=2809731826 TSecr=0
172.16.3.2 -> 172.16.1.2   TCP      [SYN, ACK]
      Seq=0 Ack=1  Win=65535 Len=0 MSS=1460
   WS=64 SACK_PERM=1 TSval=2590951192 TSecr
   =2809731826
172.16.1.2 -> 172.16.3.2   TCP      [ACK]
      Seq=1 Ack=1 Win=65728 Len=0 TSval
   =2809731836 TSecr=2590951192
```

Listing 12. Fallback to TCP for Non-MPTCP-Enabled Communications

Observation III. The proposed implementation of MPTCP V13 behaves seamlessly as the FreeBSD engineering release TCP when the receiving host or the network does not support multipath data transfer, thus confirming our design considerations of backward compatibility.

7.3. Throughput comparison: Standard FreeBSD 13.1 kernel

We run an experiment to compare the difference in throughput of the FreeBSD-13.1 MPTCP kernel against the FreeBSD-13.1 engineering release kernel by executing data transfer using a single flow.

Experiment IV. Using the topology in Fig. 2, where host 1 uses the V13.1 MPTCP kernel and host 2 uses the standard V13.1 engineering release kernel, we transfer a single file of 10 MB where an 8 Mbps cap is set in each router for the paths. The MPTCP kernel handles the transmission using the MPTCP protocol with a single subflow. The engineering release kernel does the same transmission using regular TCP. The throughput of each transfer is demonstrated in Fig. 5.

Observation IV. We find comparable throughput during both experimental runs in the same network setup — one using engineering release TCP stack V13 and another the proposed MPTCP V13 implementation. Fig. 5 demonstrates that the MPTCP implementation developed under V13.1 does not introduce significant overheads and bottlenecks that adversely affect the network throughput performance when integrated into the FreeBSD kernel.

7.4. Throughput comparison with FreeBSD 11.0 MPTCP

This test evaluates the performance of our FreeBSD V13.1 MPTCP kernel against the original CAIA FreeBSD MPTCP kernel V11 by performing a transfer using two sub-flows.

Experiment V. Using the topology in Fig. 2, a first experimental run where both Host 1 and Host 2 use V13.1 MPTCP kernel and a second experimental run where both Host 1 and Host 2 use CAIA V11 MPTCP kernel while transferring a single file of 10 MB over 8 Mbps paths.

The throughput of each subflow, along with the combined total, for each kernel is shown in Fig. 6.

Observation V. We find a comparable throughput performance of both the V13.1 and V11.0 implementations, demonstrating that the inclusion of pluggable modules and system calls on FreeBSD MPTCP V13.1 has introduced no significant performance overheads.

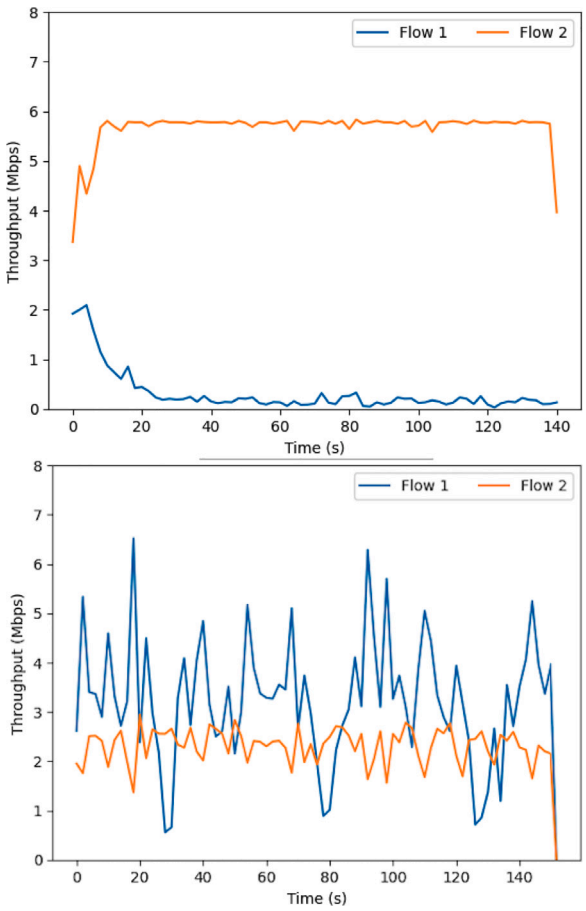


Fig. 7. FreeBSD13.1 DRL-MPTCP (bottom panel) vs. FreeBSD13.1 MPTCP (top panel) flow level throughputs over Flow 1 (10 Mbps, 10% packet loss) and Flow 2 (8 Mbps, 3% packet loss).

