P 4 ce: Consensus over RDMA at Line Speed

Rémi Dulong, Nathan Felber, Pascal Felber, Gilles Hopin, Baptiste Lepers, Valerio Schiavoni, Gaël Thomas, Sébastien Vaucher

▶ **To cite this version:**

Rémi Dulong, Nathan Felber, Pascal Felber, Gilles Hopin, Baptiste Lepers, et al.. P 4 ce: Consensus over RDMA at Line Speed. ICDCS 2024 - IEEE 44th International Conference on Distributed Computing Systems, Jul 2024, Jersey City, United States. pp.508-519, 10.1109/ICDCS60910.2024.00054 . hal-04895326

HAL Id: hal-04895326
https://inria.hal.science/hal-04895326v1

Submitted on 17 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

P4CE: Consensus over RDMA at Line Speed

Rémi Dulong*, Nathan Felber†, Pascal Felber*, Gilles Hopin‡, Baptiste Lepers*,
Valerio Schiavoni*, Gaël Thomas§, Sébastien Vaucher*
*University of Neuchâtel, Switzerland. †EPFL, Switzerland. ‡Ecole Polytechnique, France. §Inria, France.

***Abstract*—P4CE is the first replication protocol that exhibits the same latency and requires the same network capacity as sending data to a single server.**

P4CE builds upon previous RDMA-based consensus protocols. They achieve consensus with a single network round-trip, but with a reduced network throughput. P4CE also achieves consensus with a single round-trip, but without degrading throughput by decoupling the consensus decisions from the RDMA communications. The decision part of the consensus protocol runs on a commodity server, but the communication part of P4CE is fully implemented on a programmable switch, which replicates data and aggregates the acknowledgements in the network, avoiding the throughput bottleneck at the leader. Although simple in its principle, the implementation of P4CE raises many challenging issues, notably caused by the complexity of RDMA and the underlying network protocols, the intricacies of packet rewriting during replication and aggregation, and the restricted set of operations that can be implemented at wire speed in the programmable switch.

We implemented P4CE and deployed it on a commercially-available Intel Tofino switch, achieving up to $4\times$ better throughput and better latency than state-of-the-art consensus protocols.

***Index Terms*—rdma, consensus, programmable switch, smr, tofino**

I. Introduction

Consensus is at the basis of any crash-tolerant distributed system. Ideally, reaching consensus should impose minimal overhead, but all existing protocols make trade-offs between minimizing latency and maximizing throughput.

Modern consensus protocols have shown the importance of using remote direct memory access (RDMA) to minimize latency [1], [2], [3], [4]. For instance, in Mu [1], a leader replicates data by doing a single RDMA `write` to the log that each replica keeps. The network cards of the replicas acknowledge the `writes`, and once enough acknowledgements have been received by the leader, consensus is reached. Key to the low latency of Mu is the use of the RDMA `write` operation that allows the leader's data to be written and acknowledged without involving the replicas' CPUs, drastically reducing the latency exhibited by acknowledgements. While close to optimal in terms of latency, RDMA-based protocols intensively use the CPU of the leader and only use a fraction of the capacity of replicas' network links because the leader divides its own network capacity by the number of replicas. Other protocols are designed to maximize network capacity usage [5], [6] by making the replicas forward messages to each other in a ring. Such protocols are suboptimal in terms of latency because a message has to be forwarded multiple times in order to be accepted by a majority.

The trade-offs presented above arise because, in order to minimize latency, the cost of coordinating between replicas has to be minimized. In consensus protocols that aim towards minimal latency, a single machine *decides* which values to replicate. This one machine then has to *communicate* these values to the other replicas, inevitably leading to the former becoming a throughput bottleneck.

In order to both minimize latency and maximize throughput, we propose to decouple decision and communication. We accomplish this by handling the communication part of the consensus protocol in the network, in the data plane of a programmable switch. Shifting the communication part of the protocol to the switch is advantageous because it saves CPU cycles in the leader, and can replicate and aggregate packets to and from all replicas at link speed.

We implement this idea in **P4CE** (*"P4 Consensus Engine"*). In short, in P4CE a leader machine decides which values to replicate and then sends them to its replicas by sending a single message to a switch, instead of n messages to n replicas. A remarkable consequence of handling the communication part of the protocol in a programmable switch is that P4CE is the first consensus protocol able to reach consensus in a single round-trip (*minimal latency*) while optimally using the network links of both the leader and the replicas (*maximal throughput* by having a single request/response per network link and per consensus).

Despite an apparent simplicity in its principle, implementing the RDMA communications at the switch level yields some intricate issues. Indeed, the RDMA protocol does not support multicast one-sided operations (*e.g.*, the `write` operation used to write in a replica's memory without involving its CPU) [7]. A switch can easily multicast a carbon-copy of a packet but, to support one-sided RDMA operations, the various copies of the packet have to be modified to maintain the illusion that each machine is talking to a single other machine. To that end, the switch keeps track of various stateful metadata, such as the authentication keys that authorize a particular server to read or write a particular region of memory of another server (these keys are randomly generated and different on each server).

