MTP: Transport for In-Network Computing

Tao Ji
UT Austin

Rohan Vardekar
University of Illinois Chicago

Balajee Vamanan
University of Illinois Chicago

Brent E. Stephens
Google and University of Utah

Aditya Akella
UT Austin

Abstract

In-network computing (INC) is being increasingly adopted to accelerate applications by offloading part of the applications' computation to network devices. Such application-specific (L7) offloads have several attributes that the transport protocol must work with—they may mutate, intercept, reorder and delay application messages that span multiple packets. At the same time the transport must also work with the buffering and computation constraints of network devices hosting the L7 offloads. Existing transports and alternative approaches fall short in these regards. Therefore, we present MTP, the first transport to natively support INC. MTP is built around two major components: 1) a novel message-oriented reliability protocol and 2) a resource-specific congestion control framework. We implement a full-fledged prototype of MTP based on DPDK. We show the efficacy of MTP in a testbed with a real INC application as well as with comprehensive microbenchmarks and large-scale simulations.

1 Introduction

Datacenters today are experiencing ever-increasing demand for computing with the emerging of large-scale distributed applications and intensive workloads such as ML/AI and VR/AR. These applications run heavy computational logic and rely on infrastructure services (e.g., for caching and load-balancing) that themselves impose non-trivial computational needs. In-network computing (INC) has emerged as a promising approach to meet the resulting compute demand by leveraging specialized in-network hardware for offloading. With INC, application-specific and infrastructure processing functions—which we together refer to as L7 functions—can be run atop network devices. This can provide vastly improved application throughput and latency compared to approaches that rely on conventional server architectures [65].

We have seen a blossoming of ideas for offloaded L7 functions (or "L7 offloads"). Examples include key-value caches [35, 46, 75], file and storage systems [41, 48, 59], machine learning (ML) tasks [45, 55, 69], transaction and database systems [32, 34, 70], consensus [47], as well as load balancing [17, 81]. Alongside, we have seen a growing number of devices and platforms that can support INC, including programmable switches [1–3, 12, 40, 78, 79], and SmartNICs [10, 53, 54, 64]. While the growing number of showcases for INC is promising, we argue that there are fundamental problems that need to be addressed before INC can be utilized in a systematic and effective manner to accelerate datacenter computation. This paper focuses on one such problem, namely the design of an INC-compatible transport protocol.

Despite the wide diversity of INC possibilities and platforms, we argue that all of them need in common a transport protocol with two properties:

(1) The transport's basic functions—reliability and congestion control—should work in the face of key attributes of INC, namely that the L7 offloads can induce mutation, intercept, reordering and arbitrary delays upon application messages. For example, an HTTP load balancer can insert cookies to an HTTP request [17], and an in-network aggregator can delay producing the result until the straggler provides the input [78]. We detail all such attributes in §2.1.

(2) The transport should not impose additional transport-specific resource needs on INC hardware (e.g., needing significant buffering or state maintenance). Switches and SmartNICs that host INC have limited memory and compute power.

Most datacenters today use either TCP variants or RDMA as their transport [29, 72, 82], and new experimental transports have emerged, such as the receiver-driven Homa protocol [61]. Unfortunately, none of these transport protocols can accommodate the attributes of INC. When used alongside INC, they result in either broken reliability semantics, poor and ineffective congestion control, or both (§2.2). Alternative approaches either have prohibitive compute and memory overheads or are ad-hoc, eschewing reliability and/or congestion control while only supporting a subset of the INC attributes (§2.3), and hence are not widely applicable.

This paper presents MTP (Message Transport Protocol), the first INC-compatible transport protocol. It makes key design choices that elevate L7 offloads—how they function and how

they are used in datacenter networks—to first-class entities from a transport perspective. MTP works on *messages* as this aligns with the processing semantics of L7 offloads—all key decisions in MTP, be they for loss detection/recovery, congestion control or path selection, operate at the message granularity. To handle message mutation, intercept and reordering, MTP decouples key transport actions, e.g., loss detection and recovery, from the packet or byte sequence numbers (§4).

MTP recognizes and operates on *pathlets*, groups of offloads that share fate. For example, a pathlet can be either a single instance of an offload or multiple replicas of an offload that a NIC load balances across. MTP elicits *per-pathlet feedback* both for MTP's reliability protocol to set appropriate timeouts when messages are delayed by pathlets rather than dropped (§5) and for congestion control to take per-message decisions on when to transmit and which pathlets to use. This controls the load on pathlets, avoiding the overloaded ones and causing load imbalance (§6).

Additionally, using MTP has low overheads: pathlets need not maintain MTP-specific state and can easily leverage MTP primitives by generating appropriate ACK packets (§4–6). The endpoint-maintained state is constant and proportional to (an upper-bounded) number of in-flight messages and pathlets in use (Appendix B). The MTP header is only 20 B plus optionally a 5 B pathlet congestion feedback (Appendix A), and the ACK primitives have low bandwidth and endpoint processing overheads (§8.2.3).

To evaluate MTP, we implement a full-fledged MTP prototype based on DPDK [23] and perform end-to-end benchmarks by running on top of MTP NetCache [35], a real-world INC application. We show that MTP improves the throughput by over 15% (§8.1). We conduct comprehensive microbenchmarks (§8.2) showing MTP's effective loss recovery and congestion control in the face of INC. We also simulate MTP's operation at large scales (§8.3) and find that MTP achieves 65% less tail latency and can sustain 10–20% higher loads compared to TCP under heavy message reordering.

2 Motivation

In this section, we argue that a novel transport for INC is warranted. For that, we capture the essential message operations that INC performs (§2.1), describe how existing transports are not compatible with these operations (§2.2), discuss how the ways in which INC is made to work with transports today are not widely applicable (§2.3), and extract the requirements of a novel transport (§2.4).

2.1 On-Path Message Processing

INC enables applications to offload portions of their (L7) logic to NICs or switches. Traditional offloads, such as network address translation and transmit segmentation offload, have dedicated network- or transport-level logic and process the corresponding protocol headers in packets. In contrast, L7 offloads process application-level (L7) messages, which are the payloads that span one or more network packets. L7 offloads have more diverse functions. Examples include various key-value caches [35, 46, 49], intrusion detection/prevention [80], L7 load balancing [17, 81], transaction and database acceleration [32, 34, 70], as well as in-network aggregation (INA) for machine learning jobs [45, 55, 69, 78].

These offloads are "on-path": located on the communication path between end hosts, and multiple offloads that perform different processing (e.g., load balancing and intrusion detection) or different parts of the same processing (e.g., two levels of INA [45]) can form a "chain" on the path.

However, the operations that these offloads perform on L7 messages can often interact poorly with the underlying transport, potentially causing low performance and even breaking the transport protocol. We summarize the operations below and discuss their impact on transport in the next section.

Mutation. An offload can change the L7 message payload. At times this can result in changes in the message size and the number of packets that the message spans. For example, an HTTP load balancer [17] can inject cookies into HTTP responses, which makes the response messages longer. The in-network aggregator of SwitchML [69] replaces the per-worker gradients in the RDMA message with aggregated gradients.

Intercept. In some scenarios, an offload can explicitly drop a message so that it does not reach the transport's receiving endpoint. Some RPC load balancers, for example, can proactively drop the requests that will anyway violate the service-level objectives [42].

Reordering. An offload can reorder messages between the same sender and receiver for two reasons. First, the offload explicitly does so as part of its logic. Transaction triaging, for example, reorders messages to batch similar transactions, which improves server performance [34]. Second, many hardware architectures support running replicated instances of the same offload logic in parallel to improve performance [53, 54, 78]. Messages that take different amounts of processing time at various instances can finish in a different order.

Delaying. An offload can impose delays on a message in various cases, and such delays can be long and unpredictable. (1) Complex INC logic or wimpy hardware: FPGA-based intrusion detection, for instance, can incur a tail processing delay of ∼4x median for infrequent slow-path processing [80]. Some wimpy SmartNICs can even incur ∼80μs tail delay (∼2x average) on packet steering [54]. These delays are of the same order of magnitude as the network fabric delay (10s of μs [43]). Such offloads themselves are often bottlenecks and when experiencing high load may incur increased queuing delay. (2) In some cases, the processing of a message cannot complete if another message is delayed. E.g., in INA, gradients cannot be completely aggregated if a straggler worker delays its part, and mitigating a straggler can take 10s of ms, which is much higher than the usual aggregation [78] and network fabric delay.

2.2 Today's Transports Are Incompatible

We discuss three transports, i.e., TCP (including its variants such as QUIC [44]), RDMA [29, 82], and Homa [61]. All of them provide reliability and congestion control, which are essential transport-layer functions. TCP variants and RDMA are arguably the most widely adopted transports in production datacenters. Homa is an emerging receiver-driven transport providing message transfer semantics with efforts to deploy it in production [31, 63]. Unfortunately, these transports are largely incompatible with INC operations which can have a fundamental impact on the correctness and efficacy of the transports' reliability and congestion control.