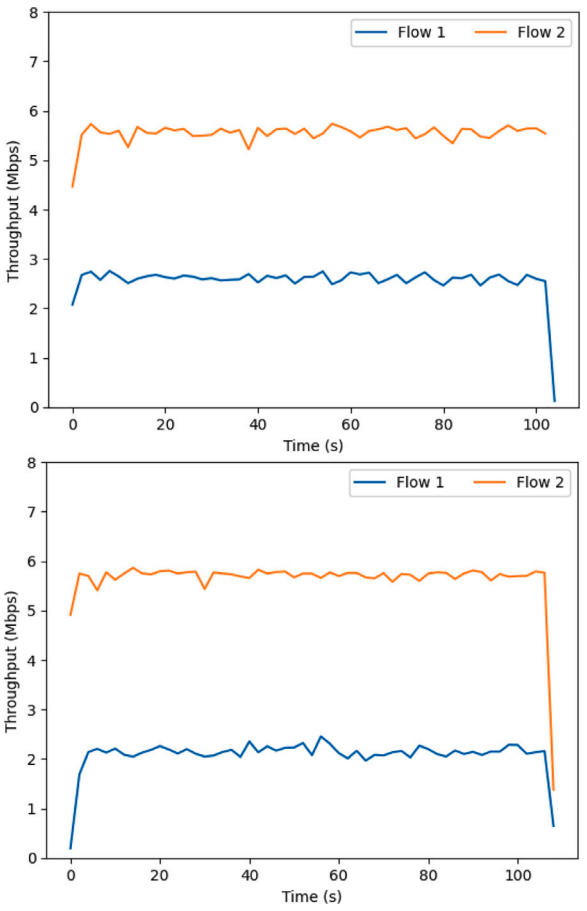


Fig. 8. DRL-MPTCP FreeBSD13.1 (top panel, **this project**) vs. DRL-MPTCP Linux (bottom panel, [13]) MPTCP flow-level throughputs when 20 MB file transfer over Flow 1 (8 Mbps, 5% packet loss) and Flow 2 (5 Mbps, 2% packet loss) paths.

7.5. With ML driven congestion control

This test evaluates the performance of our FreeBSD V13.1 MPTCP kernel against the FreeBSD V13.1 DRL-MPTCP with similar and different packet losses over paths by performing a transfer using two sub-flows. It also benchmarks the FreeBSD V13.1 DRL-MPTCP with the existing Linux equivalent proposed in [13].

Experiment VI. Using the topology in Fig. 2, a first experimental run where both Host 1 and Host 2 use V13.1 MPTCP kernel and a second experimental run where both Host 1 and Host 2 use DRL-MPTCP V13.1 while transferring a large single file over Flow 1 (10 Mbps, 10% packet loss) and Flow 2 (8 Mbps, 3% packet loss) paths.

The throughput of each subflow, along with the combined total, for each kernel is shown in Fig. 7.

Observation VI. We find a comparable throughput performance of the V13.1 implementation with and without DRL at the usespace, demonstrating that the inclusion of ML and system calls on FreeBSD MPTCP V13.1 has significant performance gains.

