FlexCP: A Scalable Multipath TCP Proxy for Cellular Networks

DUCKWOO KIM, KAIST, South Korea
YOUNGGYOUN MOON, Samsung Research, South Korea
JAEHYUN HWANG, Sungkyunkwan University, South Korea
KYOUNGSOO PARK, KAIST, South Korea

Research has shown that Multipath TCP (MPTCP) improves the quality of a TCP connection by exploiting multiple paths, but its adoption in the wide area network is still fledgling. While MPTCP-TCP proxying is often employed as a practical solution, the performance of a split-connection proxy is suboptimal – it wastes CPU cycles on content relaying between two connections while it does not efficiently leverage multiple CPU cores in packet processing.

We present FlexCP, a high-performance MPTCP-TCP proxy based on the following properties. First, FlexCP operates by translating the two protocols on a packet level. This approach not only avoids the overhead of flow reassembly and memory copying, but it greatly simplifies the implementation as the proxy stays away from reliable data transfer, socket buffer management, and per-hop congestion/flow control. Second, FlexCP maintains connection-to-core affinity for multiple subflows of the same MPTCP connection and its corresponding TCP connection by leveraging SmartNIC. This enables a lock-free implementation for packet processing, which significantly improves the performance. Our evaluation demonstrates that FlexCP achieves 281 Gbps of connection proxying on a single machine, outperforming existing proxies by up to 6.3× in terms of throughput while it incurs little extra latency over direct TCP/MPTCP connections.

CCS Concepts: • **Networks → Transport protocols**; **In-network processing**.

Additional Key Words and Phrases: MPTCP, TCP proxy, packet processing, ATSSS

ACM Reference Format:

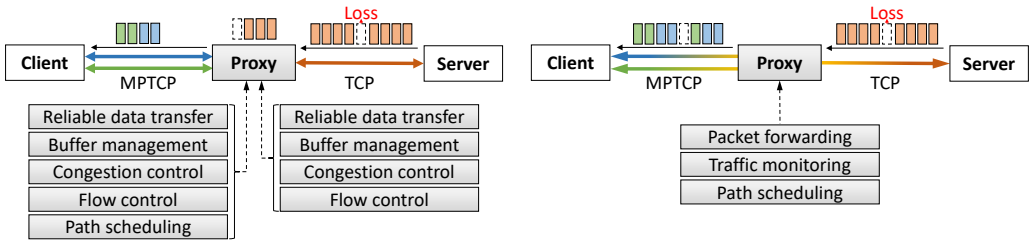
Duckwoo Kim, YoungGyoun Moon, Jaehyun Hwang, and KyoungSoo Park. 2023. FlexCP: A Scalable Multipath TCP Proxy for Cellular Networks. *Proc. ACM Netw.* 1, CoNEXT3, Article 21 (December 2023), 21 pages. https://doi.org/10.1145/3629143

1 INTRODUCTION

Multipath TCP (MPTCP) [15, 44] presents a great potential in improving the latency, throughput, and resilience to packet loss of existing TCP connections by exploiting multiple network paths between two end hosts. It is attractive not only to client devices that can leverage multi-homing (e.g., Wi-Fi and cellular networks) [10, 39], but also to data center networks that can leverage multiple internal paths [43]. In fact, the cellular network community has recently standardized the Access Traffic Steering, Switching, and Splitting (ATSSS) feature [2] to embrace non-3GPP network

Authors' addresses: Duckwoo Kim, 0731kdw14@gmail.com, KAIST, Daejeon, South Korea; YoungGyoun Moon, ygyoun.moon@samsung.com, Samsung Research, Seoul, South Korea; Jaehyun Hwang, jh.hwang@skku.edu, Sungkyunkwan University, Suwon, South Korea; KyoungSoo Park, kyoungsoo@gmail.com, KAIST, Daejeon, South Korea.

2834-5509/2023/12-ART21 $15.00
https://doi.org/10.1145/3629143



(a) Connection-terminating MPTCP-TCP proxy (b) FlexCP without connection termination

Fig. 1. Comparison of MPTCP-TCP proxies with and without connection termination

access (e.g., Wi-Fi), and some cellular ISP has already adopted MPTCP for hybrid, multi-path network access [24]. On the client side, MPTCP has been supported by Linux and iOS networking stacks [4, 7].

Unfortunately, the MPTCP adoption by the end-to-end connections in wide area network (WAN) is still very fledgling. According to the measurements from 2021 to 2022 [5], the protocol adoption has increased rapidly in recent years, but the absolute number of MPTCP-capable servers is fairly small – they report less than 20K MPTCP-capable servers in the entire IPv4/IPv6 space [1]. This is partly because old middleboxes filter out the MPTCP options [5] or more seriously, MPTCP-unaware load balancers could malfunction, which disrupts the correct operation of a backend server farm [49]. Nevertheless, the current status quo not only gives up the opportunity to exploit multiple network paths but it also blocks the deployment of innovative new cellular network architectures such as Cellbricks [30] that depends on MPTCP.

A more practical approach for wider deployment on the client side is MPTCP-TCP proxying as seen in the deployment by KT [8] and Tessares [46]. This approach is in part backed by the ubiquitous support for an MPTCP networking stack by modern smartphones [4, 11]. Protocol-level proxying enables a cellular ISP to provide blanket multi-path support to their subscribers as part of the ATSSS solution. Or one can deploy a reverse proxy that translates MPTCP to TCP (and vice versa) before old middleboxes. However, as the wireless bandwidth in the cellular network has substantially improved from 3G to LTE/5G, the performance bottleneck has gradually moved to the connection proxying system in the cellular core network. In fact, we observe that MPTCP-TCP proxying lowers the throughput of a persistent MPTCP connection by 3.1× to 3.4× over a direct end-to-end connection, and up to 10.2× over a direct TCP connection in our experiments.

The root cause for suboptimal performance of the protocol-translating proxy lies in its connection-terminating (or split-connection) architecture illustrated in Figure 1a. This design presents two performance issues. First, a significant portion of CPU cycles are wasted on connection-level proxying as demonstrated also in TCP-TCP proxying [17, 36]. Each connection segment (either an MPTCP connection from a client to the proxy or a TCP connection from the proxy to a server) needs to run reliable data transfer, buffer management, congestion and flow control, etc. Also, the proxy can relay only the flow-reassembled content to the other connection, which incurs frequent memory copies. While Linux's `splice()` may help avoid frequent copies and crossings between kernel and user levels, the overhead for operating two separate TCP/MPTCP connections still remains. Second, efficient management of connection-to-core affinity is difficult for packet processing in the existing MPTCP implementation. That is, multiple subflows that belong to the same MPTCP connection can be mapped to different CPU cores as NIC-based receive-side scaling (RSS) does not work for different four tuples of MPTCP subflows. The lack of connection-to-core

[1] A large fraction of them belong to Apple, Inc., as they have publicly announced the support for MPTCP [4].

affinity requires frequent synchronizations through locking for processing received packets, which leads to inefficient CPU cache usage, and it ultimately limits the scalability on multicore systems.

In this paper, we present FlexCP, a scalable MPTCP-TCP proxy on a multicore system for multi-homed clients. FlexCP addresses the challenges described above as follows. First, FlexCP relays the two logical connections of heterogeneous protocols with a single physical connection that translates the headers on a packet level without connection termination (see Figure 1b). This effectively avoids the TCP protocol conformance overhead of each segment and substantially improves the performance. Second, FlexCP leverages the programmability of a modern network interface card (NIC) to transparently maintain the connection-to-core affinity of the subflows in the same MPTCP connection. The underlying SmartNIC carefully tracks the TCP option of SYN packets to steer the packets from multiple MPTCP subflows of the same connection consistently to a single CPU core. This ensures high processing scalability on a multicore system without locking. Third, FlexCP presents a set of APIs through which the operator can easily configure different packet scheduling policies. FlexCP currently supports load balancing, random, and smallest-RTT-first, but one can implement an additional policy as well.

Our evaluation shows that FlexCP achieves 281 Gbps MPTCP-TCP proxying performance on a 24-core machine with 4 x 100Gbps NICs, outperforming `nginx` by 3.3× to 6.3× in terms of throughput. It also outperforms a high-performance user-level TCP-TCP proxy (`epproxy`) [36] by 1.8× to 3.5× as it effectively avoids the overhead of connection termination. In terms of latency, FlexCP presents almost similar response time to direct TCP/MPTCP connections while reducing the response times of other TCP-TCP and MPTCP-TCP proxies by 1.5× to 3.1×.

2 BACKGROUND

This section presents the background of the MPTCP protocol and its performance implication with connection proxying.

2.1 Overview of the MPTCP Protocol

MPTCP is an extension of the traditional TCP protocol that enhances its capabilities by allowing the establishment of multiple TCP subflows within a single MPTCP-level socket [15]. It achieves this while maintaining compatibility with existing TCP implementations. To create and manage the subflows, MPTCP introduces additional control messages that incorporate new subtype TCP options and a sequence mapping method. These additions provide the necessary mechanisms for negotiating, establishing, and controlling multiple TCP subflows within an MPTCP connection.