The complexity of transparently replicating RDMA connections is further exacerbated by the fact that each RDMA request is acknowledged. So, after multicasting a request, the switch needs to aggregate the associated acknowledgment packets coming from the replicas before forwarding a single acknowledgment packet to the leader. Acknowledgments packets are used by RDMA network cards to inform each other of their relative congestion status. The switch also aggregates

TABLE I: Metadata contained in an RDMA packet.

RDMA field	Usage
Operation code (OpCode)	Type of the packet: *ConnectRequest*, *ConnectReply*, RDMA *write* request, Acknowledgment, RDMA *read* request, RDMA *read* reply, *etc*.
Queue pair identifier	Which queue pair the packet is destined for (conceptually similar to TCP destination ports)
R_key	Randomly-generated shared key between the client and server, which authorizes a particular client to perform one-sided RDMA operations against a server's memory region
Packet sequence number (PSN)	Identifies the position of a packet within a sequence of packets
Credit count	How many requests the client may send to the server at this time

congestion information to avoid a replica from becoming overloaded with pending RDMA requests.

We implement and test P4CE on a commercially-available Intel Tofino switch, and compare its performance against Mu. P4CE outperforms Mu by up to $4\times$ in terms of throughput (on 4 replicas), while exhibiting lower latency. Its source code is available at https://zenodo.org/records/11177068.

To summarize, we make the following contributions:

- We implement an RDMA-compliant multicast interface on a Tofino programmable switch;
- We propose P4CE, the first consensus protocol that achieves *optimal throughput* and *optimal latency*;
- We evaluate P4CE against Mu.

Roadmap. We provide some background on RDMA network protocols and programmable switches in §II. We present the design of P4CE in §III and its implementation in depth in §IV. Evaluation results are shown and analyzed in §V. We finally discuss related work in §VI, before concluding in §VII.

II. Background

We here introduce the underlying technologies upon which P4CE builds, *i.e*., RDMA and programmable switches.

A. Remote direct memory access

RDMA reduces networking overheads by allowing a client to read from and write to a remote server without involving any other component than the network card on the remote side. The Infiniband [7] network protocol provides a comprehensive implementation of RDMA, and it is supported by network cards from several vendors [8], [9], [10]. Unless explicitly stated, we refer to the InfiniBand implementation of RDMA. Next, we explain the terminology relevant to our work, *i.e*., the main metadata that P4CE modifies in order to replicate and aggregate RDMA packets. Table I summarizes these terms.

Queue pair. RDMA operations are transmitted using a point-to-point communication channel between a client and a server. Internally, the client and the server create a *queue pair*, a structure located in the network card that contains a send and a receive queue. The receiving end of the queue pair is uniquely identified by a number, *i.e*., the *queue pair identifier* to disambiguate the target connection of each packet.

Connection handshake. The goal of the connection handshake process is to prepare both endpoints for RDMA transfers. They need to know each other's initial states to be able to later create request packets. There exists a standardized mechanism to manage InfiniBand connections: the *connection manager* (CM). Opening a connection using the CM works as follows. An RDMA server awaits for incoming connections. A client establishes a connection by sending a `ConnectRequest` message to the server. The message contains several fields disclosing information about the client, including the identifier of its queue pair. This identifier will be used by the server when replying to the client, *e.g*., to acknowledge its requests. Once the server receives the `ConnectRequest`, it creates the other half of the queue pair, and sends a `ConnectReply` message to the client. The `ConnectReply` contains the identifier of the server's queue pair (*i.e*., used by the client to send requests to the server). The connection becomes ready for RDMA transfer after a final `ReadyToUse` message gets transmitted from the client to the server.

Key In RDMA, a key (the `R_key`) is associated to every memory region exposed via RDMA. This key is used to avoid bugs, i.e., only the client with the adequate key can access to exposed memory region. During handshake, the server communicates this `R_key` to its client, which is then used by the client for every RDMA `read` or `write` request.

Permissions. Memory regions registered for RDMA operations can be assigned a set of permissions. They decide what operations a given distant machine is allowed to perform on the memory region over RDMA (read and/or write). Any attempt to read or write without the right permissions, or outside of the memory region, will raise an RDMA error.

Read/write requests. One-sided RDMA requests start with the client posting a `read` or `write` request in its send queue. It contains the authentication `R_key` of the memory region the request targets, as well as the virtual address where to read from or write to. In the case of a `write` request, the data to write is attached to the request. The request is asynchronously dequeued by the network card of the client, which builds a corresponding `read` or `write` request network packet. The packet is addressed to the server, and more specifically to the right *queue pair* by including the *queue pair identifier* that was negotiated during the initial connection handshake.

On the receiving side, the network card of the server checks the validity of the `R_key`, executes the given operation (provided the key is correct), and finally sends a response to the client without involving the CPU.