Similar to **Observation VI** in a different context performance improvement of DRL-MPTCP has already been reported over Linux implementation [2,13].

Experiment VII. Using the topology in Fig. 2, a first experimental run where both Host 1 and Host 2 use V13.1 DRL-MPTCP kernel and a second experimental run where both Host 1 and Host 2 use DRL-MPTCP Linux [13] while transferring a large single file over Flow 1 (8 Mbps, 5% packet loss) and Flow 2 (5 Mbps, 2% packet loss) paths.

The throughput of each subflow for each kernel is shown in Fig. 8.

Observation VII. We find a comparable throughput performance of the V13.1 DRL-MPTCP implementation with and without DRL-MPTCP Linux, demonstrating that the inclusion of ML and system calls has significant similar gains.

8. Concluding remarks

We present a novel ML-driven design and implementation aspects of MPTCP congestion control and multipath scheduling with the related parameter settings based on our experience in constructing the first publicly available FreeBSD implementation under 13.1. This article will be helpful to the Internet research community's ongoing effort to evaluate and independently implement innovative multipath congestion control algorithms. The congestion window evolutions of the MPTCP subflows, as reported in the IETF and original MPTCP articles, are reproduced by this independent implementation. We perform preliminary research on how the ongoing advances in MPTCP congestion control mechanism, including machine learning-based approaches, can be evaluated using the developed pluggable implementation.

8.1. Impending challenges and future directions

This research work presents several challenges for future work and further directions. Some of the impending problems we identified during our design and implementation include the following.

1. *Ablation of Kernel Panic.* As discussed earlier, the current MPTCP implementation over FreeBSD suffers from kernel panic where

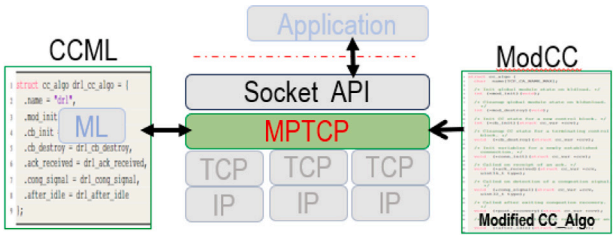


Fig. 9. Anticipated architecture of multipath TCP implementation under FreeBSD-13 for pluggable ML models. The core idea is to redesign the MPTCP implementation in FreeBSD and enhance ModCC in the FreeBSD TCP stack. We must design and develop a new CCML kernel module to incorporate ML models. Such a design will avoid the need to marshal data from the kernel to ML models in the user space and vice versa.

several consecutive packet losses occur in the flows. This requires the development of a new handler module to respond effectively to correlated packet loss and retransmissions in MPTCP subflows.

2. *Application layer ML interacting with MPTCP under FreeBSD*. A full-fledged implementation of ML, DRL, and other application-agnostic ML modules with MPTCP requires further implementation, which is beyond the scope of this kernel-based implementation work. Current DRL implementations build upon the Tensorflow or PyTorch libraries, which FreeBSD does not support. However, it can be achieved by pursuing integration and support of the PyTorch and Tensorflow libraries under FreeBSD, or by developing open-source ML/DRL implementation as a pluggable application layer module, with the illustrated application layer developments, integration, and the tuning of the congestion control considering parameter selection, reward mechanisms, and weighting, online or offline performance over synchronous vs. asynchronous operation.

8.2. Tightly coupled ML-based stack

Green-field tightly coupled implementation of ML, as anticipated in Fig. 9, in the MPTCP stack at the transport layer under FreeBSD for jointly handling all subflows is a way moving forward, instead of handling subflows independently from the ML modules running at the application layer. However, it also requires extensive research and improved design beyond the existing state-of-the-art Linux-based implementation. In principle, the BSD kernel is not a stand-alone monolithic kernel like Linux but is developed as modular, being part of a whole, and therefore such an integrated design seems feasible in FreeBSD (which is non-trivially challenging in Linux).