Connection establishment. MPTCP follows the standard TCP three-way handshake for establishing an initial connection. The client initiates the MPTCP negotiation by including the MP_CAPABLE option in the SYN packet; if the server supports MPTCP, it responds with its own MP_CAPABLE option in the SYN-ACK packet. This ensures that both endpoints are aware of the MPTCP capability. Once the handshake is complete, any endpoint can advertise its additional IP addresses and ports using the ADD_ADDR control message, so that it sends the MP_JOIN control message to join the MPTCP connection as a subflow. Conversely, an endpoint can also terminate a specific IP address or port from the MPTCP connection using the REMOVE_ADDR control message. To securely identify the MPTCP connection that a new subflow joins, the initial connection establishment exchanges per-endpoint 64-bit keys and the MP_JOIN option contains a 32-bit token which is a truncated one-way hash of the key. To prevent a replay attack, the MP_JOIN option also authenticates each endpoint with a nonce and hash-based message authentication code (HMAC).

Data sequence mapping. To run multiple independent TCP subflows under the same MPTCP-level connection, MPTCP employs two types of sequence numbers (SNs): data sequence numbers

(DSNs) and subflow sequence numbers (SSNs). Each subflow has its own SN space and maintains its independent set of SNs. And then MPTCP uses a Data Sequence Signal (DSS) option to map the SSNs to the DSNs. This mapping enables MPTCP to manage the ordering of data packets transmitted across different subflows. Note that the same DSNs can be mapped to by different subflows, hence the same data can be sent on multiple subflows at the same time for resilience purposes.

Data splitting and assembling. When an MPTCP sender transmits data across multiple subflows, the MPTCP packet scheduler is responsible for assigning each data packet to one (or more) subflow(s). The default scheduling policy employed by the current kernel implementation is known as Lowest-RTT-First, where the subflow with the lowest RTT among the available subflows is selected to transmit data packets until its TCP sending window (cwnd) is fully utilized. Once the window is filled, the scheduler moves on to the next subflow with the second lowest RTT and continues the transmission. On the receiver-side, both the MPTCP-level socket and the subflow-level socket(s) maintain their own receive buffers and out-of-order buffers while each subflow performs reliable transmission using SSNs, similar to a standard TCP flow. When data packets arrive in order by the SSN, there are two possibilities for the treatment of these packets. If they are also in order in terms of the DSN, MPTCP moves them directly to the receive buffer of the MPTCP-level socket for delivery to the application. Otherwise, they are moved to the out-of-order buffer of the MPTCP socket while waiting for the missing data sequence-level packets to be received from other subflows. This mechanism enables reliable and efficient data delivery in MPTCP, even when there are missing packets across different subflows.

2.2 Proxy Support for Multi-Connectivity

Mobile devices, or User Equipments (UEs), today are equipped with a network modem supporting multiple generations of cellular communication (3G/4G/5G) and Wi-Fi technologies. Prior works have explored the opportunities for utilizing heterogeneous access networks with complementary coverage and capacity, and have shown the potential benefit in terms of better reliability, transmission efficiency, and throughput in real-world deployment cases [26, 28, 35, 42]. Mobile Network Operators (MNOs) are also interested in integrating multiple accesses as they can save duplicate capital expenditure (CapEx) when adopting a newer generation of radio access technologies. Several approaches with different architectural designs have been studied to support seamless data transmission across heterogeneous access networks. A traditional approach in the telecommunications domain is to provide Dual Connectivity (DC) [3] between UEs and base stations. In this approach, the user traffic delivered via multiple accesses are seamlessly distributed and aggregated at Packet Data Convergence Protocol (PDCP) stacks running at UEs and base stations, without requiring any modification on upper-layer protocol stacks and applications. DC-based traffic multiplexing, however, is possible only when the target access network supports the radio protocol stack standardized by 3GPP while the need for integrating non-3GPP accesses is increasing. For example, Wi-Fi APs not maintained by MNOs can be utilized for better indoor coverage, and MNOs also have been working on converging a broadband wireline network with a cellular network for reducing operational cost and enabling new services [16].

To embrace the heterogeneity in access network technologies, the 5G network is designed with the "access-agnostic core" principle, where any type of access network, even an untrusted 3rd-party Wi-Fi network, can be integrated at the core network. In this architecture, handling multi-connectivity is performed in the transport layer, and there are two options for it. The first option is to support multi-connectivity in an end-to-end manner, for example, running an MPTCP stack at servers and clients. This option mandates upgrading all the existing server applications to support MPTCP, which makes it difficult to deploy all at once in practice. The second option is to

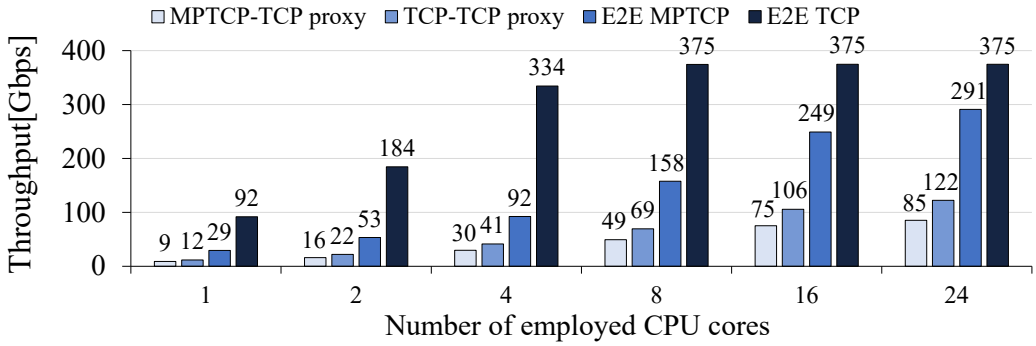


Fig. 2. Performance comparison of an MPTCP-TCP proxy (`nginx`) with split-connection and direct MPTCP or TCP connection (E2E)

place a proxy anchoring multiple connections inside the MNO's network infrastructure. Recent proposals on ATSSS follow this approach so that an MPTCP-TCP proxy can be collocated with a User Plane Functionality (UPF) gateway in the core network. In this option, MNO can monitor the network delivery status of each access network [2], and utilize the information to decide the traffic transmission policy via multiple accesses. The ATSSS specification proposes supporting several steering modes for distributing traffic between accesses in addition to the default "Lowest-RTT-First" scheduling mode in the Linux kernel: (1) active-standby (switching to an available access for reliability); (2) load balancing (splitting traffic across both accesses with a defined ratio); (3) priority-based (steering all traffic to the high-priority access until being congested, and splitting to low-priority access on congestion); (4) redundant (duplicating traffic to multiple accesses if they are available) [2]. The scheduling decision can be made at a service data flow (SDF) level where each SDF consists of one or more IP packet flows, or at a session level (e.g., in case the network has no strong steering requirement for a multi-access session). The scheduling decision is dependent on whether the QoS flow has any guaranteed bit ratio, and can be updated during runtime based on the rules provisioned by control plane (e.g., SMF in 5G) and congestion signal such as RTT and loss rate measured in cellular networks [3].

2.3 MPTCP Proxying Overhead

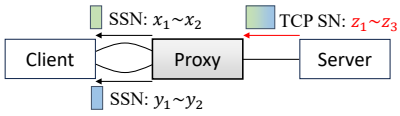
An MPTCP-TCP proxy can be a promising solution for providing multi-connectivity across heterogeneous access networks, but MNOs are now responsible for proxying the entire multi-access sessions within their networks. Traditional proxy models have adopted the *split-connection* design, where data downloads are relayed from one connection over a wired network to another over a mobile cellular network. Such a design often assumes that the cellular networks are the primary bottleneck due to their lower bandwidth and higher latency compared to wired networks. However, the performance bottlenecks are gradually shifting towards the proxies with the evolution of cellular technologies to 5G, which offers significantly increased bandwidth, reaching tens of Gbps, and low latency, often below 8 ms [14]. Therefore, it is now crucial for MNOs to consider the processing overhead introduced by the MPTCP-TCP proxying model to fully leverage the benefits of MPTCP while ensuring optimal performance.

To analyze the overhead of MPTCP-TCP proxying in the existing split-connection architecture, we conduct a performance comparison between an MPTCP-TCP proxy and end-to-end (E2E) TCP and MPTCP scenarios as shown in Figure 2. To push the bottleneck to the proxy machine, we use

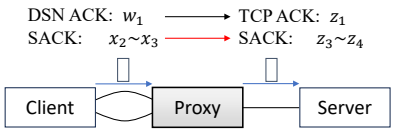
[2]Recently introduced in 3GPP Rel-18 spec [1] to support Ultra-Reliable Low Latency Communications (URLLC)
[3]Performance Measurement Function (PMF) in a 5G core can be used to measure the metrics actively at UE or proxies [2].

TCP SNs	Subflow ID	DSNs	SSNs
$(z_1 \sim z_2)$	1	$(w_1 \sim w_2)$	$(x_1 \sim x_2)$
$(z_2 \sim z_3)$	2	$(w_2 \sim w_3)$	$(y_1 \sim y_2)$
$(z_3 \sim z_4)$	1	$(w_3 \sim w_4)$	$(x_2 \sim x_3)$

(a) A mapping table for TCP SN ranges of the packets sent from the server to MPTCP subflow DSN/SSN ranges. The TCP server sent three packets whose aggregate SNs range from z_1 to z_4, and the proxy forwarded them through two different subflows whose SSNs range from x_1 to x_3 and y_1 to y_2, respectively. Figures (b) and (c) on the right side consult this table for SN translation.