The TCP variants and RDMA do not work in the face of mutation, intercept, or reordering. Consider these transports' reliable delivery: they both use a per-connection continuous byte/packet sequence number space, and all the bytes/packets transmitted by the sender must be acknowledged by the receiver. When an offload mutates a multi-packet message in a way that changes its length or intercepts it, the byte and/or packet counts that reach and are ACK'd by the receiver also change. In that case, the sender cannot correctly decide if a missing byte/packet needs retransmission, and can be confused if ACK'd bytes/packets are more than what it has transmitted. Also, reordering can cause severe out-of-order packet arrivals resulting in discontinuous received sequence, which both protocols can interpret as loss and trigger wasteful retransmission as well as reduction in congestion window.

Homa cannot properly perform loss recovery with mutation. Homa's receiver detects missing bytes with a timeout and asks the sender to retransmit the missing part. However, if the message is mutated to a different length, the range of bytes in the mutated message that the receiver asks for is likely not the same bytes in the original message. Note that Homa could work with intercept and reordering because the transmission of each message is independently controlled, and there is no cross-message state such as a sequence number that can be disrupted by dropping or reordering messages.

Finally, delaying can also disrupt congestion control in fundamental ways in all three protocols. TCP and RDMA rely on some end-to-end measurements such as ECN [8, 82] and RTT [43, 60] to quickly and accurately detect congestion. These measurements traverse the forward path and are eventually reflected back to the sender by the receiver. When messages can be delayed at an offload for unpredictably long, the measurement might not arrive at the sender in a timely manner, which prevents the sender from quickly reacting to congestion. To achieve line rate and low queuing, Homa's receiver tries to keep 1 BDP (bandwidth-delay product) of credits in flight. However, this assumes that both network RTT and bandwidth is known and constant. With INC, message mutation that causes changes in length, below-line rate bottleneck offloads and unpredictable delays can make it extremely difficult to accurately control the outstanding credits.

2.3 How INC Works With Transports Today

Some existing INC works have made simplifying assumptions or workarounds so that transports can correctly function with a specific offload. However, such approaches have important limitations.

One approach to be compatible with INC operations is for an offload to "terminate" the transport, running both the receiver- and sender-side protocol logic [28]. However, transport stacks need significant hardware resources on the offload for transport processing and message buffering. The hardware limitations of some INC platforms like switches preclude running transports altogether [55, 69], while the limitations of others like SmartNICs may limit scalablity and require careful parallel decomposition and many cores for performance [71].

Furthermore, the simplifying assumptions made by prior work are also usually unrealistic. For example, prior work has assumed that an RPC's length never exceeds one MTU (maximum transmission unit) and hence can always be carried by just one packet [35,46,49,53,64]. Other work has assumed that a new connection can be established for each RPC [81]. However, these don't generally hold: A significant proportion of RPCs in datacenters are indeed larger than the MTU [8,61], and RPCs typically reuse connections because connection establishment incurs overhead [4].

Some works have proposed workarounds, which usually cover only the subset of message operations required by the specific offload and are not generalizable. As a result, these solutions require case-by-case human efforts to apply to other use cases. For example, NetReduce [55] accommodates mutation in a way that is transparent to TCP and RDMA, but the message size must not change. As such, this approach is not compatible with an HTTP load balancer that adds cookies. Additionally, this approach does not support intercept or reordering. SwitchML [69] and ATP [45] build their own transport functions such as reliable delivery based on UDP, but neither work in the face of intercept. The transaction triaging offload [34] supports intercept and reordering but does not work amidst mutation of the message (transaction) body.

Others have relied on specific hardware architectures. The HTTP load balancer [17], for example, accommodates mutation that changes the message length based on NICs that have general-purpose cores. However, it must maintain per-connection state that correctly maps between the altered and original byte sequences, and it must buffer the mutated messages for retransmission. Maintaining such per-connection state can be expensive in hardware, and buffering payload might not be supported by some hardware architectures.

2.4 Design Requirements

We argue for a novel transport that is widely applicable to INC. Such a transport must meet these requirements:

- Provide correct reliable delivery in all scenarios where the packets or bytes are altered in the network as a result

of message mutation, intercept, or reordering;

- Provide effective congestion control for all potential bottlenecks in the network, including the network fabric and slow offloads, despite the delaying of messages.
- Support all kinds of INC hardware without assuming that special transport-specific state or buffering can be maintained by the offload.

We present MTP (Message Transport Protocol), the first transport designed for INC, satisfying all of these requirements.

3 Pathlets

Before describing MTP, we provide a brief overview of pathlets, which is a key building block. Then, we discuss the infrastructure support needed to allow MTP to leverage pathlets and our assumptions on how pathlets work.

3.1 Pathlets and Grouping

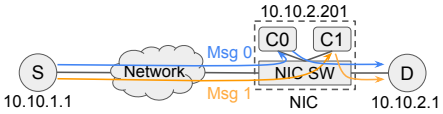
We observe that on the path of a message, between the transport sender and receiver, there can be multiple offloads, potentially with different types of computation. For example, an L7 load balancer [17, 81] can be followed by intrusion detection [80], if the requests can come from untrusted senders. In a two-level INA deployment, a message can go through two aggregators on different switches [45].

We refer to each such offload instance on the path as a pathlet [1]. Pathlets that perform the same computation are said to have the same pathlet type. An ordered chain of pathlet types defines the computation that the message goes through in the network.

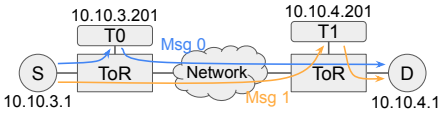
Instances of a pathlet type can be arranged in two forms. First, as shown in Figure 1b, offload instances can be at different network locations; in this case, as we describe below, MTP allows a sender to select which specific pathlet instance to use. Second, as shown in Figure 1a, the instances are co-located on the same NIC or switch, and an on-NIC/switch scheduler locally balances their loads [53]. Thus, these should not be viewed as individual pathlets as the scheduler hides internal load imbalance or partial failure. MTP groups the latter such instances together and views them as one pathlet.[2]

3.2 Pathlet Discovery and Routing

For the MTP transport to work with pathlets, the network must provide two services: (1) Pathlet discovery: The datacenter network (DCN) should determine the chain of pathlet types to be used for messages towards a given receiver, and provide for each type the specific pathlet instances for the transport to use. (2) Pathlet routing: The network should route the packets of a message to the transport-selected pathlets in the same order. We now describe how these services can be provided in an MTP-enabled DCN.



(a) Locally load-balanced offloads C0 and C1.



(b) Non-locally deployed offloads T0 and T1.

Figure 1: **Possible deployment models for offload instances that perform the same computation.**

Discovery. We assume that the applications that leverage INC capabilities follow the client-server model. When a client attempts to reach a service, it first queries for the server(s) with a service discovery protocol (SDP), such as [5, 6]. The SDP provides, in addition to server addresses, a list of pathlet types that client messages should go through toward the service. The client then queries a pathlet registry for the available instances of each pathlet type and the address of each instance. The replied pathlet information for the service is cached in the client host hypervisor or OS for fast retrieval.

This pathlet registry is similar to a service registry that would back a SDP today. It can be scaled up with standard sharding techniques that are used for service registries, such as employing a distributed key-value store [68, 76].

Routing. To reach the pathlets, we employ a form of source routing. When a client connects to a service, it configures MTP with the server and pathlet addresses that it queries from the SDP. When sending messages, MTP encodes the list of pathlet addresses it selects in the packet header, and the network forwards the packets through the hops with given addresses in order, and eventually to the server. This can be achieved with IPv6 segment routing [22] on commodity off-the-shelf network devices. We assume that each pathlet has an IPv6 address.

3.3 Pathlet Processing

We observe that there can be two ways in which the pathlet gathers data to process: 1) *full buffering*, where the pathlet buffers messages in full, and the processing can only be done when the whole message has arrived at the pathlet; and 2) *streaming*, where the pathlet starts processing with partial messages, and can receive and transmit packets of the message while processing. Full buffering can utilize *cut-through* forwarding, so the biggest difference between streaming and full buffering is in the buffering requirements.

We design MTP to cater for full buffering, which we argue is the most common case, for two reasons. First, applica-

[1] While inspired by prior art on "pathlet routing" [26] where a pathlet is a link or set of links, we use the term "pathlet" differently.

[2] A datacenter operator can conservatively decide to have just one instance for each pathlet type as default. However, if the operator chooses to expose multiple pathlets, MTP can offer improved performance.

tions can often break down large data into packet-sized messages for offload processing. For example, SwitchML [69] and ATP [45] encapsulate partial gradients in single-packet messages to fit in the programmable switch, each message with an L7 header as application context. Also, for many applications in datacenters, messages are small, with over 90% messages each fitting in one Ethernet jumbo frame, and many offloads also assume single-packet messages [35, 46, 51, 81].