Implementing the high-level idea shown in Fig. 9 requires a significant rethink and redesign of MPTCP implementation in FreeBSD and will improve the design of ModCC and rebuild the TCP stack demonstrated in this work. Such design and development of a new CCML kernel module to incorporate ML models can avoid the need to marshal data from the kernel to ML models; the system calls from the user space to kernel and vice versa, which is beyond the scope of this work and will be considered in future.

CRediT authorship contribution statement

Shiva Raj Pokhrel: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Jonathan Kua:** Funding acquisition, Methodology, Project administration, Supervision, Writing – review & editing. **Brenton Fleming:** Formal analysis, Software, Validation, Visualization, Writing – original draft. **Sebnem Ozer:** Funding acquisition, Investigation, Resources, Supervision, Writing – original draft, Writing – review & editing. **Jeff Howe:** Funding acquisition, Supervision, Writing – original draft, Writing – review & editing. **Anwar Walid:** Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This research was funded by COMCAST Innovation Fund 2022 “Experimental Implementation of new Multipath TCP and AQM in FreeBSD” Comcast Corporation.

References

[1] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, C. Paasch, TCP extensions for multipath operation with multiple addresses, rfc 8684, 2020, IETF.
[2] S.R. Pokhrel, J. Choi, A. Walid, Fair and efficient distributed edge learning with hybrid multipath TCP, IEEE/ACM Trans. Netw. (2022).
[3] B.Y.L. Kimura, et al., Interpath contention in MultiPath TCP disjoint paths, IEEE/ACM Trans. Netw. 27 (4) (2019) 1387–1400.
[4] S. Kumar, M.P. Andersen, H.-S. Kim, D.E. Culler, Performant {tCP} for {low-power} wireless networks, in: 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 20, 2020, pp. 911–932.
[5] N. Williams, Implementing a Multipath Transmission Control Protocol (MPTCP) stack for FreeBSD with pluggable congestion and scheduling control (Ph.D. thesis), 2016.
[6] W. Li, et al., SmartCC: A reinforcement learning approach for multipath TCP congestion control in heterogeneous networks, IEEE J. Sel. Areas Commun. (2019).
[7] Z. Xu, et al., Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning, IEEE J. Sel. Areas Commun. (2019).
[8] J. Luo, X. Su, B. Liu, A reinforcement learning approach for multipath TCP data scheduling, in: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC, 2019, pp. 0276–0280.
[9] S.R. Pokhrel, S. Garg, Multipath communication with Deep Q-Network for Industry 4.0 automation and orchestration, IEEE Trans. Ind. Inform. (2020) 1.
[10] T. Mai, H. Yao, Y. Jing, X. Xu, X. Wang, Z. Ji, Self-learning congestion control of MPTCP in satellites communications, in: 2019 15th International Wireless Communications & Mobile Computing Conference, IWCMC, IEEE, 2019, pp. 775–780.
[11] B. Liao, G. Zhang, Z. Diao, G. Xie, Precise and Adaptable: Leveraging Deep Reinforcement Learning for GAP-based Multipath Scheduler, in: 2020 IFIP Networking Conference (Networking), IEEE, 2020, pp. 154–162.
[12] F. Silva, M.A. Togou, G.-M. Muntean, AVIRA: Enhanced Multipath for Content-aware Adaptive Virtual Reality, in: 2020 International Wireless Communications and Mobile Computing, IWCMC, IEEE, 2020, pp. 917–922.
[13] S.R. Pokhrel, A. Walid, Learning to harness bandwidth with multipath congestion control and scheduling, IEEE Trans. Mob. Comput. (2021) early access.
[14] G. Armitage, L. Stewart, M. Welzl, J. Healy, An independent H-TCP implementation under FreeBSD 7.0: Description and observed behaviour, ACM SIGCOMM Comput. Commun. Rev. 38 (3) (2008) 27–38.
[15] L. Stewart, J. Healy, Light-Weight Modular TCP Congestion Control for FreeBSD 7, Tech. Rep. 071218A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 2007.