(b) A retransmitted packet may need to be forwarded into multiple different subflows



(c) SACK packet translation from an MPTCP subflow to the TCP connection

Fig. 3. Corner case examples in sequence number translation

400 Gbps links and generate 1,000 persistent connections per CPU core while each connection downloads a distinct 1 MB content. The aggregate size of the downloaded contents exceeds 1 GB, which simulates a memory-bound workload that avoids delivering the content directly from CPU cache. We use `nginx` v1.24.0 as both the proxy and the server in our experiments. Details of machine configurations are described in Section 5.

Figure 2 shows the results from which we make two interesting observations. First, the MPTCP-TCP proxy demonstrates significantly lower throughput compared to the E2E scenarios – E2E connections outperform MPTCP-TCP proxying by up to 3.4× in throughput. This suggests that the split-connection proxying process introduces substantial overhead, impacting the overall data transmission capacity. The main overhead for running two connections is attributed to memory copies due to buffering, in-order data delivery after flow reassembly, per-segment congestion/flow control, etc. Second, when comparing the E2E scenarios of MPTCP and TCP, we observe that the throughput of MPTCP is less than half of TCP's until E2E TCP saturates the link. This is because MPTCP on Linux generally incurs higher processing overhead compared to standard TCP as each subflow is implemented as an independent TCP connection with separate buffers. In addition, MPTCP connection-level operations often require frequent memory copies and locking, which involve multiple subflows while performing data splitting and assembling. However, the performance gap between the TCP-TCP and MPTCP-TCP proxies is not considerably large. This implies that the performance bottleneck of a proxy primarily originates from the nature of the split-connection model itself, rather than inherent protocol overheads. These observations provide valuable insights that allow exploration of an alternative proxying model, specifically, a *splitless-connection* model that mimics the properties of E2E connections while still leveraging the benefits of MPTCP deployment.

3 FLEXCP DESIGN

This section presents the design of FlexCP. Two key traits of the FlexCP design are (1) splitless, packet-level connection proxying and (2) share-nothing parallel connection handling.

3.1 Challenges of Splitless Protocol Translation

The primary design choice of FlexCP is to perform protocol translation of MPTCP/TCP without splitting the E2E connection at the transport layer. While this "splitless-connection architecture"

avoids the performance overheads demonstrated in Section 2, it introduces a number of new challenges mainly due to the intricacy of the MPTCP protocol [15] designed to ensure reliable data transfer even with many concurrent subflows.

First, MPTCP-TCP proxying must maintain a per-connection state for consistency when mapping the TCP SN space to the MPTCP SSN space [4]. This is drastically different from packet-level SN translation in TCP-TCP proxies [17, 36] that can operate completely in a stateless manner by keeping the delta of the initial SNs of the two connections. For instance, when a TCP data packet is forwarded by a proxy to one of the MPTCP subflows, any retransmission of it must go with the same SSN range of the original packet if it is forwarded through the same MPTCP subflow. Also, a TCP retransmission packet may have to be split into multiple smaller MPTCP packets so that each packet can be forwarded into a distinct subflow as shown in Figure 3b. In terms of acknowledgements, a data packet is ACKed not only at the connection level (DATA ACK), but also at the subflow level (subflow ACK), so the system must maintain the mapping between them. Moreover, a selective ACK (SACK) packet from the MPTCP side is expressed as SSN ranges that are independent of the DSN space. So, the proxy should carefully handle it to avoid creating any inconsistency. In general, a SACK packet arriving at a TCP endpoint does not necessarily indicate packet loss since the missing data might have been sent through a different subflow, and the corresponding ACK simply has not arrived yet. To efficiently translate packet-level SNs, one should come up with a scalable data structure for storing and looking up the mapping of SN spaces. Note that FlexCP primarily employs flow-level packet steering, but these problems can still occur when the proxy switches to another subflow path in the middle of network transfer.

Second, in order to achieve multi-core scalability in the proxy system, it is desirable to handle the proxied connections on the same CPU core. This helps avoid any potential lock contention when accessing connection-level and subflow-level metadata for MPTCP-TCP proxied connections. The de-facto standard scheme for server-side load balancing of multiple TCP connections is receive-side scaling (RSS). RSS ensures parallel processing of TCP connections as all packets belonging to the same connection are forwarded to the same CPU core. Furthermore, TCP connection-level proxies can leverage symmetric RSS [48] to consistently map upstream and downstream packets of the same TCP connection to the same CPU core as well. However, as illustrated in Figure 4, one cannot blindly apply symmetric RSS to multiple MPTCP subflows as different subflows would have distinct 4-tuples, resulting in heterogeneous RSS hash values. Consequently, packets belonging to different subflows of the same MPTCP connection would be steered to different CPU cores. We note that the Linux kernel MPTCP stack utilizes locks to process multiple subflows of the same connection [7], but this approach is not scalable. As of now, there is no proposed scheme to affinitize a set of MPTCP subflows and their corresponding TCP connection to the same CPU core. We present our solution to this challenge in Section 3.3.

3.2 Scalable Sequence Number Translation

FlexCP realizes MPTCP-TCP connection proxying through packet header translation. The common operation of FlexCP is to receive a packet, translate its SNs and ACK/SACK numbers, add or remove an MPTCP option, and forward the updated packet to the other endpoint. Internally, it handles MPTCP-specific control packets (e.g., those with MP_JOIN or DATA_FIN) so that they are invisible to the other TCP endpoint. All other TCP control logic such as reliable data transfer (including packet retransmission), send/receive buffer management, congestion/flow control, error processing,

[4]The translation between MPTCP DSN and TCP SN can still be done in a stateless manner as one can leverage a delta in the two SN spaces and a wrap-around counter to handle the difference in the SN space size (32-bit TCP SN vs. 64-bit MPTCP DSN).

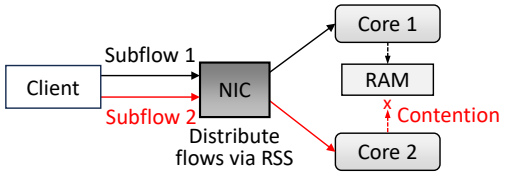


Fig. 4. Broken flow-to-core affinity with RSS for MPTCP subflows

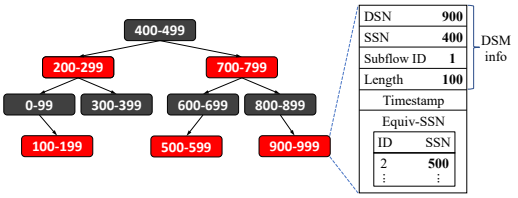


Fig. 5. A sample D-tree for DSM management

etc., is handled by the connection endpoints. Essentially, FlexCP operates as a "smart" packet switch, which simply forwards the received packets from one connection to the other without buffering nor reassembling the received packets in order.

The key challenge lies in translating the SNs between MPTCP subflow and TCP packets. Each subflow manages its own SN space, and an MPTCP connection may employ multiple subflows concurrently. Consequently, mapping the SNs between each individual MPTCP subflow and the corresponding TCP connection can be complex and non-trivial as discussed in Section 3.1. The simplest solution to address this issue would be to maintain a data sequence number mapping (DSM) across DSNs, SSNs and TCP SNs for each data packet. Then, the proxy needs to retain the mapping information for each packet until the whole DSN range is ACKed by the other endpoint [5]. However, a naïve approach like a linear search in the list of the DSMs would not scale as the number of data packets grows.

The D-tree data structure. We address the challenge by introducing our solution called *D-tree*, a balanced binary search tree that maintains a set of unACKed DSMs as illustrated in Figure 5. Each proxied MPTCP-TCP connection keeps two D-tree instances, one for MPTCP-to-TCP and the other for TCP-to-MPTCP SN translation. The D-tree is based on the red-black (RB) tree structure, but it is customized to support SN translation for an MPTCP-TCP proxy. The key of a node is a DSN range, and the node itself contains the DSM information such as the subflow ID (indicating the origin or the destination of the packet), the corresponding SSN range, and so forth. In addition, the D-tree stores a timestamp associated with each packet so that the proxy can measure the RTT on the subflow path. The D-tree supports efficient DSM lookup with a DSN as input; the node whose key overlaps with the input number can be found in O(log n) time, where n is the number of nodes within the tree. The D-tree also supports efficient DSM lookup by an SSN as input and it implements optimizations such as adjacent node merging and batched node removal, which is explained below.