Each RDMA packet contains a field, *i.e*., a *packet sequence number* (PSN) to uniquely identifies the packet within the sequence of packets that are communicated on a queue pair. Each new request increments the PSN. Write requests or read responses that span across multiple packets due to their length use multiple consecutive PSNs. An important aspect of PSNs is that the reply or acknowledgment packet associated to a

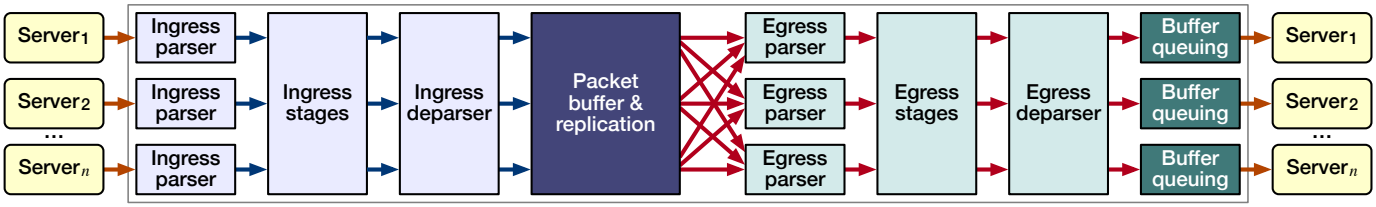

Fig. 1: Portable switch architecture pipeline. Blocks with a light background (black font) are fully-programmable whereas those with a dark background (white font) are configurable. Each server link has its own ingress and egress parser, while the gresses themselves have enough capacity to process simultaneous line-rate traffic across all links.

request with PSN p will have the same PSN p.

Congestion. The *credit count* is an hardware counter, maintained by every RDMA-capable network card. It is used to estimate how many extra requests can be buffered by the card. The idea is that most server-to-client RDMA responses announce the server's current *credit count* back to the client. Those packets can be RDMA `read` response packets or standalone `ACK` packets used to acknowledge writes. When a network card receives the credit count of another card, it uses that information to throttle its request rate to avoid overloading the other card with queries.

B. Programmable switches

Programmable switches allow fine-grain control on packet flows, as well as executing custom operations on those packets, effectively leading to *in-network* processing. Conceptually, switches are split in two layers: the *control plane* above and the *data plane* below. The data plane consists of a specialized application-specific integrated circuit (ASIC), which can process packets at line-rate across all ports of the switch. The control plane, on the other hand, executes on a standard processor and can be programmed using any programming language. Both layers can be programmed arbitrarily. Intuitively, the control plane *controls* how the data plane processes *data*.

The predominant language used to program the *data plane* of a switch is P4 [11]. Intel Tofino is a fully-programmable P4 switch ASIC [12]. The portable switch architecture [13]—a $P4_{16}$ architecture upon which Intel Tofino is loosely based on—defines a *pipelined* architecture composed of an *ingress* and an *egress* (Fig.1). Both *gresses* are composed of three parts: (i) a programmable parser that extracts and organizes data from packet headers; (ii) a series of stages that can modify the packets and decide where to forward them, and (iii) a deparser that rearranges internal metadata to a stream of bytes. The stages are a composition of "match-action" steps: when (part of) the header of a packet matches a set of criteria, an action is performed, *e.g.*, choosing the next step of the processing, modifying the packet, deciding where to forward the packet, *etc*. These "match-actions" are stored in *tables*, the equivalent of a C switch/case, implemented in hardware. In between the ingress and egress sits a buffer and the replication engine. The latter enables flexible duplication of packets across multiple physical output ports. This design forces routing and replication decisions to be taken in the ingress. Conversely, operating on packet replicas must be done in the egress.

As the *control plane* executes on a traditional processor, it is possible to use it for packet processing, with a lower barrier-of-entry in terms of development complexity, but with much lower performance than the data plane.

Intel Tofino switches extend the P4 programming language with stateful operations. In the ingress and egress pipelines, one can store a value in a *register* when a packet matches a set of conditions, and read back the value when a later packet matches another set of conditions. Registers are very flexible: they embed an arithmetic–logic unit (ALU) to perform computations when storing and when reading back its elements.

Computations in the data plane of Tofino can be done at line speed: 100 Gbit/s per link, or 6.5 Tbit/s in total, for the Tofino v1.0, and 400 Gbit/s, or 12.8 Tbit/s in total, for Tofino v2.0. These numbers highlight that a programmable switch can process data at a much higher rate than a normal server machine.

III. Design of P4ce

In P4ce, consensus is done in two phases. First, a leader *decides* the values to agree on, and then it *communicates* these values to the replicas. The key idea behind the design of P4ce is to decouple *decision* and *communication*. This section details the decision protocol, and shows how we optimize the communication part using in-network processing

Decision protocol. P4ce adopts the same decision protocol of Mu [1] (same algorithm for the leader's election, same view change procedure, same process to decide which value to replicate, *etc*.). Each server participating in the protocol keeps a log of values. The leader appends data to its own as well as the replicas' logs. Both the leader and the replicas consume the content of their own logs, asynchronously.

To ensures that logs stay consistent, at any given time only a single leader is allowed to write to the logs. Replicas rely on RDMA permissions to control which machines are allowed to write to their log, *i.e.*, which machine they consider to be the current leader.