[16] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, S.-M. Hu, {Tcp-fuzz}: Detecting memory and semantic bugs in {tCP} stacks with fuzzing, in: 2021 USENIX Annual Technical Conference, USENIX ATC 21, 2021, pp. 489–502.
[17] F. Jowkarishasaltaneh, An SDN-based Network Layer Solution to Improve the Fairness and Throughput of Multi-path TCP (Ph.D. thesis), Phd Thesis, Swinburne University of Technology, 2023.
[18] Y. Cheng, N. Cardwell, N. Dukkipati, P. Jha, RFC 8985: The RACK-TLP loss detection algorithm for TCP, 2021.
[19] N. Williams, L. Stewart, G. Armitage, Design Overview of Multipath TCP version 0.4 for FreeBSD-11, Tech. Rep. 140822A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 2014.
[20] S.R. Pokhrel, M. Mandjes, Improving multipath TCP performance over WiFi and Cellular Networks: an analytical approach, IEEE Trans. Mob. Comput. (2018).

[21] M. Kheirkhah, I. Wakeman, G. Parisis, Multipath-TCP in ns-3, 2015, arXiv preprint arXiv:1510.07721.
[22] B. Chihani, C. Denis, A multipath TCP model for ns-3 simulator, 2011, arXiv preprint arXiv:1112.1932.
[23] M. Prakash, A. Abdrabou, On the fidelity of ns-3 simulations of wireless multipath tcp connections, Sensors 20 (24) (2020) 7289.
[24] K. Nadeem, T.M. Jadoon, An ns-3 MPTCP implementation, in: Quality, Reliability, Security and Robustness in Heterogeneous Systems: 14th EAI International Conference, Qshine 2018, Ho Chi Minh City, Vietnam, December 3–4, 2018, Proceedings 14, Springer, 2019, pp. 48–60.

Dr Shiva Raj Pokhrel is a distinguished Marie Curie Fellow, currently serves as a Senior IEEE Member and is a Senior Lecturer in Mobile Computing at Deakin University Australia. He earned his PhD in ICT Engineering from Swinburne University of Technology, Australia. Dr Pokhrel's extensive career includes a tenure as a research fellow at the University of Melbourne and a seven-year role as a Network Engineer. Dr Pokhrel's cutting-edge research spans semantic communication, federated learning, quantum machine learning, 6G, cloud computing, dynamics control, the Internet of Things, and cyber-physical systems. His innovative work finds practical applications in intelligent manufacturing, satellite networks, autonomous vehicles, and smart cities.

Recognized for his expertise, Dr. Pokhrel has served on the editorial boards of leading journals in his field. He has also played pivotal roles in organizing major IEEE/ACM conferences, including IEEE INFOCOM, IEEE GLOBECOM, IEEE ICC, and ACM MobiCom, where he has been Workshop Chair and Publicity Co-Chair. Dr. Pokhrel was honoured with the prestigious Marie Curie Fellowship, underscoring his significant contributions to the field of mobile computing and his dedication to pioneering technological advancements. Beyond his academic and research achievements, Dr Pokhrel is an active member of seven working groups within the Internet Society and was recently named an APNIC Fellow and Australian Government Internet Engineering Task Force (IETF) Ambassador.

Dr Jonathan Kua received the B.Eng. (First Class Hons.) degree in telecommunications and network engineering and the Ph.D. degree in telecommunications engineering from Swinburne University of Technology, Melbourne, Australia, in 2014 and 2019, respectively. He is currently Senior Lecturer in Internet of Things within the School of Information Technology at Deakin University, Melbourne, Australia. His research interests include data transport protocols and active queue management, adaptive streaming and content delivery, Internet of Things and Industry 4.0, with an overarching focus on improving their network performance and quality of service. He was the recipient of the Netflix Ph.D. scholarship award (2015-2019) and the second runner-up of the DASH-IF Best Ph.D. Dissertation Award on "Algorithms and Protocols for Adaptive Content Delivery over the Internet" (2019).