Second, it is usually an application requirement or choice to process larger messages in whole, in which case the pathlet should already have the capability to achieve full buffering of multi-packet messages. For instance, NetReduce [55] works with large messages that contain ML gradients by relying on the transport to chunk the message, and the FPGA-based aggregator can reassemble the message based on transport sequence numbers.

We note that some applications employ limited-capability offload hardware, such as Tofino [3], where it is hard to perform processing that involves maintaining cross-packet (message) state. MTP is still applicable: such offloads can leverage MTP's reliability and congestion control primitives by simply generating appropriate ACK packets (detailed later) upon receiving a "single-packet message".

We discuss how MTP can be used as a building block to support streaming applications in §9.

4 Overview

We now present MTP, starting with an overview of its architectural components and a basic end-to-end workflow.

4.1 MTP Architecture

MTP consists of two major components: (1) a message-oriented reliability protocol that natively accommodates message mutation, intercept and reordering; and (2) a multi-pathlet congestion control framework that enables congestion control algorithms to operate effectively in the face of pathlets causing long and unpredictable delays. Neither component requires additional state at pathlets. As such, MTP meets all of the requirements in §2.4.

To design such a reliability protocol, we make a key observation about why existing transports fall short (§2.2): The receiver cannot tell whether a missing place in the received packet or byte sequence is because of loss or intercept/reordering; nor can it, in case of mutation, provide the before-mutation sequence number for the sender to retransmit. Our design choice is therefore to rely on the sender to detect losses with *per-message retransmission timeouts (RTOs)*: The receiver, upon the arrival of all packets of the message, sends an end-to-end (E2E) ACK, which disarms the timer. Otherwise the timer triggers retransmission of the whole message. We discuss the challenges that come with this design in §5.1.

To enable effective congestion control (CC) under long and unpredictable delay caused by INC, our framework provides two primitives. (1) Per-pathlet early congestion feedback: A pathlet has the choice of communicating its own congestion state, as well as reflecting any congestion signal along the path (such as ECN for link congestion), directly back to the sender in the form of a *dedicated packet*, that does not incur additional delay from the offload processing. (2) Per-message pathlet decision: If multiple same-type pathlets are available, the congestion control can *decide the instance to use on a per-message basis*. This allows quickly steering messages away from the congestion point. We show how these primitives can be leveraged to build effective congestion control in §6.

4.2 Basic Workflow

We now walk through an example (Figure 2) of end-to-end MTP message delivery with in-network mutation.

MTP is connection-based. Each connection endpoint is bound to an IP address and a port. A flow is defined by the 5-tuple as in TCP. We focus on transport operations over an established connection. MTP's connection establishment is achieved with a three-way handshake similar to TCP.

Before sending the first message, the sender queries the service and pathlet discovery protocols for pathlets, and CC remembers these pathlets. To send a message, the application posts to the MTP stack a message descriptor, which carries metadata such as the length and location of the message body. The MTP stack assigns a message number to the descriptor, which increments in the order of descriptors polled, and stores the descriptor in the connection state. CC then selects the pathlets to use for this and subsequent messages and stores the pathlet addresses in the descriptor.

Once CC determines that this message is admissible into the network because there is sufficient room in the congestion window, the MTP stack divides the message body into segments each up to the MSS (maximum segment size, 4096 bytes in this example). A message smaller than the MSS is sent in one segment of the message length. Each segment is identified by a segment number unique within the message. Then MTP encapsulates the pathlet addresses and segments into packets and transmits them.

Upon receiving the segment, pathlet A *immediately* generates a congestion feedback that potentially includes its own congestion signal and link congestion signal such as ECN. We explain how the feedback values are decided in §6. Pathlet A then performs message mutation that increases the message size by 8 bytes so that the message now consists of two segments.

When receiving the first arrived segment of a message, the receiver creates in its connection state a message descriptor that keeps track of the received segments of a message with a bit mask of the received segment numbers. To enable the receiver to determine when all segments are received, every segment packet carries the message length, which divided by MSS gives the number of segments in the message. Once all segments have arrived, the receiver sends an ACK for the

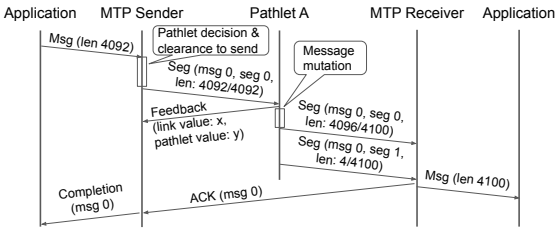


Figure 2: **The MTP basic workflow with mutation.**

(now mutated) message to the sender host. The MTP stack then provides the message to the receiver application and notifies the sender application of completion.

This example shows how MTP works with mutation. Note that the receiver is "passive": It only sends out the E2E ACK for fully received messages and does not check the continuity or order of arrived message numbers. Also, the receiver does not act on out-of-order arrived packets within a message, and can create the message descriptor based on any segment packet of a message that arrives first. Therefore, no special accommodation is needed for reordered messages and even reordered packets, which can be a result from fine-grained network load balancing [20]. However, such a passive receiver design makes it challenging to provide an exactly-once guarantee, which we discuss in §5.2.

By design, MTP does not guarantee in-order message delivery, since this is known to cause head-of-line blocking between messages, which hurts latency-sensitive applications [74]. Applications that require ordering can achieve this by adding their own ordering mechanism [74].

In MTP, a pathlet is expected to output at most one message for each incoming message. The pathlet can intercept a message by sending an E2E ACK to the sender as if the message was delivered to the receiver. Otherwise, the pathlet must forward the message towards the next pathlet or receiver. Any additional payload generated should be added to a message in the form of mutation. We leave the support for branching pathlets in MTP, i.e., multicast, to future work.

5 Reliability

In this section, we discuss the challenges to MTP's reliability protocol that arise from the design choices that we have made, and how we address them.

5.1 Delays and RTOs in MTP

We have previously made the design choice to rely on the sender to detect losses with per-message RTO. In a legacy DCN, relying on a constant RTO for loss detection is acceptable because the network fabric delay is bounded and the RTT is predictable, and the operator can configure a short RTO [9, 61]. However, as discussed in §2.1, an offload can incur a long and unpredictable delay that can be much higher than the fabric delay. Setting a long RTO for the worst-case scenario hurts the quickness of loss recovery, whereas a short RTO causes spurious retransmissions (§8.2.1).

To resolve this, our idea is to distinguish whether the message is in the network or in a pathlet that can incur long or unpredictable delays. With this knowledge, the sender can apply *different RTOs* that fit the respective cases. MTP achieves this with two dedicated types of ACKs as follows.

Pathlet ACKs. A pathlet can leverage two types of ACKs to notify the sender of the arrival and departure of a message, allowing the sender to apply different RTOs. When the pathlet receives all message segments, it sends a pathlet receive ACK, or *PRX ACK*, to the sender to indicate that the whole message is buffered. Until further notice, the sender assumes that the message is being processed, and does not retransmit until the long RTO (*"pathlet RTO"*) is triggered. When the pathlet finishes message processing and sends all the segments out, it sends a pathlet transmit ACK, or *PTX ACK*, to the sender, indicating that the message has left the pathlet. Upon PTX ACK, the sender restarts the short RTO timer for timely retransmission for fabric drops.

Loss of PRX and PTX ACKs does not impact correctness, since the sender always maintains one RTO timer and will retransmit upon timeout until eventually getting the E2E ACK for the message. We prioritize PTX and PRX ACKs, as well as the E2E ACK in the network so that they are rarely dropped.

Configuring the RTOs. With the pathlet ACKs, the fabric RTO can be set to the fabric RTT between the furthest hosts in the network under heavy congestion (i.e., a few hundred μs [43]), plus the message serialization delay, to prevent false positives, because such delay is nearly the worst case between the sender/any pathlet and the next pathlet/receiver. For generally underutilized networks or latency-sensitive messages, the fabric RTO can be set shorter but this risks spurious retransmissions under sudden congestion.

The pathlet RTO should be set to cover the processing delay and the buffering time under pathlet congestion, which is known to the pathlet owner and distributed with the SDP (§3.2), plus the message serialization delay. In practice, though, assuming the message is seldom lost once buffered by a pathlet, the pathlet RTO can be set loosely.

5.2 Exactly-Once Guarantee

Most applications are programmed under the assumption that the transport delivers a message to the receiving application exactly once, and it is important that MTP provides the same semantics. However, MTP's receiver design (§4.2) so far does not guarantee this. Suppose a message is successfully delivered to the receiving application, the MTP receiver removes the message descriptor from the connection state, and the E2E ACK is then dropped; the sender times out and retransmits the whole message. When the duplicate message arrives, the MTP receiver cannot realize that the message is duplicated, and will deliver it to the application again.