MPTCP-to-TCP packet translation. Upon the arrival of a data packet from an MPTCP subflow, the proxy looks up the D-tree to find a node whose key includes the data SN range of the packet. If a matching node is found, it indicates that the received packet is either a retransmission or a redundant packet from another subflow for resilience [15]. In the latter, the proxy adds the subflow ID and its SSN to the DSM node (i.e., `Equiv-SSN` (equivalent SSN) field) for correct SN translation in the future. If no such node is found, it indicates that the received packet contains new data. Then, the proxy creates a new DSM node using the information in the data sequence signal (DSS) option and inserts it into the D-tree. In this case, the SN allocation in the translated TCP packet is straightforward – the SN of the TCP packet is set to $(DSN_{packet} + \delta_{init})$ where DSN_{packet} refers to the DSN of the packet and δ_{init} is the difference between the initial TCP SN and the initial MPTCP data SN (i.e., initial DSN). This is because the MPTCP DSN space is one-to-one mapped to the

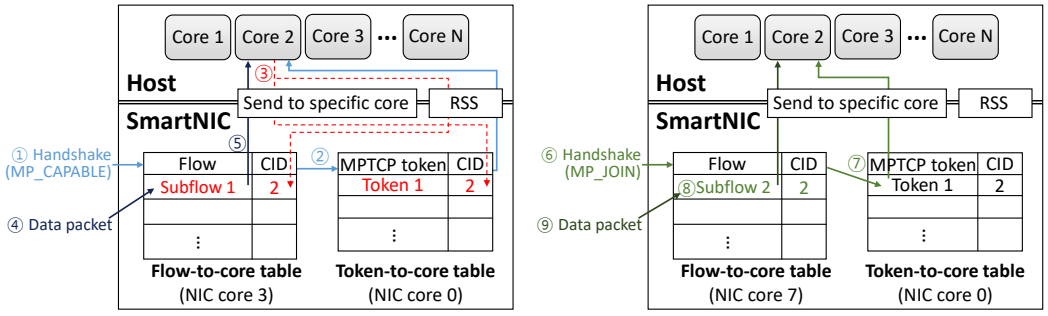
[5]The proxy should keep track of the cumulatively ACKed SN as ACK packets may get lost during transmission.

translated TCP SN space. The ACK number also needs to be translated correctly by looking up the D-tree in the reverse direction. SACK range translation is tricky as SACK numbers are represented in the individual SSN space, and the key of a D-tree node is a DSN range rather than an SSN range. To address the problem, FlexCP maintains the following property for easy SN translation – if two TCP data packets, $p1$ and $p2$, were forwarded through the same subflow and if $DSN_{p1} < DSN_{p2}$, then $SSN_{p1} < SSN_{p2}$ (where DSN_{px} and SSN_{px} refer to DSN and SSN of px, respectively). Each node in the D-tree keeps the corresponding SSN range information, so one can run binary search to efficiently find the node with the SSN. This enables fast SACK range translation into DSNs which are translated into the TCP SNs.

TCP-to-MPTCP packet translation. Upon the arrival of a data packet from the TCP connection, the proxy performs a lookup operation in the D-tree for the corresponding direction. (1) If a DSM node already exists for the packet, it indicates that the packet is a retransmission. In this case, the proxy translates the SN and forwards the packet through the same subflow as sent previously. If the previous subflow path has become unavailable for some reason, the proxy sends the retransmission through a different subflow path. This is a special case and the DSM information must be handled separately from the D-tree as it would violate the property (e.g., if $DSN_{p1} < DSNp2$, then $SSN_{p1} < SSN_{p2}$). Note that a retransmission packet may need to be split into multiple smaller packets that are sent through different subflows. (2) If no node exists for the packet in the D-tree, it represents a new data packet. Then, the proxy selects the appropriate subflow, creates and adds a new DSM node to the D-tree, and forwards the packet with the correct DSS option. The proxy ensures that SSN_{packet} is monotonically increasing as DSN_{packet} in the same subflow grows in the D-tree. Even if a new TCP data packet arrives out of order, the proxy still forwards it without buffering the packet. Then, the proxy fills in the SN hole in the D-tree by inserting dummy DSM nodes with the same subflow ID.

Batched DSM node removal. When an ACK packet is received, the proxy removes DSM nodes from the D-tree that correspond to fully ACKed data SN ranges. However, it retains intermediate nodes (i.e., non-leftmost nodes) that have been ACKed by SACKs as an endpoint receiver may potentially renege ACKed SN range on these intermediate nodes and request retransmission later [31]. Thus, node removal always involves removing the leftmost portion of the D-tree. To minimize the overhead, the node removal process splits the D-tree using the ACKed SN as a pivot, rather than deleting each node at a time, which would require tree rebalancing. During the split operation, all nodes located on the left of the pivot are freed, while the nodes on the right side are preserved as a new D-tree for the direction.

Optimizations. Each D-tree operation runs in O(log n), but rebalancing of the tree with a large number of nodes can be often very costly. Therefore, it is desirable to keep n small in the tree. We enforce two optimizations during node insertion. First, the D-tree actively merges adjacent nodes with the node being inserted if they can be combined. This represents a common case in an uncongested path as data packets are normally forwarded in order without packet loss. Second, the proxy leverages large receive offload (LRO) and generic receive offload (GRO) to merge multiple contiguous data packets into a single packet. Then, the proxy utilizes TCP segmentation offload (TSO) to make the packet size conform to the MTU size. The use of LRO/GRO/TSO allows inserting or removing an MPTCP option without worrying about the MTU size. We also employ an optimization for node removal. Splitting the D-tree can result in multiple tree concatenations to combine the right side of the pivot into a single D-tree. However, each concatenation operation takes O(log n) time, making frequent tree-splits costly. To address this, the proxy (1) batches multiple ACKs and (2) enforces tree-splitting only once when the pivot goes to the right side of the root



(a) Initial MPTCP connection setup (b) A new subflow joins the MPTCP connection

Fig. 6. SmartNIC-assisted MPTCP subflow-to-CPU-core affinity management. (a) The initial MPTCP connection (subflow1) is handled by NIC core 3 and the packets in subflow1 are forwarded to CID (CPU core ID) 2 by RSS. (b) Subflow2 joins the connection with token1, and the packets in it are handled by NIC core 7. Then, NIC core 7 looks up the token-to-core table and maps subflow2 to CID 2 in its flow-to-core table.

node. While the latter may delay node removal, the increase in tree height due to this delay would be at most 1, thus the benefit outweighs the cost.

3.3 Lock-free Connection Relaying on Multicore Systems

FlexCP leverages modern SmartNIC technology to steer all incoming packets from the same MPTCP connection to the same CPU core, regardless of the subflow to which they belong. This enables lock-free processing in connection proxying, which achieves high scalability on multicore systems. Figure 6 illustrates our packet steering scheme that ensures connection-to-core affinity. We configure the internal hardware switch on our SmartNIC (specifically, NVIDIA Bluefield-2) such that packets originating from the client-side MPTCP connection go through the SmartNIC Arm processors, while packets originating from the server side are directly delivered to the host CPU cores. This configuration is based on the observation that client-side MPTCP connections typically send a smaller number of packets (e.g., HTTP requests) to the server, while server-side TCP connections send a much larger number of packets (e.g., HTTP responses) to the client.

Connection initiation. When a client initiates the first MPTCP subflow with a server, FlexCP establishes a TCP connection with the server and starts relaying packets between the two connections (1). CPU core (as well as SmartNIC core) selection for all received packets is achieved through symmetric RSS on the NIC (2). Upon receiving a SYN packet with MP_CAPABLE, the SmartNIC simply forwards the packet to the corresponding CPU core on the host side using symmetric RSS. The CPU core then sends two special command packets to the SmartNIC (3). First, it sends a `Subflow_Create` command packet with an entry of <4 tuples of the subflow, CPU core ID> to the same SmartNIC core that received the SYN packet, and then the SmartNIC core adds the entry to its local flow-to-core table. Second, the CPU core sends a `MPTCPConn_Create` command packet to SmartNIC core #0. The core #0 adds an entry of <MPTCP connection token, CPU core ID> to the global/shared token-to-core table. All subsequent packets from the subflow are forwarded to the same CPU core based on the flow-to-core table lookup (4 and 5). When the client creates another MPTCP subflow within the same connection, the SYN packet with MP_JOIN may be mapped to a different SmartNIC core due to the different source IP address/port (6). However, since the SYN/MP_JOIN packet carries the MPTCP connection token, the SmartNIC core can look up the token-to-core table to determine the CPU core responsible for the MPTCP connection (7). The SmartNIC core then inserts a flow entry of <4 tuples of the new subflow, CPU core ID> to its

own flow-to-core table (8), so that subsequent packets are forwarded to the same CPU core based on the flow-to-core table lookup (9). Note that the flow-to-core affinity is not preserved across SmartNIC cores, as packets from different subflows sent by the client may be received by different SmartNIC cores. However, they are eventually forwarded to the same CPU core on the host side by consulting the per-SmartNIC-core flow-to-core table. We also note that SmartNIC-based packet steering can incur an overhead in case a client needs to upload a large content, but one can minimize the overhead by offloading the flow forwarding rules in the flow-to-core table to NIC hardware using features like ASAP^2 [33] in NVIDIA NICs or Flow Director [21] in Intel NICs.

Subflow/Connection termination. When a subflow is terminated/deleted or encounters an error, the CPU core responsible for the MPTCP connection sends a special command packet (`Subflow_Close`) with the four tuples of the subflow to the SmartNIC. Upon receiving this packet, the SmartNIC removes the corresponding forwarding rule from the flow-to-core table. When the entire MPTCP connection is being terminated, the CPU core sends a `MPTCPConn_Close` command packet to SmartNIC core #0, along with the MPTCP connection token. The SmartNIC then removes the corresponding entry from the token-to-core table.