Specifically, every involved machine is assigned an identifier. The leader is always the live machine with the lowest identifier. To prove its liveness, each machine keeps a *heartbeat* value, periodically increased. Machines frequently read each

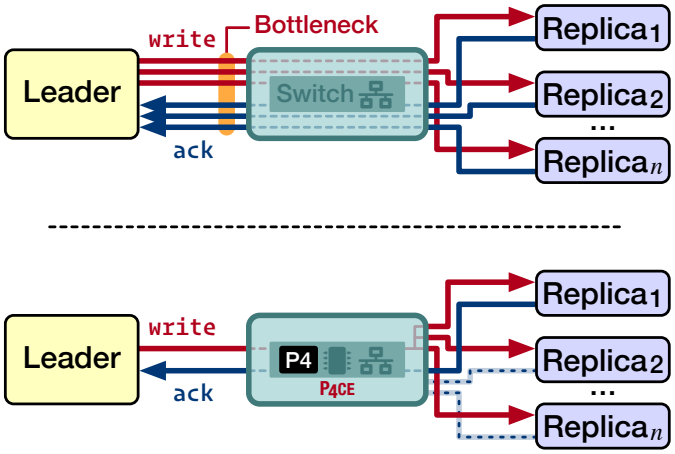

Fig. 2: Communications in Mu (top) and P4CE (bottom). In P4CE the leader sends a single RDMA write to the switch.

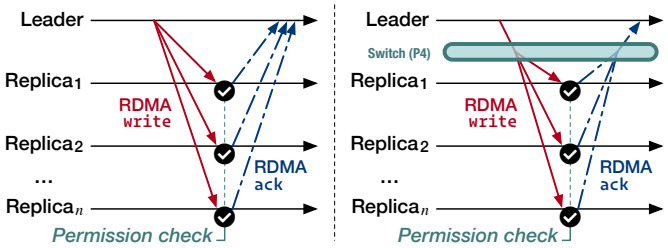

Fig. 3: Communication pattern used for consensus: without the programmable switch (left), messages are sent and response are processed by the leader, which represents a bottleneck; with P4CE, these tasks are performed directly within the switch.

other's heartbeats: the liveness of other machines is assessed by checking if their heartbeats increase over time.

Once a replica has chosen another machine as the current leader, it reconfigures its RDMA permissions to exclusively allow the newly-chosen leader to write to its log. A replica that elects itself as a leader waits until it gets the permission to write to a majority of the other machines before writing any message. This ensures that the actions of a leader can always be seen by a majority of the participating machines.

When the leader decides a value to write, it first writes it in its log, and then broadcasts it to the replicas. It does so using RDMA `write` commands to append data to the replicas' logs. A value is considered replicated once the network cards of f replicas have acknowledged the write.

Communication protocol. Figure 2 presents the dataflows of Mu (top) and of P4CE (bottom). The communication patterns are shown in Figure 3. With P4CE, when the leader `writes` in the log of the replicas, instead of individually sending a write request to each replica, it sends a single write request to the switch, which in turn replicates the request to the replicas on behalf of the leader. Hence, the switch waits for f positive acknowledgment packets to send a single acknowledgment packet to the leader. By reducing the amount of messages exchanged, P4CE fills up the network capacity of all links available, regardless of the number of replicas.

To implement the communication logic of P4CE in a switch, we implement a transparent RDMA group communication library within a programmable switch. To handle a request, the switch ensures two key features: broadcast and gather.

Broadcast. When the switch receives a request, it broadcasts it to a set of replicas. For that, the switch first duplicates the request, and then tailors each copy for each replica. At a high level, it rewrites the destination queue pair, the authentication key, the virtual address of the buffer accessed by the request, the packet sequence number and the IP address of the destination.

Gather. The RDMA protocol specifies that write requests have to be acknowledged. All replicas will reply to the switch with an acknowledgment message, which can be positive in case of success, or negative in case of a failure. The switch waits for f positive acknowledgments before transmitting an acknowledgment to the leader. If one of the servers sends a negative acknowledgment (NAK), the switch forwards it immediately to the leader. This allows the leader to become aware that one of the replicas is misbehaving and to handle the error—either by excluding the replica from future broadcasts, or by doing more in depth diagnosis.

Upon receiving the f^{th} acknowledgment, the switch rewrites several fields of the message to prepare it for forwarding to the leader: the destination queue pair, the packet sequence number and the IP address of the destination. Key to the proper working of the RDMA protocol, the switch also has to re-compute the fields related to congestion control.

A. Impact on correctness

Having a switch handle the communication may seem like introducing a single point of failure. In this section, we discuss the impact of the switch on correctness.

Faulty replica. P4CE only uses in-network acceleration to replicate the logs. The *heartbeats*, exchanged by the replicas to prove their liveness, are not accelerated (hearbeats only represent a few hundred messages per second and are not performance-critical). Hence, the switch does not add any complexity in the detection and handling of crashed replicas.

The only change is the handling of RDMA errors, which can happen in case of packet drops, or in case of congestion. When the switch receives a negative acknowledgment (NAK), it unconditionally forwards it to the leader. P4CE then reverts to un-accelerated communications: the leader starts sending packets to individual replicas instead of using the switch to replicate packets. If further NAKs are received, their handling is done with the legacy decision and communication protocols, and P4CE has the same correctness guarantees as Mu. If no further errors are received, the leader then periodically tries to re-establish a connection through the switch to enable in-network replication again.

Faulty leader. In case of a view change, one of the replicas becomes the new leader. The leader configures the switch

so that its messages get multicasted to the other replicas. It is possible that, for a while, the switch maintains both the multicast group of the old leader and of the new leader. However, any attempt to broadcast from the old leader will result in an error, because the replicas have updated their permissions to only permit write operations from the new leader.