Mr Brenton Fleming received the Bachelor of Software Engineering (Honours) from Deakin University, Australia, in 2023. He was an undergraduate student researcher working on computer networking-focused projects, including Multipath TCP under FreeBSD. He is currently an Engineering Delivery Manager within the Training and Simulation business at Thales Australia. Prior to the current role, he worked for Thales Australia as Senior Software Engineer focusing on aircraft simulation. His research interests include real-time systems, modular software design, and engineering life-cycle practices, operating systems development, and building secure software for aviation systems.

Dr Şebnem Zorlu Özer (S'99–M'03) received the B.Sc. degree in electronics and telecommunications engineering with highest honors from Istanbul Technical University, Istanbul, Turkey, in 1992, the M.Sc. degree in electrical engineering from Bogazici University, Bogazici, Turkey, in 1995, and the Ph.D. degree in electrical engineering from New Jersey Institute of Technology, Newark, in 2001. From 1997 to 2001, she was a Research Assistant with the New Jersey Center for Multimedia Research, New Jersey Institute of Technology. Since 2001, she has been in top-notch industry such as MeshNetworks, Motorola, ARRIS, COMCAST and Charter communications. Her areas of interest are wired and wireless technologies with extensive experience in HFC, PON, Wi-Fi, SDN/NFV, and IP networks. Her experience in managing multisite engineering teams, leading projects with cross-functional organizations, and contributing to industry standards with strong technical skills to drive business opportunities on new technologies are phenomenal. She has 21 granted patents and numerous publications, and patent submissions.

Jeff Howe, currently the Director of DOCSIS Engineering at Comcast, previously held the position of VP, Systems Engineering at ARRIS. In this role, he directed architectural work and future product planning. Before his time at ARRIS, Jeff served as the Director of System Architecture at the startup Cadant, overseeing architecture and integration for the development of the next-gen C4 Cable Modem Termination System. His extensive career includes managerial, architectural, hardware and software development, and test roles at Lucent Bell Laboratories, where he contributed to various data communications platforms. Jeff holds a BSEE from the University of Wisconsin, an MSEE from Stanford University, and an MBA from Capella University. His research interests focus on highspeed switching and Quality of Service (QoS) for next generation converged services platforms.

Prof Anwar Walid received the B.S. and M.S. degrees in electrical and computer engineering from New York University, New York City, NY, USA, and the Ph.D. degree from Columbia University, New York City, NY, USA. He was at Nokia Bell Labs, Murray Hill, NJ, USA, as the Head of the Mathematics of System Research Department and as the Director of University Research Partnerships. He is currently the Director of Network Intelligence and Distributed Systems Research and a Distinguished Member of the Research Staff at Nokia Bell Labs. He is also an Adjunct Professor at the Electrical Engineering Department, Columbia University. He has over 20 U.S. and international granted patents on various aspects of networking and computing. His research interests are in the control and optimization of distributed systems, learning models and algorithms with applications to Internet of Things (IoT), digital health, smart transportation, cloud computing, and software-defined networking. He is a fellow of the IEEE, and an Elected Member of the International Federation for Information Processing Working Group 7.3 and the Tau Beta Pi Engineering Honor Society. He received awards from the IEEE and ACM, including the 2017 IEEE Communications Society William R. Bennett Prize and the ACM SIGMETRICS/IFIP Performance Best Paper Award. He served as an Associate Editor for the IEEE/ACM TRANSACTIONS ON CLOUD COMPUTING, IEEE Network Magazine, and the IEEE/ACM TRANSACTIONS ON NETWORKING. He served as the Technical Program Chair for the IEEE INFOCOM, as the General Chair for the 2018 IEEE/ACM Conference on Connected Health (CHASE), and as a Guest Editor for the IEEE IoT Journal — Special Issue on AI-Enabled Cognitive Communications and Networking for IoT.