3.4 Enforcing Flexible Policies at FlexCP

FlexCP enforces the scheduling policy for data packets in the downstream direction (i.e., TCP-to-MPTCP direction). The policy for the upstream transfer is controlled by the client, and the proxy simply forwards the packets to the server side. In the downstream direction, however, the proxy must choose a subflow path for steering the data packets from the server side.

The granularity of traffic scheduling in FlexCP is each flow rather than an individual packet. That is, FlexCP determines the subflow path for each TCP flow rather than spreading the TCP packets in the same flow to multiple subflow paths concurrently. If the quality of the chosen subflow path becomes poor or unavailable, the proxy chooses another subflow path for steering the downstream TCP packets in a single flow. This is mainly because FlexCP adopts the splitless-connection architecture that does not buffer nor reassemble the received packets from the server side. In the splitless-connection architecture, one TCP sender performs the congestion/flow control in the end-to-end path, so it is impossible to do congestion/flow control in the multiple subflows concurrently.

However, we believe that the flow-level traffic steering can support or at least approximate most of the ATSSS scheduling policies in 5G networks. FlexCP naturally supports "active-standby". FlexCP does not fully support "smallest delay (or Lowest-RTT-First)" nor "priority-based" steering as it cannot split the traffic to multiple paths at congestion. However, it can approximate both scheduling modes and it can steer the flow into another available path when the priory path becomes congested. Also, one may not literally achieve "Load balancing" on the packet level with FlexCP, one can still realize the same effect with the flow granularity. To monitor the subflow path quality in real time, FlexCP measures the path RTT by leveraging D-tree structure; each DSM node in the D-tree holds a timestamp value when it is added, and the RTT for the subflow path is computed when an ACK arrives. FlexCP ignores the cases where RTT calculation can be incorrect due to node merging or a DATA_ACK packet that acknowledges DSN sent via the other subflow. FlexCP also provides a method to track the packet loss rate per subflow by running a counter for retransmitted bytes – this can be calculated during the lookup operation on the D-tree. Since FlexCP is responsible for forwarding every data between two endpoints, it can provide an estimated congestion window size of each subflow as well, in case the network operator wants to further improve the scheduling algorithm. We leave the details of those schedulers in FlexCP to our future work.

To allow the users to configure the scheduling policies in a flexible manner, FlexCP provides an interface with the following features for applications. First, FlexCP application can distinguish at least individual MPTCP-TCP connections and possibly each subflows to be managed separately. Second, FlexCP applications can (re)configure the path scheduling decision at runtime. We describe some details of FlexCP API in the next section.

4 IMPLEMENTATION

FlexCP consists of host CPU and SmartNIC stacks that are both implemented as a DPDK [22] application. The host CPU stack reuses the mTCP [23] stack code while adding 6K lines of C code (LoC) for flow scheduling, header translation, and D-tree management. Appendix A briefly explains the packet processing steps in the host stack. The SmartNIC stack includes 3K LoC for managing flow-to-core and token-to-core tables for communication with the host CPU. In this section, we provide the implementation detail of some salient features in FlexCP.

Developer APIs for a splitless-connection proxy Our prototype API introduces a *proxy socket* to abstract a splitless MPTCP-TCP connection. An application creates a listening socket with `socket()`, but it uses the *SOCK_PROXY* type for a proxy connection. Like the existing MPTCP socket interface in the upstream kernel, `accept()` only exposes a new proxy socket that represents an established proxying connection, instead of underlying subflows. However, an application can manage individual MPTCP connections or subflows via `setsockopt()`/`getsockopt()` by providing more specific parameters on the proxy socket. We also add `getsockstat()` to retrieve the accounting information of a socket such as the amount of data that have been forwarded in each direction. FlexCP exposes a knob to configure the forwarding policy and to monitor the connection by extending those socket-level functions to retrieve per-subflow status including RTT and loss ratio, and change the scheduling policy at runtime. At a high level, in ATSSS-enabled 5G networks, SMF can insert the scheduling policy in the form of Multi-Access Rule (MAR). In the future, we plan to update FlexCP to support an RPC interface to allow the agents to translate MAR and update its path scheduler, which performs flow-level traffic steering.

MPTCP connection/subflow establishment. FlexCP handles MPTCP-specific control packets on its own. For the initial MPTCP connection establishment, the proxy runs the 0-RTT TCP convert protocol [9] to smoothly link an MPTCP to a TCP connection. For all subsequent subflow establishments, the proxy handles them without forwarding the control packets to the server side. The timestamp value for a proxy-sent SYN/ACK packet in response to SYN/MP_JOIN is internally adjusted by caching the timestamps exchanged at the initial handshake.

Connection closure. MPTCP employs an extra DATA_FIN packet to terminate an MPTCP connection while a TCP-level FIN is used only to terminate an individual subflow. So, FlexCP translates MPTCP ACK for MPTCP DATA_FIN to TCP ACK for TCP FIN and vice versa while it ACKs any FIN packets from a subflow without forwarding them. Also, FlexCP is responsible for closing all remaining subflows at MPTCP connection teardown. To avoid handling the loss of a FIN packet, we always have the endpoint send a FIN first so that the proxy piggybacks its own FIN on the ACK packet. This allows the proxy to receive a retransmitted FIN again even if the packet is lost.

SmartNIC-to-CPU packet forwarding. We use the `rte_flow` library of DPDK to send and receive a packet at a specific CPU core between host and SmartNIC. When a packet needs to be delivered to some CPU core, SmartNIC marks the core ID into the ToS field of the IP header, and sends it by the pre-configured `rte_flow` rule. Exploiting the ToS field can support up to 64 host CPU cores as we use 6 bits due to the constraint of the field. On the host side, the application creates two queues per CPU core – one for packet RX via RSS and the other for packet RX via `rte_flow` rules.

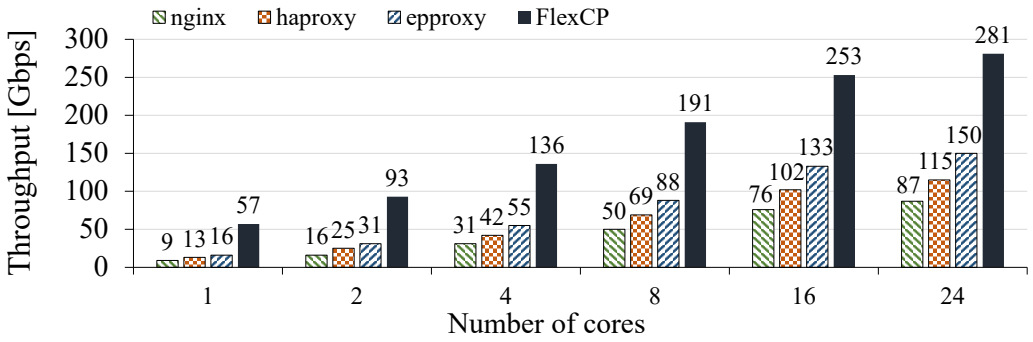


Fig. 7. Throughput comparison of MPTCP-TCP proxies over varying numbers of CPU cores

Zero-copy data forwarding. Packet forwarding without memory copy is an effective optimization in the splitless-connection model. Instead of copying the packet data to a separate buffer, FlexCP simply reuses the memory of a received packet to modify the header in place and then it sends out the updated packet. To efficiently insert or remove an MPTCP option into (or out of) the TCP header, FlexCP leverages NIC scatter-gather I/O –it enables hardware NIC to aggregate data chunks in different memory locations into a continuous packet for transmission. FlexCP exploits the same feature to split the data into multiple different packets in the zero-copy manner.

5 EVALUATION

We evaluate FlexCP to answer following questions. (1) Does FlexCP present better throughput and delay performance compared with existing proxy solutions? Does the throughput scale well over multiple CPU cores? (2) Does the flow-to-core affinity in FlexCP help improve the performance? (3) Does FlexCP support load balancing and dynamically adapt to varying network path conditions?

5.1 Experiment Setup

Our evaluation testbed consists of one proxy server, two backend servers, and four client servers. The proxy machine has an Intel Xeon Gold 6342 CPU @ 2.8GHz (24 cores) with 512GB DRAM. It is equipped with two dual-port NVIDIA BlueField-2 100GbE SmartNICs connected to a 100G switch (400 Gbps in total). Each server has a dual-port 100G ConnectX-6 NIC (200 Gbps in total per machine), and each client has a 100 Gbps ConnectX-5 NIC. All servers has an Intel Xeon Gold 6326 CPU @ 2.90GHz (16 cores) with 256GB DRAM. Client machines have one of Intel Xeon Gold 6342 CPU @ 2.90GHz (24 cores), Intel Xeon CPU E5-2683 v4 @ 2.10GHz (16 cores), or AMD EPYC 7282 (16 cores) with 128GB or 64GB DRAM. We confirm that clients and back-end servers are not the bottleneck in all experiments below.

We compare FlexCP with three baselines: `nginx` (version 1.25.1) and `haproxy` [20] (version 2.4.18, with `splice()`) for a MPTCP-TCP proxy, and `epproxy` [36], a TCP-TCP proxy on a scalable user-level TCP stack called mTCP [23]. We use `mptcpize` [38] to have `nginx`, `haproxy` and client applications create an MPTCP socket supported by the upstream kernel [7], running on Ubuntu 22.04 with Linux 6.1. Since mTCP does not support MPTCP sockets, `epproxy` is evaluated solely as a TCP-TCP proxy. Both FlexCP and `epproxy` run on DPDK 21.11 [22]. The backend servers run `nginx` as a Web server configured to accept only TCP connections.