Faulty switch. Similarly to previous work, we assume that switches either work correctly or crash (we do not handle Byzantine behavior). In case of a switch failure, the leader's requests will eventually time out waiting for an ACK. Upon timeout, the leader reverts to un-accelerated communications. Provided that the replicas can be reached via another network route—which is frequent in datacenters to tolerate failure—the leader will be able to connect to a majority of replicas and will run the protocol using manual replication. In that case, the leader periodically tries to re-establish a connection with a P4CE-enabled switch to check its liveness and eventually regain in-network acceleration.

Summary. In-network replication is not a problematic point of failure. Consensus protocols are already able to use a fast path, used during normal operation, and a slow path, used in the presence of faults. Similarly, in case of a switch fault, P4CE reverts to an un-accelerated fast path, and then, possibly, to a slow path. As a consequence, P4CE offers the same correctness guarantees as Mu.

IV. Implementation

With P4CE, the leader decides a value, and forwards the decision to the programmable switch, which replicates the value on the replicas. For that, the data plane of P4CE implements group communication for RDMA in the programmable switch. Since RDMA, as specified by the InfiniBand protocol, only supports point-to-point `read` and `write` commands [7], the switch acts as a middle man that transparently multicasts and gathers RDMA packets between a source and multiple destinations. The data plane of P4CE thus needs to duplicate, redirect, and transform individual packets so that every participant in the network has the illusion of communicating with a single machine.

Internally, the data plane of P4CE maintains a set of *communication groups*. A group is composed of a single *source* endpoint (*i.e.*, the leader) and a set of *destination* endpoints (*i.e.*, the replicas). When the switch receives a request from the source, it broadcasts the request to all the destinations. In the reverse direction, it aggregates all responses and forwards a single response to the source.

We implemented the data plane of P4CE in $P4_{16}$ on an Intel Tofino-based programmable switch [12]. Because our target platform expects the physical layer to be Ethernet instead of InfiniBand, our implementation is based on RoCE v2 (IP Routable RDMA over Converged Ethernet), a standard variant of InfiniBand that replaces its lower layers with UDP, IP, and Ethernet. The RDMA semantics offered when using RoCE are totally identical to those offered when using pure InfiniBand.

This section presents the main data structures used by P4CE, how a machine creates a communication group and how we implement multicasting in the data plane.

A. Communication groups and connections

Without in-network replication, the leader initiates the consensus protocol by establishing an individual RDMA connection with each replica. This operation is now handled in the switch by P4CE. With P4CE, the leader establishes a single RDMA connection *to the switch* and the switch broadcasts the connection request to the replicas.

Capturing incoming connections. P4CE configures the data plane of the switch to have all `ConnectRequest`s intended for the switch, *i.e.*, that contain its IP address as destination, to be redirected to the control plane (see Section II-A, a `ConnectRequest` is the first message of the handshake necessary to establish a RDMA connection). New connections are not a frequent operation, so handling them in the control plane simplifies the development effort and has negligible overhead in practice.

Setting up the connection. The RDMA protocol allows `ConnectRequest`s to be piggybacked with custom data. In P4CE, we use the custom data to store the IP addresses of the replicas participating in the communication group. The control plane of the switch establishes a connection with each of the replicas. Each replica receives a `ConnectRequest`, to which it replies with a `ConnectReply` as it would while communicating directly with the leader. The control plane aggregates the answers and sends a single `ConnectReply` to the leader.

The `ConnectReply` sent by the servers can also be piggybacked with custom data. As for the legacy RDMA protocol, we use the custom data to send a virtual address and a virtual authentication key (the R_key described in Table I). The virtual address is equal to zero, and adjusted during replication of write requests to match the addresses of the clients (see Section IV-B). Similarly, the virtual authentication key is equal to a random number, and adjusted during replication.

In case the replica refuses to establish the connection, it sends an error in the form of a `ConnectReject` packet. In that case, we follow the logic of the Mu [1] protocol (see Section III).

Getting ready for future RDMA commands. After establishing connections, all requests are handled by the *data plane* to guarantee that further processing is done at line speed. To allow the data plane to broadcast the packets to the right set of replicas, P4CE creates a multicast group in the replication engine of the switch. The multicast group has a unique identifier that will be used by the data plane when scattering packets.

On top of handling future RDMA commands for the established connections, the control plane still listens for new `ConnectRequest` packets to create new parallel connections, as P4CE supports multiple consensus groups in parallel.

Metadata per group. For each communication group (Table II), P4CE generates a `BCast` queue pair, which is sent

TABLE II: Multicast metadata.

Name	Meaning
BCastQP	Virtual queue pair identifier for the source; all packets received on this QP will be broadcasted
MulticastGroup	Unique identifier used by the replication engine to know the destination ports of a multicast command
AggrQP	Virtual queue pair identifier for the replicas; all packets received on this QP will be aggregated and sent to the source
NumRecv[PSN]	An array used to store the number of replies for a given PSN (the PSN is the one of the leader)
MinCredit[Replica]	Minimum credit of the replicas of the communication group

TABLE III: Connection structure. P4CE keeps one connection structure for each connection it made or received from a server (leader or replica). The connection structures can be found using an *endpoint identifier*.