Challenge. For the receiver to detect and drop duplicate messages, it must track some state about the delivered messages.

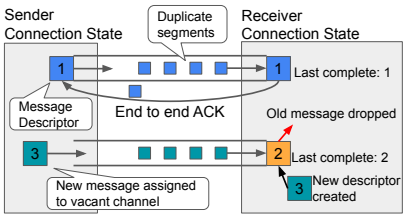


Figure 3: **Workflow of virtual channels.**

The approach must only occupy constant space so as not to exacerbate the connection state scalability issue in datacenters [37].

A strawman is to leverage a fixed-sized window with the lower bound representing the highest message number below which all messages are received and use a bitmap to represent message numbers received within the window. However, when messages can be heavily reordered like in an INC-enabled DCN, a slow-arriving head-of-line (HoL) message can block the window's stepping forward, preventing messages beyond the window from being processed.

Virtual Channels. We devise a scheme called virtual channels[3] to track the delivered messages, which uses constant per-connection state, and is tolerant to reordering. A virtual channel (Figure 3) essentially represents the correspondence between the sender-side state of a message and its receiver-side counterpart. Each inflight message uses a separate virtual channel, and each connection maintains a fixed number of virtual channels (the figure shows 2).

This scheme prevents the delivery of duplicate messages to the application. Each virtual channel maintains the last message number fully received. Suppose the E2E ACK from the receiver is lost, the sender retransmits the message (upper channel in the figure). Note that the message remains in the channel upon RTO. The receiver checks the message number specified in the packets against the last message number. If they are the same, or the just-received segments bear a smaller message number, then the just-received segments are duplicates of the last fully received message. The receiver drops them and retransmits the E2E ACK. On receiving the E2E ACK the sender removes the message state from the channel, which then becomes vacant.

When a new message comes from the application, the sender assigns it to a vacant virtual channel (lower channel in the figure). The virtual channel starts transmitting segments when CC provides the pathlets to use and clearance to send. When the receiver obtains the first arrived segment of the new message, it finds that a new message has arrived because the packet bears a greater message number than its last message number for this virtual channel, then it removes the state of the last message and creates the state of the new message.

This scheme can tolerate high degrees of reordering because each virtual channel independently performs message transfer, and the connection makes progress as long as one channel does. The number of virtual channels determines the maximum number of inflight messages and is communicated during connection establishment. Generally, this number should be set to allow at least one BDP of messages in flight to sustain the line rate. For example, with 4KB messages and a network of 40μs RTT (including the usual processing time of offloads) and 100Gbps line rate, $\sim$122 virtual channels should achieve the line rate.

[3]We use this term differently from existing works where it refers to the multiplexing of the physical interconnect in a computer [18, 39].

6 Congestion Control

MTP provides for congestion control algorithms (CCAs) a framework that consists of two mechanisms, per-pathlet early congestion feedback and per-message pathlet selection (§4.1). We now demonstrate how one can build an effective congestion control solution for INC-enabled DCNs combining these mechanisms with existing ideas for congestion control [8, 43, 67, 77].

6.1 Using MTP's Congestion Feedback

To leverage MTP's per-pathlet congestion feedback, it is important to unify the semantics in which different pathlets express their congestion conditions: Unlike network links, pathlets have vastly different implementation details and performance characteristics, based on which programmers may tend to expose distinct congestion information for each pathlet; however, doing so would require many different CCAs to interpret all the feedback, which is impractical. Also, mixing different CCAs for different pathlets on a path may cause interference among the CCAs.

To this end, we observe that in an INC-enabled DCN, each offload must maintain some sort of logical buffer or queue to temporarily store the pending workload for work conservation, and the buildup of the buffer or queue indicates offload congestion. Therefore, we can use 8 bits to map the pathlet queue or buffer occupancy to 256 quantiles. This is extending the existing idea of 2-bit queuing encoding [77]. The 8-bit encoding is then communicated to the sender with MTP's dedicated congestion feedback packet.

We then employ the Swift CCA to act on the pathlet feedbacks, inspired by how it has been used for endpoint congestion [14]. Swift [43] is a datacenter CCA that is highly effective, achieving very low loss rates at high loads, and deployed in production. It naturally fits because it reacts on RTT measurements, which are multi-bit indirect proxies of the queuing condition that MTP now directly provides, and because it is the state-of-the-art CCA based on multi-bit feedback. Our adapted algorithm essentially adjusts the congestion window (`cwnd`) based on the buffer occupancy feedback to maintain the occupancy at a moderate target level to achieve work conservation without overflowing, which is detailed in Appendix C with parameters we adopt in evaluation.

MTP maintains per-pathlet `cwnd`s and updates them in-

dependently to avoid conflicting feedbacks from different pathlets. It also uses a separate `cwnd` for link congestion. The receiver or next pathlet that provides feedback immediately reflects a marked CE bit to the sender in feedback packets. A message is only "cleared to be sent" when all pathlet `cwnds` and the link `cwnd` have sufficient room.

6.2 Proactive Pathlet Switching

So far we have assumed that all messages of a flow use the same sequence of pathlets. However, recall that the network can provide alternative instances of the same pathlet type (§3). It has been shown that mapping a flow to a fixed set of network resources, though the resources might be randomly chosen with approaches like ECMP, can result in hotspots [67], where many flows are mapped to the same resource, leaving alternative resources underutilized. In an INC-enabled DCN, pathlets are also resources that can become hotspots.

Therefore, it is important to enhance the CCA with the capability to resolve hotspots, which may further improve resource utilization. To that end, we complement our CCA with a mechanism called *proactive pathlet switching (PPS)*. The idea, inspired by PLB [67], is simple: When congestion at a pathlet cannot be quickly resolved by the CCA, which implies that the resource is likely a hotspot, the flow should be steered to use an alternative pathlet. Specifically, PPS monitors the congestion state of pathlets: If congestion causes the CCA to fail to decrease the buffer occupancy reflected in the multi-bit feedback of a pathlet below a threshold for some time, then PPS randomly chooses a pathlet from all the same-type pathlets provided by the DCN for future messages (Line 14 in Algorithm 1 in Appendix C). This is enabled by per-message pathlet selection.

7 Implementation

Our implementation has two types of components: an MTP stack that runs on end-hosts, and wrappers for making L7 offlaods MTP-compatible.

MTP Stack: We used DPDK [23] to implement an MTP stack prototype for testbed experiments. The prototype adopts a queue pair (QP)-based API (resembling RDMA's verbs [57]) for applications to communicate with the MTP stack. The MTP stack prototype is a process, similar to the Snap userspace networking system [56]. However, this is not fundamental, and the MTP stack could also be implemented as a library, in the kernel, or offloaded to the NIC.

Different from RDMA, the MTP stack prototype is a process, which, unlike the NIC, cannot access arbitrary memory in the application. Therefore, when creating the QP, the MTP stack also creates two shared memory regions, one for outgoing (TX) message bodies and one for incoming (RX), and the application maps the shared message body region to its address space. These memory regions are asynchronously accessed by both the application and MTP. To receive a message, the MTP stack must allocate space from the RX region to accommodate the message body; meanwhile, the occupied space of a message can only be freed by the application when it finishes using the received content, and vice versa for sending a message. The freeing of message space is not necessarily in the same order as allocation due to, for example, the reordering of messages. We implement a lock-free memory allocator to efficiently support this.

We encapsulate the MTP API into a shared library called `libmtp`, and implement a message generator that works with this library for evaluating MTP. The MTP stack, `libmtp`, and the message generator add up to ∼6K lines of C/C++ code.

Wrapper and Reference Offload: To allow commodity off-the-shelf L7 offloads to work with MTP, we implement our own MTP-compatible reference message processing wrappers, for complex L7 offloads or "middleboxes". The wrapper is implemented over DPDK with the ability to generate PTX and PRX ACKs, as well as providing early congestion feedback. We assume that routing packets from/to the middlebox is handled by separate switching logic in the INC hardware.

One key challenge is in efficiently utilizing the middlebox's buffer: With the assumption of full buffering, the middlebox cannot start processing a message until all segments have arrived. When one segment is dropped, the received segments occupy the buffer until retransmission comes. During this time other messages that would have been received in full might be dropped due to the lack of space. Our insight here is that a partially received message can be removed if the missing segments don't arrive within a short timeout. This is based on the protocol design that MTP messages are always sent in full and one can expect that the gap between packets of a message is predictable, only subject to serialization and fabric queuing delays. In our experiments, we conservatively set the receive timeout to 100μs.