5.2 Comparison of Throughputs

We evaluate the performance of FlexCP in terms of throughput and multicore scalability. In our setup, a client initiates an MPTCP connection with two subflows, which are then relayed through a

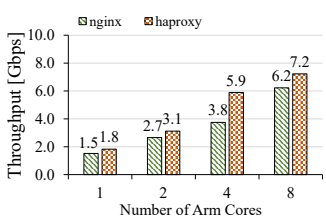


Fig. 8. Performance of baseline proxies on SmartNIC

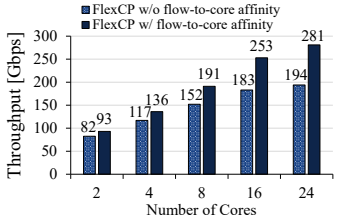


Fig. 9. Performance of FlexCP w/ and w/o flow-to-core affinity

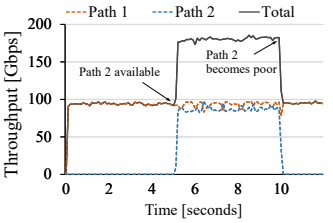


Fig. 10. FlexCP throughputs over dynamic network condition

single TCP connection by the proxy to a server. We have clients generate 256 concurrent MPTCP connections per each CPU core employed by the proxy, and every single connection emulates an active UE. Each MPTCP connection is persistent, and it repeatedly requests a 1 MB file at a time in plaintext HTTP/1.1. FlexCP uses flow-level traffic steering which forwards TCP traffic to one of the two MPTCP subflows at any given time while the Linux kernel uses the Lowest-RTT-First policy, which is the only policy currently supported by the upstream kernel [7].

Figure 7 compares the throughputs over an increasing number of CPU cores of the proxy. The performance of FlexCP scales well with multiple cores and it achieves 281 Gbps with 24 CPU cores. FlexCP outperforms the `nginx` proxy by 3.2× to 6.3× and `haproxy` by 2.4× to 4.4×, respectively. `haproxy` performs better than `nginx` as it employs the `splice()` system call to avoid expensive memory copy. `epproxy` outperforms `nginx` and `haproxy` as it runs on a user-level TCP stack on DPDK, but FlexCP performs better than `epproxy` by 1.8× to 3.5× thanks to its splitless-connection architecture. Furthermore, it is notable that FlexCP achieves throughputs comparable to those of E2E MPTCP shown in Figure 2, performing slightly better until 16 cores. This is because the E2E MPTCP case uses more CPU cycles to manage multiple subflows on the server-side whereas FlexCP works with more efficient TCP servers.

One might note that the performance comparison above is unfair as FlexCP uses both host CPU and Arm processors on SmartNICs while baselines use only host CPU. However, we find that the additional performance boost is minimal even if `nginx` or `haproxy` uses the Arm processors on SmartNICs. Figure 8 shows the throughputs of the baseline proxies over a different number of Arm cores, but none of them show more than 7.2 Gbps even if they use all eight Arm cores on SmartNIC. Note that we configure both `nginx` and `haproxy` to operate as a TCP-TCP proxy as the Linux kernel running on our Bluefield-2 SmartNIC does not currently support MPTCP [32]. In summary, we confirm that the performance benefit of FlexCP is still significant even if baselines use both host CPU and SmartNICs.

5.3 Benefit of Flow-to-core Affinity in FlexCP

We evaluate the benefit of SmartNIC-based flow-to-core affinity in FlexCP. We build a modified version of FlexCP that does not use SmartNIC for MPTCP packet steering. Instead, this version shares the MPTCP connection states with all CPU cores and accesses them via mutex locks, assuming that the packets in the same MPTCP connection can be forwarded to different cores.

Figure 9 compares the performance of FlexCP with and without flow-to-core affinity with the same evaluation setup as in Figure 7. We initially tried to configure each client to forward request packets over alternating subflow paths of the MPTCP connection, but the MPTCP stack on our Linux kernel does not support this – it tends to use only one of the subflow paths. Instead, we run the same experiment as before but configure FlexCP to do *packet-level* traffic steering so that the ACKs from the clients are delivered through different subflow paths. We observe that ensuring

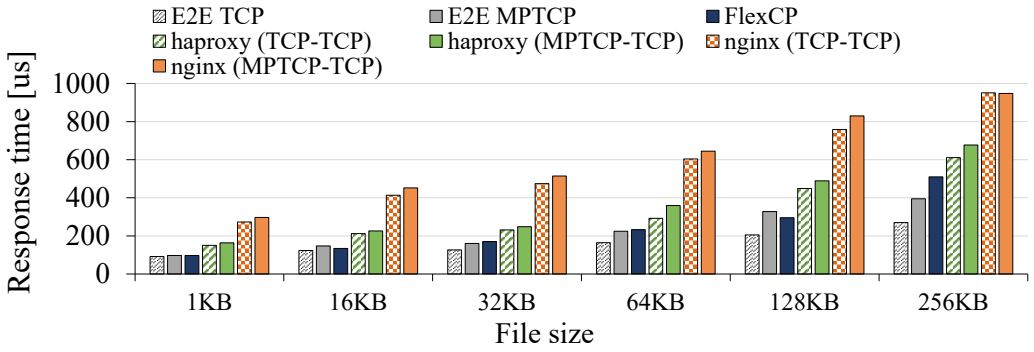


Fig. 11. Average response times for downloading files of a different size

flow-to-core affinity improves the throughput by up to 45% as the number of CPU cores increases. In fact, we find that even flow-level traffic steering shows a similar result where symmetric RSS does not always work as clients use multiple subflow paths. We conclude that the performance improvement is significant if FlexCP processes all packets in the same connection without locks.

5.4 Dynamic Switching of the Subflow Path

We evaluate if FlexCP can adapt to the dynamic change of the subflow path quality. We prepare two distinct physical paths from the clients to FlexCP where each path has 100 Gbps network bandwidth. We use 1024 MPTCP connections where each connection repeatedly downloads 1 MB file as before. FlexCP runs on 16 CPU cores, and we configure its scheduling policy to "load balancing", which selects the main subflow among available ones based on the number of main subflows per access [6]. Initially, all MPTCP connections use only one subflow path for about five seconds. Then, the second path becomes available, and each MPTCP connection adds a new subflow on it. Then, after five seconds, the second path exhibits a very high latency (i.e., we add 100ms of latency to the path using Linux `tc` [27]).

Figure 10 shows the aggregate download throughput as well as the throughput of each path over time. We observe that the aggregate throughput is near 100 Gbps for the first five seconds, and then both paths are fully utilized for the next five seconds. When the second path becomes available, we see that FlexCP steers the half of the flows to dynamically switch to the second subflow path during the content download. Then, FlexCP detects a very high delay on the second path, so it steers all flows to use the subflow on the first path again. This demonstrates that (1) flow-level traffic steering can utilize the aggregate bandwidth in multiple network paths, and that (2) FlexCP can dynamically steer the flows to avoid a network path whose quality becomes poor.

5.5 Comparison of Response Times

We evaluate the latency incurred by connection proxying. For this experiment, we measure the response time of downloading a single file in a single TCP/MPCP connection. We run each experiment for 10 times, and average the respond times. To understand the extra overhead by connection proxying, we also compare with end-to-end TCP/MPTCP connections.

Figure 11 shows the results over different file sizes. We make a few observations here. First, FlexCP incurs little extra latency comparable to the direct end-to-end connections. Surprisingly, it often shows better response times than E2E MPTCP connections – we suspect that the server-side MPTCP stack implementation on Linux is not fully optimized yet, thus proxying may provide better

[6]We plan to develop other schemes for load balancing in the future.

performance for small contents. Second, FlexCP outperforms other baseline proxies in response time – 1.7× to 3.1× for 1KB and 1.3× to 1.9× for 256KB. This confirms that packet-level protocol translation also reduces the proxying delay over the split-connection architecture.

6 DISCUSSION & LIMITATIONS

Our splitless-connection design primarily addresses the processing overhead on the proxy side, assuming that the network itself is not a performance bottleneck. However, there are certain scenarios where the split-connection proxying model can still offer benefits while our architecture imposes limitations. In this section, we describe the operating assumptions and architectural limitations of FlexCP, and discuss a few extensions that leverage the advantages of the split-connection model.

Operating assumptions and limitations. FlexCP is designed to bring performance benefits when the proxy becomes a bottleneck in relaying the content over two separate connections. We believe this is increasingly a common scenario in the near future as the bandwidth of a modern cellular network grows rapidly and the high-bandwidth Wi-Fi networks have become widespread. Another assumption is that clients are the consumer of the content, so the clients mostly download the content through multiple subflow paths rather than uploading their own content. So, the architecture of FlexCP is focused on efficiently handling the downstream traffic in the uncongested subflow paths.