Name	Meaning
Remote IP	IP address of the remote endpoint
Remote QP	Queue pair identifier of the remote endpoint
Remote Port	Port of the remote endpoint
VA	Virtual address of the remote buffer
Size	Size of the remote buffer
R_key	Remote authentication key

to the leader in the `ConnectReply`. All messages received on that QP are broadcast to the replicas. At the replicas side, the switch generates a `Aggr` queue pair used to identify the communication group.

For each communication group and each in-flight request sent by the leader, the switch maintains a counter. This counter, named NumRecv, is identified by the PSN (see Table I) of the request sent by the leader. When the switch receives a request from the leader, it initializes NumRecv[PSN] to zero and forwards the request. Then, the switch increments NumRecv[PSN] for each answer received from a replica, which allows the data plane to count the number of acknowledgment messages received from the replicas for a given request of the loader. The f^{th} ACK is forwarded to the leader because receiving f acknowledgments ensures that strictly more than half of the servers agree on the value (*i.e.*, the f replicas + the leader).

P4CE also maintains congestion control-related metadata in order to avoid packets being dropped. The `MinCredit` represents the number of packets that the RDMA card of each replica can handle. The minimum value across all replicas is communicated to the leader in each acknowledgment packet.

`NumRecv` and `MinCredit` are stateful metadata. They are directly computed in the data plane and stored in *registers* (see Section II-B).

Metadata per connection. For each established connection, P4CE stores metadata in the tables of the data plane of the switch. This metadata is used to transparently forward write requests and acknowledgments. Table III presents the per-connection metadata. For each connection to an endpoint (leader and replicas), P4CE maintains a structure named the *connection structure* (also in Table III). The structure fully identifies a connection: it contains the IP address of the endpoint, its queue pair identifier and its port. When the endpoint is a replica, the structure additionally contains the virtual address of the buffer, the size of the buffer and the authentication key. P4CE internally identifies a connection with an 8-bit integer that we refer to as *endpoint identifier*.

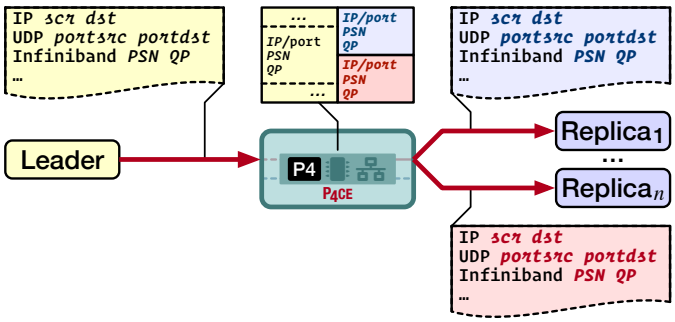

Fig. 4: Principle of packet duplication: P4CE transforms the different protocol addresses and identifiers in the headers to provide the illusion that the replicas receive the requests from the switch.

B. Scatter

Multicasting write operations is critical for performance. P4CE implements them in the data plane.

High-level implementation. Figure 4 presents the high level principle of packet duplication. The data plane mainly adapts the IP, UDP and RDMA connection fields to maintain the illusion that replicas are receiving packets as if they were coming from the switch itself.

Most importantly, after duplicating a packet, P4CE updates the PSN and queue pair used by the leader to match those expected by a replica. P4CE also replaces the virtual address/R_key given by the leader by the actual virtual address/R_key used by the replica because each replica allocates its log at its own virtual address and generates its own key.

Inside the switch. Specifically, when a `write` request enters the switch *ingress* pipeline, P4CE starts by matching the destination IP to check if the packet is addressed to the switch. If not, it means that the packet is not meant to be duplicated, and it is transmitted directly to its destination. Otherwise, P4CE matches the destination queue pair (QP) number contained in the packet (i.e., `BCast` QP) against the ones that were stored in a P4 *table* during the initialization phase. Matching against the contents of the P4 *table* returns a `MulticastGroup`.

The data plane uses the `MulticastGroup` as the key to instruct the hardware multicasting engine of the programmable switch to duplicate the packet according to the configuration

made by the control plane when setting up the communication group. To ensure that the future answers to the request are properly aggregated, the dataplane also resets `NumRecv` at the index corresponding to the PSN of the packet it is multicasting (see Table II for the definition of `NumRecv`).

After the multicast engine, n packets are generated in the *egress* pipeline. At that stage, the n packets are carbon copies of the original packet. There are a couple differences between them. First, each of them is en route to a particular physical output port of the switch. Second, the only way to properly differentiate them is by using an identifier that the multicast engine sets in the metadata associated with each packet. To speed up computations, P4CE configures the multicast engine so that the identifier consists in the *endpoint identifier* of the destination replica.

P4CE matches the endpoint identifier against a P4 *table*, which gives the connection structure of the replica (see Table III). Updating addresses, ports and similar identifiers in Ethernet, IP, UDP and InfiniBand headers of the packet requires replacing the original values with the values saved in the connection structure during the initialization phase. Updating RDMA-related fields requires to recompute some values. For instance, if the leader writes at offset o of the log of its replica, the data plane updates o to write at $VA+o$, VA being the virtual address of the replica's log.