To aid in our experiments, we embellish the middlebox wrapper with NetCache (§8.1) and a reference L7 offload. The reference offload emulates configurable processing time and buffer size. It reassembles message segments received from the wrapper, adds processing delay, segments the processed message and sends the segments out through the wrapper. This reference middlebox (wrapper and the reference offload) takes 600 lines of C code.

8 Evaluation

We first perform an end-to-end evaluation with NetCache [35] an existing INC application, showing that MTP improves its performance (§8.1). Then in §8.2, we carry out microbenchmarks on testbed to evaluate the two major design aspects of MTP that accommodate pathlet *delaying*: the pathlet ACK-based dual-RTO design for the reliability protocol, as well as MTP's congestion control that consists of pathlet feedback and proactive pathlet switching. We also study the overheads of pathlet ACKs. After that, we use large-scale simulation in

§8.3 to evaluate MTP's support for message reordering (at DCN scale), mutation, and intercept compared to alternatives.

8.1 Application Benchmarks

We adopt NetCache [35], a programmable switch-based key-value cache offload for this experiment. We run the NetCache client on one node in our testbed and run the NetCache backend server on another node. Each node has a Intel Xeon 1.8GHz 16-core CPU and 64GB DRAM. The two nodes are interconnected with 25Gbps Ethernet via a Tofino-based switch which runs the cache offload. We focus on read operations . As per our workflow, the vanilla client (which is also our baseline) encapsulates each read request RPC into a UDP packet toward the cache offload. For cache hits, the cache directly appends the value to the packet and sends it back to the client, and for cache misses, the offload forwards the requests to the backend server. The client recovers from losses of RPCs by retrying after a timeout, similar to key-value cache RPC in production [73]. The MTP-based client achieves the same workflow by establishing a connection to itself via the cache and backend, which are two *chained pathlets* in the path of this connection, and encapsulating each RPC into an MTP message. In this workflow, the messages are *mutated* with the appended values and can be *reordered* due to the cache's direct reply. Loss recovery is provided by the MTP stack.

The backend is allocated a single CPU core which runs at ∼1 Mops, and we use a query pattern of 50% hit rate, with which the system should be able to achieve a maximum throughput of ∼2 Mops. We add to both the cache and backend MTP header parsing logic, as well as to the backend pathlet feedback logic that reflects the RPC queue occupancy. We set both the UDP-based retry timeout and MTP RTO to 125μs, which is ∼25 times the RTT to the backend server. Note that this setup doesn't require dual RTOs for MTP because the delays at both pathlets are predictable.

We let the client generate read operations in an open-loop manner at variable percentages of the max throughput with Poisson inter-op arrival pattern, and observe the achieved throughput and tail op latency, as shown in Figure 4. The MTP-based client can sustain over 95% of system throughput whereas the UDP-based client starts falling below the offered load at 80%. Also, the MTP-based client sees no more than 4x p99 op latency relative to that at zero load, while the tail latency skyrockets with the UDP-based client starting from 75% load. This is because at high loads, the UDP-based client overflows the server and causes huge amounts of spurious retransmission (center figure) that wastes the server throughput, and this is not the case with MTP that provides pathlet CC, which limits the RPC buildup and results in negligible drops and retransmissions. At lower loads, the latency of the MTP-based client is slightly higher due to the extra processing of the MTP stack. Such a delay is necessary and inevitable in providing transport functions, including reliability and congestion control [33, 38, 58, 71].

Component	UDP-Based	MTP-Based
Cache (P4)	618	687
Client (C++)	825	423
Backend (C++)	323	323

Table 1: **Lines of code comparison between UDP-based and MTP-based NetCache components.**

Integration Cost. MTP is of low cost to integrate. Table 1 shows the lines of code (LoC) of UDP- and MTP-based NetCache components. MTP-based cache adds, in addition to the UDP-based, 69 lines of P4, which mainly includes the MTP header definition and parsing logic. The additional Tofino resource consumption is negligible with only 0.3% more crossbar and 2.3% more gateway usage. The MTP-based client is of much fewer LoC than UDP-based because the MTP-based client does not need the timeout-retry logic, which along with DPDK setup is handled by the MTP stack. Also, `libmtp` is light-weight to use. The MTP-based backend uses the same LoC as UDP-based: Coding packet IO with the MTP middlebox wrapper is of similar complexity to DPDK.

8.2 Microbenchmarks

For microbenchmarks we use CloudLab [21] as our testbed. We use nodes with AMD EPYC 3.0GHz 16-core CPUs and 128GB DRAM interconnected by 25 Gbps Ethernet. We place our reference middleboxes on the path between the network interface and the MTP stack emulating SmartNIC processing cores [52, 54]. We assign each middlebox on a dedicated CPU core. Packets that arrive at the host are first handled by the middlebox, which assembles the message and, after processing time, re-packetizes the processed message, and transmits the packets to the MTP stack via a ring buffer.

Though MTP makes no assumption on pathlets' service times, we evaluate MTP by making the middleboxes emulate two commonly-studied long-tail service time distributions: exponential and bimodal [19, 24, 36, 51, 54, 66].

8.2.1 Dual RTOs with PTX and PRX ACKs

MTP adopts PTX and PRX ACKs to determine whether the message is being buffered and processed by a pathlet, which enables using dual RTOs to detect fabric drops ("fabric RTO") and failures of message processing pathlets ("pathlet RTO"). We show the benefits of this with a testbed experiment.

We use our message generator (§7) to issue a flow of 128KB messages from one node to another in a closed loop. The receiver node enables our reference middlebox. The processing time is drawn from either an exponential distribution with a mean of 100 μs, or a bimodal distribution of a similar mean with 95%-ile of 50μs and 5%-ile of 1 ms. The receiver randomly drops 1% of the incoming packets from the interface, which is equivalent to fabric drops from the middlebox's and MTP stack's points of view.

Figure 5 (left) shows a comparison of goodput with a single per-message end-to-end RTO versus with dual RTOs based on

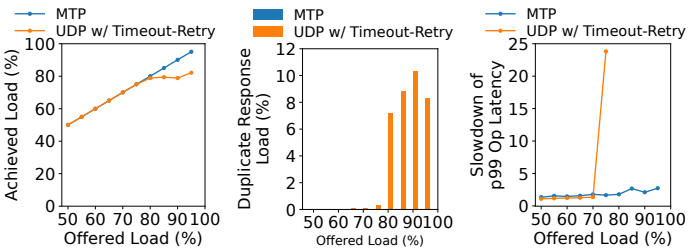


Figure 4: **Left, Center: achieved throughput and retransmission rate as percentage of system throughput, respectively; Right: p99 op latency slowdown relative to unloaded op latency.**

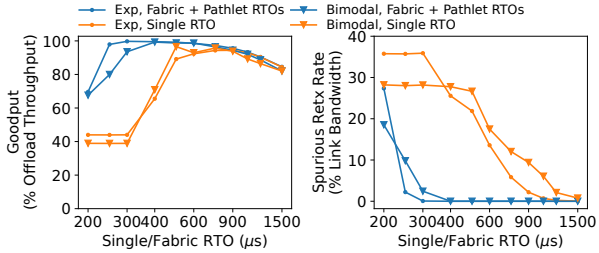


Figure 5: **Achieved throughput with respect to RTO (left) and spurious retransmission rate at the reference middlebox (right).**

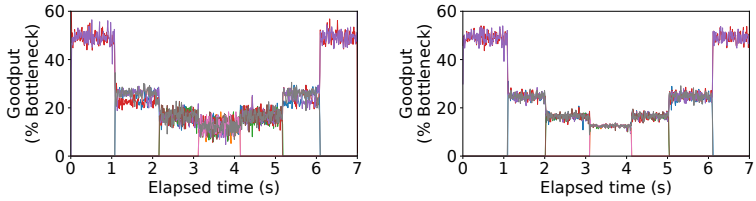


Figure 6: **Convergence under exponential pathlet processing time distriburion with legacy (left) and MTP (right) CC.**

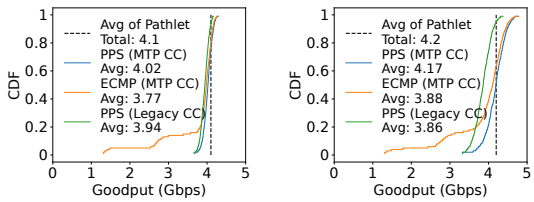


Figure 7: **Per-10ms total goodput with exponential (left) and bimodal (right) pathlet processing times.**

PTX and PRX ACKs. When dual RTOs are used, the X-axis is the fabric RTO because it is in effect as we set the pathlet RTO loosely to 4 ms.