However, FlexCP may expose architectural limitations when the above assumptions do not hold. One limitation is that FlexCP currently supports only flow-level traffic steering – it steers the downstream packets in the same flow into the same subflow path at any given time. Since FlexCP leaves all control logic to endpoints, it does not run congestion/flow control nor retransmit packets in the middle. Under this architecture, there is only one endpoint sender (i.e., a TCP server), so it cannot control the congestion in the multiple subflow paths concurrently. However, FlexCP monitors the quality of an individual subflow path at runtime, so it can avoid a subflow path whose quality becomes poor. We leave the impact of poor individual subflow(s) as future work. Also, the proxy can still enforce load balancing of the multiple subflow paths on a flow level so that a client can concurrently utilize the aggregate bandwidth if it employs multiple flows. In addition, FlexCP can balance the subflow paths *across* multiple TCP server connections.

FlexCP performs packet-level protocol translation, so one might concern that adding an MPTCP option to a full MTU-sized packet would exceed the MTU size limit. However, this is not a problem. As optimizations, FlexCP employs LRO and GRO to coalesce the consecutive content into a large packet and exploits TSO to segment the large packet into MTU-sized packets at NIC hardware. When using TSO, one can add an MPTCP option that contains the SSN-to-DSN mapping information. Then, all segmented packets would include the identical MPTCP option for the entire range, and the receiver can translate SSNs into DSNs with the option. One corner case that rarely occurs is that the original TCP packet header has a very long SACK option, so adding an MPTCP option makes the header size exceed the maximum of 64 bytes. In this case, FlexCP drops the SACK option from the header to fit the header size into the limit, which is suboptimal but fine in terms of correctness.

Fallback to split-connection mode. In scenarios where the network-side, particularly the access network, becomes a bottleneck, split-connection proxying can offer improved performance by downloading data from the servers and storing it in the proxy-side buffers. This allows the proxy to perform data retransmissions directly, mitigating the impact of network congestion, thus enhancing overall performance. Since several prior works have already addressed this issue based on the split-connection architecture, our current design does not specifically handle this scenario, but it is possible to fall back to a split-connection mode depending on the network path quality. For

instance, base stations could inform the ATSSS servers to switch to a split-connection mode during peak times when the access network experiences significant performance degradation. We leave an exploration of this direction to future work.

Corner cases in SN translation. MPTCP allows retransmitting the data via another subflow in addition to the original subflow. Exploiting this feature, when the proxy detects a failure of one subflow, it can close it and select another subflow for retransmission. Then the proxy should modify the DSM infomation as well as the Equiv-SSN field in the D-tree, which might break the rule that a D-tree node with a larger DSN range should have a larger SSN range as well (note that a newly retransmitted packet with a smaller DSN range needs to be sent with the largest SSN range). However, our design still works even in this case because (1) we keep the Equiv-SSN field for this case, and (2) the MPTCP side will send ACKs with DATA_ACKs regardless of the subflow ID, that are simple to translate. So, even though the TCP side retransmits a packet, we can look up the D-tree and successfully find the subflow for forwarding. Another corner case is when the proxy wants to transmit every packet redundantly through multiple subflow paths. While we have not implemented the forwarding mode in FlexCP yet, we believe our D-tree structure can support it without any modification. However, our implementation currently does not handle retranmission from a different MPTCP flow to a TCP connection, which is our future work.

7 RELATED WORK

We discuss some of the key related works of the MPTCP protocol and L4/L7 protocol proxies.

MPTCP protocol. Unlike standard TCP, MPTCP faces two algorithmic challenges: TCP-friendliness and packet scheduling. The traditional congestion control algorithms allow MPTCP connections to consume more bandwidth than regular TCP connections. This is because each MPTCP connection comprises multiple subflows, with each subflow acting as an independent TCP flow. Consequently, there is an unfairness issue in terms of network bandwidth allocation. To address this problem, several algorithms have been proposed, known as *coupled* congestion control algorithms. The fundamental idea behind these algorithms is to couple the information from all subflows when determining the congestion window size for each subflow. This enables an MPTCP connection to shift as much traffic as possible away from its most congested paths, which are likely to be occupied by a larger number of regular TCP flows [47]. One such algorithm is the Linked-Increases Algorithm (LIA) [47], which provides stability while fulfilling this design objective. The Opportunistic Linked-Increases Algorithm (OLIA) [25], however, claims that LIA can be unfair towards regular TCP flows, particularly when multiple MPTCP subflows and TCP flows coexist on the same path. In response, OLIA introduces a Pareto-optimal algorithm that improves fairness even in such scenarios. Additionally, the Balanced Linked Algorithm (Balia) [41] demonstrates that there is an inevitable trade-off between TCP-friendliess and responsiveness to network dynamics. Balia strikes a suitable balance between these two aspects.

While these coupled algorithms determine the sending rate of each subflow, the MPTCP packet scheduler selects the most suitable subflow for transmission based on a specified policy, for example, Lowest-RTT-First, as discussed in Section 2.1. In [40], the round-robin (RR) policy, where each data packet is assigned one by one across available subflows, is also evaluated and compared with the default policy. The evaluation reveals that the RR policy is generally inefficient due to its lack of consideration for diverse path characteristics. Furthermore, it is reported that assigning data packets to slower or lossy paths often results in a buffer blocking issue on the receiver-side when the MPTCP-level buffer becomes full as a result of out-of-order packet arrivals [47]. This prevents other non-congested subflows from transmitting further, as the receiver reports a zero advertisement window through ACKs, ultimately leading to performance degradation.

The BLEST [13] and STMS [45] schedulers take the buffer-blocking condition into account in their designs, so that network bandwidth of non-congested paths can be fully utilized. Lastly, the Redundant [29] scheduler duplicates each data packet across all available subflows to address end-to-end latency optimization at the expense of increased bandwidth consumption due to packet duplication. As discussed in Section 6, these algorithms are applicable to MPTCP-TCP proxy models, making them complementary to our work.

L4/L7 protocol proxies. L7 protocol proxying is a widely used technique for load balancing among servers, leveraging application-level protocols like HTTPS. Since data relaying at the application-layer incurs significant processing overhead, including data copies between user and kernel space, the Linux community has introduced `splice()` that moves data directly from one socket buffer to another inside the kernel since Linux 2.6.17 [6]. Several L4 load balancers then have been proposed that perform packet-level header translation at the transport layer, akin to the `splice()` technique [12, 18, 34, 37]. More recently, this approach has been extended to L7 proxies in scenarios where the two endpoints are fixed [17, 36]. Zeng *et.al.* [49] ensures MPTCP-level per-connection-consistency, by chaining load balancers so that any subflow eventually reaches the correct backend, like Beamer [37].

We note that the majority of previous works for MPTCP-TCP proxy rely on the split-connection architecture. [19] is the only work that explores splitless-connection proxying as far as we know. However, it mainly focuses on the mobility feature of MPTCP as well as MPTCP-side connection handover rather than scalable sequence number translation with tens of thousands of subflows. In contrast, FlexCP ensures efficient packet forwarding by leveraging an efficient data structure for scalable DSM management, which promises high performance at low latency.

8 CONCLUSION

MPTCP-TCP proxying is a practical approach to bridging the benefit of multiple paths to multi-homed Internet users. This paper presents FlexCP, a scalable protocol-translation proxy design for commodity servers. We have shown that one can proxy connections of heterogeneous protocols by simple sequence number translation on a packet level. For scalable translation, FlexCP leverages D-tree, an augmented RB-tree search tree that allows fast lookup, insertion, and deletion of DSM nodes. Also, FlexCP exploits SmartNIC to ensure MPTCP/TCP connection-to-core affinity for lockless packet processing on multicore systems. We observe that our design choices lead to high performance – FlexCP achieves 281 Gbps proxying performance on a machine with a single CPU, which outperforms the nginx proxy by up to 6.3x in throughput, and its response time is almost comparable to that of a direct end-to-end MPTCP/TCP connection. The project homepage of FlexCP is https://flexcp.kaist.edu/.

ACKNOWLEDGMENTS

We appreciate insight feedback and suggestions by CoNEXT 2023 reviewers. This work is in part supported by Samsung Research under [High Performance 6G Acceleration Techniques] and ICT Research and Development Program of MSIT/IITP, Korea, under[2018-0-0693,Development of an ultra low-latency user-level transfer protocol]. This work is also supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1A2C1011090).

REFERENCES

[1] 3GPP. 2023. TS 23.501 System Architecture for the 5G System (5GS). https://www.3gpp.org/dynareport/23501.html. Last Accessed: 2023-10-26.

[2] 3GPP. 2023. TS 24.193 5G; 5G System; Access Traffic Steering, Switching and Splitting (ATSSS); Stage 3. https://www.3gpp.org/dynareport/24193.html. Last Accessed: 2023-10-26.

[3] Mamta Agiwal, Hyeyeon Kwon, Seungkeun Park, and Hu Jin. 2021. A Survey on 4G-5G Dual Connectivity: Road to 5G Implementation. *IEEE Access* 9 (January 2021).

[4] Apple. 2022. Use Multipath TCP to Create Backup Connections for iOS. https://support.apple.com/en-us/HT201373. Last Accessed: 2023-10-26.

[5] Florian Aschenbrenner, Tanya Shreedhar, Oliver Gasser, Nitinder Mohan, and Jörg Ott. 2021. From Single Lane to Highways: Analyzing the Adoption of Multipath TCP in the Internet. In *Proceedings of the IFIP Networking Conference (Networking)*.