Note that RDMA commands may spawn multiple packets when the amount of data to transfer exceeds the maximum transmission unit (MTU), *i.e.*, when the payload cannot fit in one network packet. For instance, when writing a large amount of data on a connection configured with the Ethernet-standard MTU of 1500 B, a `write` request may get split into multiple packets, each with a payload of 1 KiB. In that case, P4CE multicasts each individual packet to all replicas. As intermediate packets can be acknowledged, we apply the same actions as for any request packet.

C. Gather

At a high-level, P4CE waits for f answers from the replicas before sending an answer to the leader. However, aggregating the answers is not trivial, since the switch must compare the answers in order to count them, which makes the operation stateful.

Inside the switch. P4CE aggregates answers using the `NumRecv` *register* (see Table II). In our current implementation, we can aggregate 256 different PSNs per connection at a given time, which means that P4CE can handle up to 256 un-acknowledged packets on the fly per connection. As a comparison, with our experimental setup, a given RDMA connection can only have up to 16 pending write requests. Our current sizing thus works on current networks and is likely to remain viable on future faster networks.

Upon receiving the first response from one of the replicas, P4CE updates its *registers* accordingly: (i) it translates the PSN to what the leader expects; (ii) it sets NumRecv[PSN], the number of received replies, to 1, and (iii) it tells the *deparser* of the ingress to drop the packet.

When receiving later packets, P4CE simply increments the number of replies. If less than f answers have been received, the packet is dropped. When the f^{th} answer has been received, P4CE forwards the packet to the leader.

Additionally, acknowledgment messages contain important information about current InfiniBand-related resource usage; most importantly, the "credit count" needs to be correctly sent back to the leader to prevent overloading. As replicas may handle queries at a different rate, P4CE takes the worst case into account. The data plane stores the most recent credit count announced by each replica in *registers*, and sends the minimum count across replicas to the leader (more details on how we achieve this can be found in Section IV-D). The credit counts are stored at the communication group level, and not per PSN, to be able to send the latest credit count received per replica. Otherwise, because the $(f)^{th}$ ACK is forwarded, the credit count of the slowest replicas would likely be ignored.

D. Under the hood

We now present some more intricate details about how P4CE works at a finer granularity. The *data plane* code of P4CE consists of 949 lines of $P4_{16}$ for Tofino Native Architecture (TNA) [14], while its *control plane* amounts to 1237 lines of Python. In the control plane, we use Scapy [15] as a framework to decode and craft RDMA CM initialization packets. The control plane uses Barefoot runtime (BfRt) application programming interfaces (APIs) to interface with the data plane.

Performance concerns. With our P4 code running, each ingress and each egress parser can process 121 million packets per second. While large, this number actually becomes a bottleneck if a single parser ends up parsing queries coming from multiple replicas. For instance, in our first implementation, all the ACKs coming from the replicas were first processed in the replicas ingresses and then sent to the leader's *egress* where they were dropped. As a consequence, the leader's *egress parser* was a bottleneck and P4CE was only able to aggregate a total number of 121 millions packets per second. Changing the processing of ACKs to drop the packet directly in the *ingress* of the replicas, before they reach the *egress* of the leader, allows us to handle 121 million answers per second and *per replica* (so a total of 726 millions ACKs per second with 6 replicas for instance).

Stateful in-network computations. Performing in-network computations is harder than one may think at first glance. As an example, we explain how we compute the minimum credit count of the replicas. We use one *register* per replica, arranged across the whole length of our pipeline, spanning both *gresses*. Each register stores the last credit count seen from a replica. On a standard ARM/x86 CPU, it would be easy to compute the minimum of these values. However, doing the computation on a programmable switch is tricky because, on a Tofino switch, it is not possible to write the following code:

```
if(a < b) min = a else min = b
```

... because it is not possible, in hardware, to compare two variables (the ASIC can only compare a variable with a constant). Hence, we work our way around the limitations of

the ASIC using indirect ways to perform computations. In that case, the computation of the minimum ends up being:

```
if(identity_hash( (a - b) underflows? ))
        min = a else min = b
```

Checking if a subtraction underflows is a standard way of emulating comparisons, but the result of the underflow cannot be used in a conditional clause (because no cabling exists between the underflow information of the ALU and any conditionally programmable hardware). We thus forward the underflow information to an `identity_hash` (a module that simply returns the input value), which can finally be used in a conditional clause.

While this example may seem atrivial, one of the major source of complexity when performing in-network computing stems from the stringent hardware constrains of the Tofino ASIC. Notably, every computation that a developer wants to program in P4 could be implemented in dozens of possible ways, but most of them cannot be deployed in hardware.

V. Evaluation

In this section, we present our detailed evaluation of the latency and throughput of P4CE, comparing it against Mu.

A. Experimental setup

We tested P4CE on 5 Supermicro SYS-520P-WTR machines, each with an Intel Xeon Gold 6326 CPU (16 cores). Three of these machines have 64 GiB of DDR4 random access memory (RAM), and two have 128 GiB. These are all connected through an Edgecore Wedge 100BF-32X programmable switch that contains a 2-pipe first generation Intel Tofino ASIC. All machines are equipped with NVIDIA ConnectX-5 network cards interfaced through PCI Express 3.0 x16 links. Each card is directly connected to the programmable switch using 100 Gbit/s Ethernet.