Generally, both overly short and long RTOs cause goodput degradation. Long RTOs impair the sender's ability to quickly recover from losses. Short RTOs cause false positives, where the sender retransmits the messages when they are actually not lost. The right figure shows the rate of packets received at the offload that are spurious retransmissions (or "false positives") and hence dropped. The middlebox determines that a packet is a false positive if it has buffered the same packet of the same message. The message can be pending for processing (received in full), being processed, or waiting for other packets to arrive. We find that MTP's dual RTO design leads to a wider range (400-1050 μs) of fabric RTO choices where MTP achieves >90% goodput with zero false positive (hence no waste of link bandwidth) for both processing time distributions because the PRX and PTX ACKs allow the fabric RTO to not account for the middlebox's unpredictable processing time, whereas such range does not exist for single RTO.

8.2.2 Congestion Control

We compare MTP's CC solution with early multi-bit pathlet feedbacks, reacted by Swift-inspired CCA and PPS, to a legacy solution, where pathlets provide legacy ECN feedback, marking the CE bit in packets when they are congested (>50% buffer occupancy), and the receiver reflects the per-packet ECN back to the sender. We use DCTCP [8] to react on the ECN.

Convergence. We start 8 flows, 2 at each time, of 4KB messages from 4 sender nodes towards one receiver node that has a bottleneck middlebox. We configure the middlebox such that the processing time is drawn from either a exponential distribution averaging 12 μs or a bimodal distribution of 95% 6 μs and 5% 120 μs. The middlebox has a 32KB message buffer, which is roughly 4 times the bandwidth-delay product (BDP). Messages go through this middlebox before entering the receiver MTP stack.

Figures 6 compare how MTP flows converge based on the legacy and MTP's CC solution under exponential processing time distribution. The results for bimodal distribution can be found in Appendix D.1. Each line in the figures represents the throughput of a flow averaged every 10 ms. Most notably, as more flows share the middlebox, legacy CC fails to converge whereas MTP stably converges to a fair share. Early and multi-bit pathlet feedback allows MTP to perform AIMD based on the extent of congestion collected from each feedback than having to wait for at least a round trip to collect and smooth the congestion signals as with single-bit ECN [8].

Proactive Pathlet Switching (PPS). We show how PPS improves performance in addition to congestion control. We use a similar topology as above, with additionally a secondary middlebox of the same type on another node, and provide it to the sender-side MTP stack as an alternative pathlet. We configure the two middleboxes to be asymmetric such that the "secondary" incurs either exponential processing times of 24 μs or bimodal of 95%-ile 12 μs and 5%-ile 240 μs. We generate 4 closed-loop flows, all initially towards the "primary" pathlet, and expect PPS to distribute the flows across both pathlets. PPS randomly chooses a middlebox if the buffer occupancy is above 60% for 3 consecutive feedbacks (MTP CC) or round trips (legacy CC). We use ECMP as a baseline and rehash the flows every 100 ms.

Figure 7 shows the cumulative distributions of the total application-observed throughput every 10 ms. The achieved throughput can at times exceed the total of the pathlets' average throughput because of the long-tail processing time distributions. We find that for both distributions MTP with

PPS can achieve a total goodput that is ∼98% of the average total throughput of the two pathlets, whereas with ECMP can only achieve ∼90%. This is because PPS can quickly redistribute the flows based on congestion, despite random choice having a small chance of mapping all flows to the same pathlet, whereas ECMP cannot recover from such collision until next rehashing. The legacy CC combined with PPS achieves slightly lower average throughput than MTP due to the previously shown convergence problem.

This subsection shows that our CC solution based on MTP's primitives is effective in the face of typical processing time distributions, and we also evaluate MTP's CC with multiple pathlets in Appendix D.2. However, this is not necessarily the only possible way to use the primitives, and it is not our goal to innovate on the CCA. There are known limitations to DCTCP that have been addressed by other CCAs [11,15,25,27,50,60,61]. We leave a systematic study of CC for INC to future work, noting that the similar approaches to evolve from DCTCP may also be used to improve the CC performance in MTP.

8.2.3 ACK Overheads

MTP leverages PTX and PRX ACKs to communicate message states in the network and pathlet feedbacks, in addition to the E2E ACKs. These extra packets can impose overhead on both link bandwidth and the endpoints' processing. We quantify such overhead with an experiment, where one sender node sends messages to one receiver node in a closed-loop manner, with varying numbers of pathlets in between, as well as different message sizes and enabled ACK types (pathlet ACKs + feedbacks vs. feedbacks only). We use 2 CPU cores for the MTP stack in this experiment.

Figure 8 shows the percentage of link bandwidth consumed by ACK packets. The 0-pathlet bars reflect the overhead of E2E ACKs only. In general, the ACK bandwidth overhead is linear in the number of pathlets and the inverse of message size. This is expected because each pathlet ACKs on a per-message basis. Enabling both pathlet ACKs and feedbacks incurs around twice the bandwidth consumption of feedbacks only, because with the former, each pathlet ACKs twice (PRX, piggybacked by feedback, and PTX) per message, whereas with the latter the pathlet sends back only one feedback packet for each message. The worst case occurs with MTU-sized (4KB) messages, 2 pathlets and with both pathlet ACKs and feedbacks enabled, using 6% of link bandwidth, which is negligible. Note that most (if not all) existing L7 INC works place two or fewer pathlets in a path.

Figure 9 shows a breakdown of CPU cycles spent by the sender-side MTP stack with MTU-sized messages, as percentages of total CPU cycles. The total height of a bar represents the total cycles spent on the receive (RX) routine, which consists of four parts: 1) congestion control update (blue), which runs the CCA upon feedback; 2) message state update (orange), which resets the timer and switches between the fabric

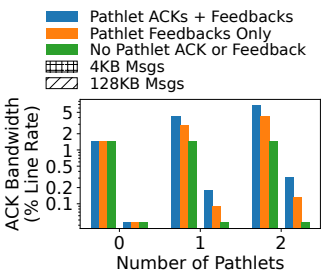


Figure 8: **Link bandwitdh overhead of MTP ACKs.**

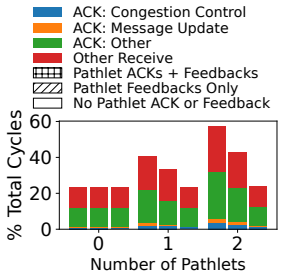


Figure 9: **Sender MTP stack CPU cycle breakdown.**

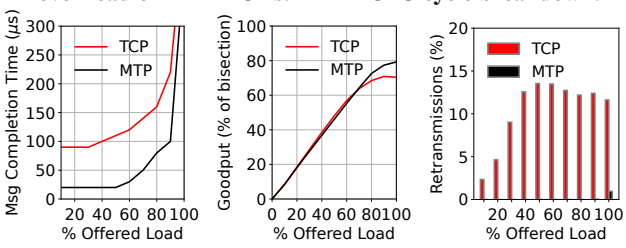


Figure 10: **Left: p99 message completion time; Center: average goodput achieved across the network; Right: retransmissions as percentage of the flows' original packets.**

and pathlet RTOs; 3) other ACK handling (green), which includes updating statistics and preparing CC input from statistics (pathlet feedback, ECN, etc. within a round trip); and 4) other RX logic (red), which involves packet header parsing and connection state lookup. Similar to link bandwidth usage, more CPU cycles are consumed with more pathlets and pathlet ACKs enabled, which incur higher ACK rates. Nonetheless, with 2 pathlets, the total CPU usage of RX is around 55%, still allowing the transmit logic to saturate the 25Gbps link. This is of higher throughput and message rate than some of the well-optimized user-space TCP [33] and software RDMA [58] stacks with the same number of cores.

In comparison, UDP does not have ACKs and hence does not incur similar bandwidth and CPU overheads. However, UDP also lacks essential transport functions such as reliability and congestion control. Building basic reliability upon UDP requires at least end-to-end ACKs which will incur overhead similar to MTP in the 0-pathlet case.

8.3 Large-Scale Simulation

To study MTP's support for reordering at scale, we use ns-3 simulations [7]. We simulate a fat-tree topology with 128 nodes with an over-subscription factor of two. We use 100 Gbps links with varying delays to model asymmetry—50% of aggregation-core links have a delay of 2 μs and the rest have a delay of 4 μs. The switches have a buffer size of 975 KB per port,which closely matches datacenter switches.

We generate *all-to-all* traffic with one-packet messages of 1 KB and a flow size of 1 MB. We use TCP as a baseline, and both TCP and MTP use the DCTCP [8] for congestion control (recall that MTP uses DCTCP for link congestion).

We leverage packet spraying to induce message reordering at scale (our messages are one-packet), instead of simulating L7 offloads. The reordering is exacerbated by the difference in path delays in our fat tree. Results are shown in Figure 10.

The left figure shows the 99%-ile message completion times with respect to load, and the center figure shows the goodput. MTP reduces tail message completion times by about 65% (speed up of 3.6x) across all loads. MTP is also able to operate at higher loads (i.e., beyond 80%) without saturating whereas TCP saturates saturates earlier (i.e., after 60% load).