[6] Jens Axboe and Larry McVoy. 2006. splice - Splice Data to/from a Pipe. https://manpages.ubuntu.com/manpages/lunar/en/man2/splice.2.html. Last Accessed: 2023-10-26.

[7] Matthieu Baerts. 2020. MPTCP stack on upstream kernel. https://github.com/multipath-tcp/mptcp_net-next/wiki. Last Accessed: 2023-10-26.

[8] Scott Bicheno. 2019. Korea Telecom and Tessares claim 5G Low Latency Multi-Radio Access Technology First. https://telecoms.com/499409/korea-telecom-and-tessares-claim-5g-low-latency-multi-radio-access-technology-first/. Last Accessed: 2023-10-26.

[9] Olivier Bonaventure, Mohamed Boucadair, Sri Gundavelli, SungHoon Seo, and Benjamin Hesmans. 2020. 0-RTT TCP Convert Protocol. RFC 8803, IETF.

[10] Massimo Condoluci, Stephen H Johnson, Vicknesan Ayadurai, Maria A Lema, Maria A Cuevas, Mischa Dohler, and Toktam Mahmoodi. 2019. Fixed-mobile Convergence in the 5G Era: From Hybrid Access to Converged Core. *IEEE Network* 33, 2 (February 2019).

[11] Gregory Detal, Matthieu Baerts, Quentin De Coninck, and Olivier Bonaventure. 2015. Android Package to Use MPTCP Feature of Linux Kernel MPTCP Project. https://github.com/MPTCP-smartphone-thesis/MultipathControl. Last Accessed: 2023-10-26.

[12] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[13] Simone Ferlin, Ozgu Alay, Olivier Mehani, and Roksana Boreli. 2016. BLEST: Blocking Estimation-based MPTCP Scheduler for Heterogeneous Networks. In *IFIP Networking*.

[14] Rostand A. K. Fezeu, Eman Ramadan, Wei Ye, Benjamin Minneci, Jack Xie, Arvind Narayanan, Ahmad Hassan, Feng Qian, Zhi-Li Zhang, Jaideep Chandrashekar, and Myungjin Lee. 2023. An In-Depth Measurement Analysis of 5G mmWave PHY Latency and Its Impact on End-to-End Delay. In *Passive and Active Measurement Conference*.

[15] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684, IETF.

[16] Broadband Forum. 2018. 5G Fixed-Mobile Convergence. https://www.broadband-forum.org/download/MR-427.pdf. Last Accessed: 2023-10-26.

[17] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[18] Rohan Gandhi, Hongqiang Harry Liu, Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[19] Georg Hampel, Anil Rana, and Thierry Klein. 2013. Seamless TCP Mobility Using Lightweight MPTCP Proxy. In *Proceedings of ACM International Symposium on Mobility Management and Wireless Access (MobiWac)*. Association for Computing Machinery.

[20] HAProxy. 2023. The Reliable, High Performance TCP/HTTP Load Balancer. https://www.haproxy.org/. Last Accessed: 2023-10-26.

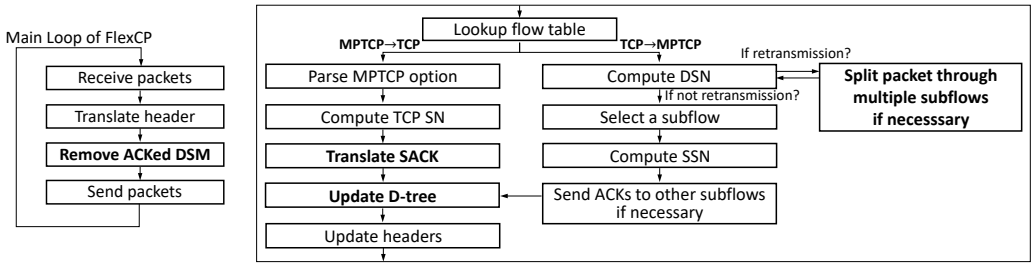
[21] Intel. 2017. Intel Ethernet Flow Director. https://www.intel.com/content/www/us/en/developer/articles/training/setting-up-intel-ethernet-flow-director.html. Last Accessed: 2023-10-26.

[22] Intel. 2020. DPDK Release 21.11. https://doc.dpdk.org/guides/rel_notes/release_21_11.html. Last Accessed: 2023-10-26.

[23] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[24] Nicolas Keukeleire, Benjamin Hesmans, and Olivier Bonaventure. 2020. Increasing Broadband Reach with Hybrid Access Networks. *IEEE Communications Standards Magazine* 4, 1.
[25] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. 2013. MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Transactions on Networking* 21 (August 2013).
[26] Jonghoe Koo, Juheon Yi, Joongheon Kim, Mohammad Ashraful Hoque, and Sunghyun Choi. 2018. Seamless Dynamic Adaptive Streaming in LTE/Wi-Fi Integrated Network under Smartphone Resource Constraints. *IEEE Transactions on Mobile Computing* 18 (August 2018).
[27] Alexey N. Kuznetsov and Bert Hubert. 2001. tc - Show / Manipulate Traffic Control Settings. https://manpages.ubuntu.com/manpages/lunar/man8/tc.8.html. Last Accessed: 2023-10-26.
[28] Kyunghan Lee, Joohyun Lee, Yung Yi, Injong Rhee, and Song Chong. 2012. Mobile Data Offloading: How Much can WiFi Deliver? *IEEE/ACM Transactions on networking* 21 (November 2012).
[29] Igor Lopez, Marina Aguado, Christian Pinedo, and Eduardo Jacob. 2015. SCADA Systems in the Railway Domain: Enhancing Reliability through Redundant MultipathTCP. In *IEEE ITSC*.
[30] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. 2021. Democratizing Cellular Access with CellBricks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*.
[31] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. 1996. TCP Selective Acknowledgment Options. RFC 2018, IETF.
[32] Mellanox. 2016. Linux Kernel to Support Mellanox BlueField SoCs. https://github.com/Mellanox/bluefield-linux. Last Accessed: 2023-10-26.
[33] Mellanox. 2020. Mellanox ASAP2 Accelerated Switching and Packet Processing. https://network.nvidia.com/files/doc-2020/sb-asap2.pdf. Last Accessed: 2023-10-26.
[34] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[35] YoungGyoun Moon, Donghwi Kim, Younghwan Go, Yeongjin Kim, Yung Yi, Song Chong, and KyoungSoo Park. 2015. Practicalizing Delay-tolerant Mobile Apps with Cedos. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[36] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[37] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[38] Ossama Othman. 2023. mptcpize - Enabling Mptcp on Existing Services. https://manpages.ubuntu.com/manpages/lunar/man8/mptcpize.8.html. Last Accessed: 2023-10-26.
[39] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. 2012. Exploring Mobile/WiFi Handover with Multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular networks: operations, challenges, and future design*.
[40] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. 2014. Experimental Evaluation of Multipath TCP Schedulers. In *Proceedings of ACM SIGCOMM Workshop on Capacity Sharing*.
[41] Qiuyu Peng, Anwar Walid, Jaehyun Hwang, and Steven H. Low. 2016. Multipath TCP: Analysis, Design, and Implementation. *IEEE/ACM Transactions on Networking* 24 (December 2016).
[42] Shiva Raj Pokhrel, Neeraj Kumar, and Anwar Walid. 2021. Towards Ultra Reliable Low Latency Multipath TCP for Connected Autonomous Vehicles. *IEEE Transactions on Vehicular Technology* 70 (June 2021).
[43] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. 2011. Improving Datacenter Performance and Robustness with Multipath TCP. *ACM SIGCOMM Computer Communication Review* 41 (August 2011).
[44] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[45] Hang Shi, Yong Cui, Xin Wang, Yuming Hu, Minglong Dai, Fanzhao Wang, and Kai Zheng. 2018. STMS: Improving MPTCP Throughput under Heterogeneous Networks. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.
[46] Tessares. 2021. Wi-Fi Cellular Convergence with Overlappng Handover. https://www.tessares.net/cellular-wi-fi-convergence. Last Accessed: 2023-10-26.
[47] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[48] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. 2013. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. (June 2013).
[49] Yijing Zeng, Milind Buddhikot, and Suman Banerjee. 2021. All Roads Lead to Rome: An MPTCP-Aware Layer-4 Load Balancer. In *Proceedings of the IFIP Networking Conference (Networking)*.



(a) Main loop of FlexCP (b) Detailed steps for "Translate header"

Fig. 12. Packet processing steps for an established connection in FlexCP. Steps requiring D-tree access are written in boldface.

A PACKET PROCESSING IN FLEXCP

Figure 12 shows how FlexCP forwards incoming packets for an established connection. The main loop of FlexCP is conceptually very simple as it operates by reacting to incoming packets and it does not need to maintain any timers. The key operation is SN/ACK translation in the header as explained in Section 3.2. We note that MPTCP proxies can be collocated with UPF in 5G networks, and session-level operations can be chained together in prior to FlexCP operations. Our implementation assumes that MPTCP path scheduler (or control plane of ATSSS proxy) runs asynchronously out of the FlexCP main loop.

Received July 2023; accepted October 2023