To understand why MTP's performance is better, the right figure show the number of packet retransmissions as a percentage of the flows' original number of packets. We observe that TCP suffers from 10–15% packet retransmissions and MTP nearly eliminates such overhead as it is robust to reordering. Also, TCP saturates at around 60% load, so the percent of retransmitted packets does not increase after 60% as many flows do not finish and get backlogged indefinitely. We believe this is because MTP can precisely respond to congestion using per-message signals and not misinterpreting reordering as drops or congestion. Further, we show MTP's operation with message mutation and intercept in Appendix E.

9 Discussion

In this section, we discuss the problems that may arise when applying MTP to real-world production scenarios, and the potential approaches to tackle them.

Scalability. In §8.2.3, we have shown that MTP's ACK packets incur reasonable overhead, to both link bandwidth and CPU, with up to 2 pathlets. Though we have argued that this is the case with most existing L7 INC works, more pathlets may be required in a path with the emergence of more INC ideas, or with the mixing of offloads from different applications, which can result in higher ACK overheads. One potential approach to optimize the ACK overheads is reducing unnecessary ACKs. In our experiments, when the dual-RTO mechanism is enabled, all messages would trigger PTX and PRX ACKs. This is not always necessary if the pathlet's processing delay is short most of the time, in which case the pathlet can start sending pathlet ACKs only when long delay is imminent. Similarly, not all messages need to trigger congestion feedback, especially when the pathlet is not heavily loaded. Also, the CPU overhead of ACKs can be mitigated by offloading the MTP stack onto a dedicated ASIC, similar to RDMA.

Streaming. MTP does not natively support streaming, because doing so requires storing transport state in the pathlet for handling losses and out-of-order arrival of data segments, which would complicate MTP and prevent some offload hardware from supporting it. Nevertheless, pathlets that need streaming can first have the sender break down the stream into single-packet MTP messages. MTP can provide reliable per-message delivery and congestion control, on top of which reordering mechanisms can be implemented at the pathlets and endpoints.

Security. Though out of our scope, security remains an open problem for, not just MTP, but INC in general, because INC involves examining and tampering with packet payload by the network, which is exactly what today's transport-layer security (TLS) strives to prevent. In an effort to resolve this, mcTLS [62] extends data privacy and integrity to middleboxes and enables explicitly controlling each middlebox's read/write access to the data being transferred. It may be complementary to MTP and provide similar security features: MTP's handshake can be amended with a similar key exchange, and a TLS record naturally fits in an MTP message. We leave further investigation into security issues to future work.

10 Other Related Work

There is a significant amount of recent research on developing new CCAs, including pHost [25], ExpressPass [15], NDP [30], Homa [61], dcPIM [13], HPCC [50], PINT [11] and BFC [27]. MTP is intentionally designed to be orthogonal and complementary to these new algorithms as datacenter operators can switch between CCAs. For example, MTP's pathlet feedback may be leveraged to communicate the "clearance to send" in receiver-driven CC [13, 15, 25, 30, 61], or it can be seen as a form of in-band network telemetry which has enabled more precise CC [11, 50]. We leave the study on applying more CCAs to MTP to future work.

The idea of using pathlets for LB and CC in MTP is inspired by previous work that uses pathlets for inter-domain routing in the Internet [26]. MTP itself can be thought of as a practical realization of the classical application-level framing idea [16] applied to transport for INC-enabled DCNs.

11 Conclusions

This paper presents MTP, a new message transport protocol designed to work with INC. Through end-to-end testbed experiments, we found that MTP improves the performance of an existing INC application. We also showed that MTP provides effective reliable delivery and congestion control for INC through microbenchmarks and simulations. As INC adoption increases in datacenters, we believe that message-based transport protocols such as MTP that treat offloads as first-class citizens will become invaluable for scaling application performance.

Acknowledgements

We thank our shepherd, Kate Lin, and the anonymous NSDI reviewers for their feedback that significantly improved the paper. This research was supported by NSF grants CNS-2214015, CNS-2207317 and CNS-2008273. Akella was also supported by a gift from Cisco Research.

References

[1] CSP-7551: Hyper Network Appliance. https://www.accton.com/product-csp-7551/.

[2] Intel Extensible Switch Architecture. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-09/intel-tofino-expandable-architecture-paper.pdf.

[3] Intel Tofino Switches. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[4] Performance Best Practices — grpc.io. https://grpc.io/docs/guides/performance/. [Accessed 01-09-2023].

[5] Service discovery - Amazon Elastic Container Service — docs.aws.amazon.com. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-discovery.html. [Accessed 26-08-2023].

[6] Service discovery and DNS | Google Kubernetes Engine (GKE) | Google Cloud — cloud.google.com. https://cloud.google.com/kubernetes-engine/docs/concepts/service-discovery. [Accessed 26-08-2023].

[7] The ns-3 discrete-event network simulator. http://www.nsnam.org.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery.

[10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, 2020.

[11] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[13] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. DcPIM: Near-optimal proactive datacenter transport. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '22. Association for Computing Machinery, 2022.

[14] Prashant Chandra, Nandita Dukkipati, Yuliang Li, Naveen Kumar Sharma, Arjun Singhvi, and Hassan Wassel. Falcon Transport Protocol Specification. https://www.opencompute.org/documents/falcon-spec-v1-1-pdf-1, April 2024. [Accessed 03-10-2024].

[15] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17. Association for Computing Machinery, 2017.

[16] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In Phil Karn, editor, *Proceedings of the ACM Symposium on Communications Architectures & Protocols, SIGCOMM 1990, Philadelpia, PA, USA, September 24-27, 1990*, pages 200–208. ACM, 1990.

[17] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. Offloading load balancers onto SmartNICs. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 56–62, New York, NY, USA, 2021. Association for Computing Machinery.

[18] William J Dally et al. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed systems*, 3(2):194–205, 1992.

[19] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138, 2013.

[21] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[22] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and Daniel Voyer. IPv6 Segment Routing Header (SRH). RFC 8754, March 2020.

[23] Linux Foundation. Data plane development kit (DPDK), 2015.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[25] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[26] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 111–122, New York, NY, USA, 2009. Association for Computing Machinery.

[27] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2022.

[28] Richard L. Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, Lion Levi, Alex Margolin, Tamir Ronen, Alexander Shpiner, Oded Wertheim, and Eitan Zahavi. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[29] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 422–434, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. P4DB - the case for in-network OLTP. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1375–1389. ACM, 2022.

[33] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[34] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. In-network support for transaction triaging. *Proc. VLDB Endow.*, 14(9):1626–1639, may 2021.

[35] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion

Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP. Association for Computing Machinery, 2017.

[36] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[38] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Nikolay Kavaldjiev, Gerard JM Smit, and Pierre G Jansen. A virtual channel router for on-chip networks. In *IEEE International SOC Conference, 2004. Proceedings.*, pages 289–293. IEEE, 2004.

[40] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21. Association for Computing Machinery, 2021.

[41] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 863–879, USA, 2019. USENIX Association.

[43] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[45] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021.

[46] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *SOSP*, 2017.

[47] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[48] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[49] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2016.

[50] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming

Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent Stephens, Hassan Wassel, and Aditya Akella. Ringleader: Efficiently offloading intra-server orchestration to NICs. In *Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, NSDI '23. USENIX Association, 2023.

[52] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using SmartNICs. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(2), may 2023.

[53] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259. USENIX Association, November 2020.

[54] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[55] Shuo Liu, Qiaoling Wang, Junyi Zhang, Wenfei Wu, Qinliang Lin, Yao Liu, Meng Xu, Marco Canini, Ray C. C. Cheung, and Jianfei He. In-network aggregation with transport transparency for distributed training. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 376–391, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Mellanox Technologies. RDMA aware networks programming user manual, 2015.

[58] Mao Miao, Fengyuan Ren, Xiaohui Luo, Jing Xie, Qingkai Meng, and Wenxue Cheng. SoftRDMA: Rekindling high performance software RDMA over commodity Ethernet. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet '17, page 43–49, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM, 2021.

[60] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[62] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in tls. *SIGCOMM Comput. Commun. Rev.*, 45(4):199–212, aug 2015.

[63] John Ousterhout. A Linux kernel implementation of the Homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115. USENIX Association, July 2021.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-accelerated network applications. In *OSDI*. USENIX Association, 2018.

[65] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19. Association for Computing Machinery, 2019.

[66] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale

networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[67] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. PLB: Congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 207–218, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[69] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2021.

[70] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.

[71] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[72] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *SIGCOMM*, 2015.

[73] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20. Association for Computing Machinery, 2020.

[75] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. Skv: A smartnic-offloaded distributed key-value store. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2022.

[76] Web Team. Service Discovery in a Microservices Architecture - NGINX — nginx.com. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. [Accessed 26-08-2023].

[77] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. *SIGCOMM Comput. Commun. Rev.*, 35(4):37–48, aug 2005.

[78] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio: Juniper networks' programmable chipset - for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 633–648, New York, NY, USA, 2022. Association for Computing Machinery.

[79] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. SwiSh: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[80] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps intrusion prevention on a single server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2020.

[81] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[82] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2015.

A MTP Packet Header

Listing 1 shows the MTP packet header formats.

```
struct mtp_hdr_t {      // MTP header
  uint16_t src_port;    // source port
  uint16_t dst_port;    // destination port
  uint16_t len;         // length of this header and payload
  uint16_t checksum;    // packet checksum
  uint32_t msg_no;      // message number
  uint16_t seg_no;      // segment number within the message
  struct {
    uint8_t msg_ack : 1;   // E2E ACK
    uint8_t prx_ack : 1;   // PRX ACK
    uint8_t ptx_ack : 1;   // PTX ACK
    uint8_t feedback : 1;  // pathlet feedback follows
    uint8_t reserved : 4;
  } flags;
  uint32_t msg_len;     // total byte length of message
  uint8_t vc;           // virtual channel
}; // packed struct

struct mtp_feedback_t {  // MTP pathlet feedback header
  uint32_t pathlet_id;   // pathlet instance identifier
  uint8_t feedback;      // feedback value
}; // packed struct
```

Listing 1: MTP header formats.

B MTP Connection State

Table 2 shows the state maintained per connection.

Transmission (TX) Connection State	
`tx_msgs[]`	Message descriptors for TX, each containing a message number, message length, pointer to payload, next-to-send segment number.
`tx_vcs[]`	TX virtual channels, each containing a last message number and pointer to the occupying message descriptor.
`cc_state[]`	CC state for link and each pathlet, each piece containing a cwnd, cwnd occupancy and CCA-specific statistics (§C)
`lb_state[]`	LB state for each type of multi-instance pathlets, containing statistics for PPS (§6.2)
Receive (RX) Connection State	
`rx_msgs[]`	Message descriptors for RX, each containing a message number, message length, pointer to payload, and bitmap that marks received segments.
`rx_vcs[]`	RX virtual channels, each containing a last message number and pointer to the occupying message descriptor.

Table 2: **MTP Connection State.**

C MTP Congestion Control Algorithm

Algorithm 1 shows the implementation details of our Swift-inspired CCA.

D Additional Congestion Control Evaluation

D.1 Convergence

Figure 11 shows the convergence of per-flow goodput with legacy and MTP CC under bimodal pathlet processing time distribution. This time distribution does not cause severe fluctuation in goodput with many flows, even with legacy CC. Despite this, MTP CC still mitigates the divergence of legacy CC with 4 flows (1-2 and 4-5 seconds).

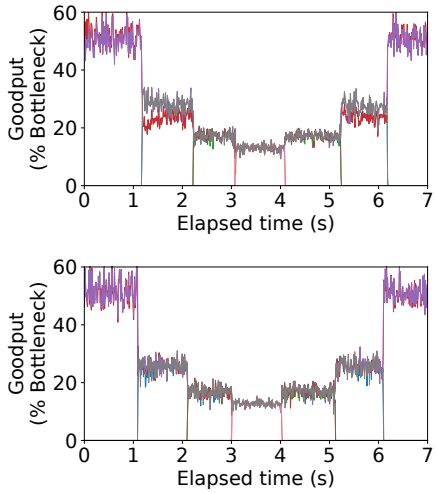


Figure 11: **Convergence under bimodal pathlet processing time distriburion with legacy (upper) and MTP (lower) CC.**

D.2 Multiple Pathlets

There can be more than one pathlet on a path, and it is important that MTP handles this. We evaluate MTP's CC with pathlet chains with different bottleneck locations. We use the topology in Figure 12, which represents a scenario where multiple paths share a bottleneck pathlet after some

Algorithm 1: MTP CC and PPS Algorithms

Data: $active\_pathlet_{type} \forall type \in path$

Procedure on_rx_pathlet_feedback(*pathlet, feedback, msg_seg_num*)

```
    /* Update cwnd with Swift                                   */
    if feedback > 127 then
        cwnd_pathlet ← (0.4 + 0.6 * 128/feedback) · cwnd_pathlet
    else
        cwnd_pathlet ← (1 + msg_seg_num/(2·cwnd_pathlet)) · cwnd_pathlet
    end
    /* Decide whether to switch pathlet                         */
    type ← typeof(pathlet)
    if pathlet = active_pathlet_type ∧ feedback > 160 then
        high_congestion_count ← high_congestion_count + 1
    else
        high_congestion_count ← 0
    end
    if high_congestion_count = 3 then
        active_pathlet_type ← random(type)
        high_congestion_count ← 0
    end
```

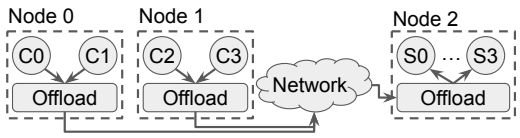


Figure 12: **Multi-pathlet topology. Two client processes on each of Node 0 and 1 send messages to a server process running on Node 2. Messages go through both client- and server-side offloads (pathlets). The bottleneck is the pathlet on Node 2.**

	Exponential (12 μs)		Bimodal (95% 6 μs, 5% 120 μs)	
CC	Avg	Stdev	Avg	Stdev
MTP	5.24	0.179	5.18	0.363
Legacy	5.30	0.360	5.04	0.544

Table 3: **Average and standard deviation of per-10 ms aggregate goodput in Gbps in chained pathlet topology.**

other pathlets. The offloads' processing times are drawn from distributions described in Table 3. We use the message generator as the client to generate 8KB messages (2 segments) in closed loops, set the offload buffer to 96KB, and configure the MTP stack to use MTP or legacy CC.

Table 3 shows the statistics on the total goodput across all flows for every 10 ms. We find that MTP and legacy CC achieve similar average goodput but the legacy CC yields 100% more standard deviation for exponential distribution and 50% more for bimodal distribution. This is due to the drawbacks of legacy end-to-end single-bit feedbacks, similar to the convergence problem discussed in Section 8.2.2.

E Simulation Results

We use a dumbbell topology to study MTP's operation with message mutation and intercept. The topology consists of 8 nodes on each of two sides of the bottleneck. All links are 10 Gbps and 10 μs.

Operation with Message Mutation. We make eight clients from one side send 4KB (4-segment) messages in a closed-loop manner to eight servers on the other side. We simulate

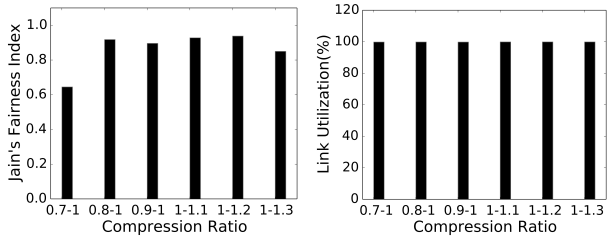


Figure 13: **MTP operation with mutation.**

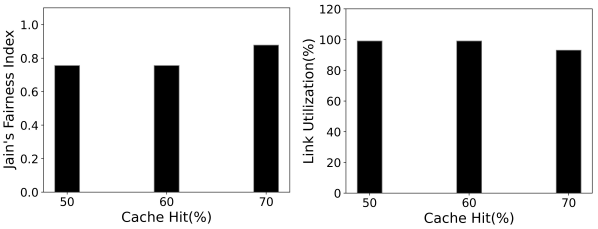


Figure 14: **MTP operation with intercept: fairness (left) & link utilization (right)**

an offload on the switch that (de)compresses data before the bottleneck link with an average processing delay of 2 μs. This represents not only compression engines in the real world but also other offloads such as HTTP load balancer that can change the message size [17]. The offload randomly chooses a compression ratio within a given range. A ratio > 1 means a decrease in message size.

Figure 13 shows the Jain's fairness index across all clients (left) and the utilization of the bottleneck link (right). We observe that MTP achieves good fairness as well as 100% utilization even as the offload decreases or increases the payload size by as much as 30%. However, fairness begins to deteriorate somewhat as the range expands beyond 20% (increase or decrease). We believe that this is due to DCTCP, which is not optimal for handling sudden data size changes in the network, and an optimized CCA for mutation is warranted, which we leave for future work. Note that TCP would fail to operate correctly in such cases.

Operation with Message Intercept. Using the above topology and workload we simulate an offload, which opportunistically intercepts a message. We vary the intercept rate from 50% to 70% (i.e., with 70% intercept rate, only 30% of messages will reach the dumbbell bottleneck). MTP remains unaffected by message intercept. We observe in Figure 14 that MTP achieves good fairness and full utilization of the bottleneck link even as the offload intercepts 50%–70% of messages. Both TCP and RDMA would fail to operate correctly in such cases.