B. Methodology

Each point of measure is an average value of 1 million operations. We noticed very small variations (less than 1%). We compare P4CE with Mu on various setups, and in particular varying the number of replicas. We report our lessons learned for each benchmark.

C. Network capacity and throughput

Maximum goodput. We first evaluate the maximum goodput achievable by P4CE, varying the size of the values sent by the leader (Figure 5). The goodput is the number of useful bytes sent per second, which excludes protocol overhead bits. Compared to Mu, P4CE mutiplies the goodput by 2 when the consensus involves 2 replicas, and by 4 when the consensus involves 4 replicas. P4CE is able to achieve consensus at link speed with value sizes above 500 B (11 GB/s of goodput shown in the figure, 12.5 GB/s total throughput). These results are expected: in P4CE the leader sends a single message to the switch and so can use its network link at maximum capacity, while in Mu the leader shares its network capacity between replicas.

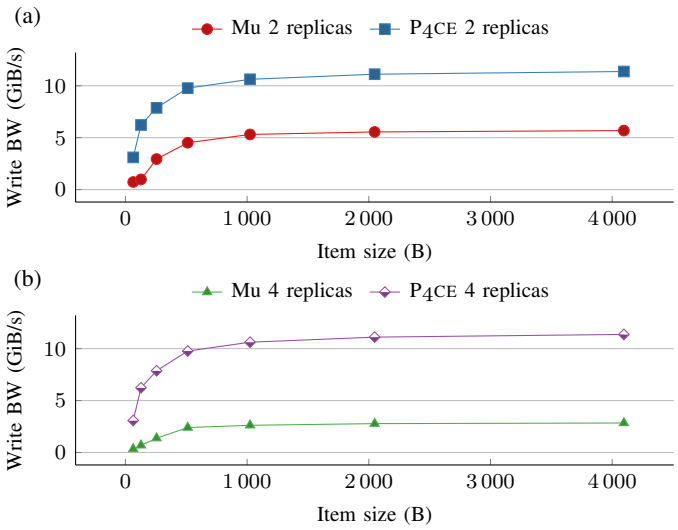

Fig. 5: Write goodput with different item sizes. P4CE maximizes the available network capacity while Mu is limited by the leader's ability to duplicate packets. (a) With 2 replicas; (b) with 4 replicas.

Maximum number of consensus per second. In order to show that sending a single message is also advantageous in terms of CPU usage, we performed throughput measurements on small values (64 B). In this configuration, the network is not a bottleneck; the consensus is limited by the rate at which the leader can generate RDMA packets and the bottleneck is the CPU of the leader.

P4CE can sustain 2.3 million consensus per second, a 1.9× speed increase over Mu with 2 replicas and around 3.8× with 4 replicas. P4CE achieves higher throughput because it generates fewer packets (half with 2 replicas, quarter with 4 replicas, *etc.*) and because it does not need to aggregate the ACKs of the replicas.

Lesson 1 : *When doing consensus on large values,* P4CE *outperforms Mu because the leader does not have to share its network capacity between replicas. When doing consensus on small values,* P4CE *outperforms Mu because it has a lower CPU overhead. Decoupling the decision and the communication is thus beneficial, regardless of the size of the values to be replicated.*

D. Latency

In this section, we evaluate the impact of P4CE on latency. Figure 6 presents the relationship between throughput and latency, for P4CE and Mu. Below 700 k consensus per second with 2 replicas and 400 k consensus per second with 4 replicas, P4CE's latency is 10% lower than that of Mu. The small difference is due to P4CE doing a bit less work on the critical path of queries (fewer RDMA requests, and no aggregation of ACKs), but neither P4CE nor Mu are CPU bound.

However, above 700 k consensus per second (400 k with 4 replicas), Mu becomes CPU-bound and its latency starts to

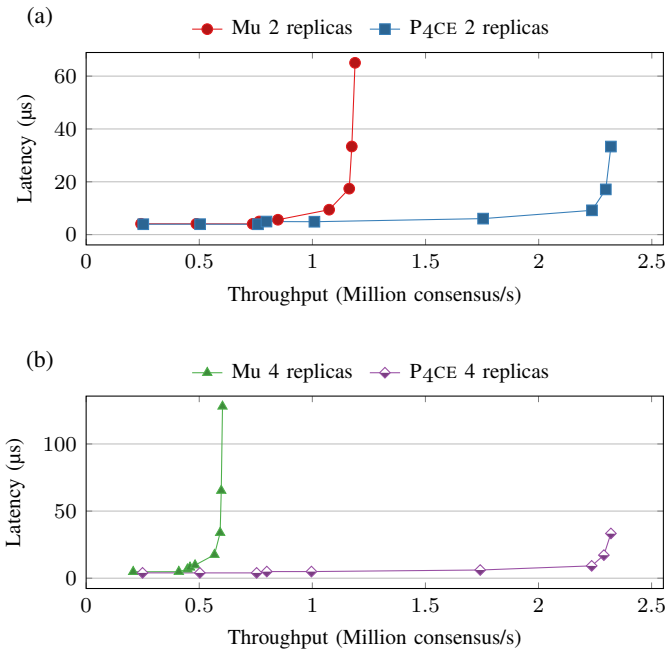

Fig. 6: Evolution of latency with 64 B requests vs. throughput. (a) With 2 replicas; (b) with 4 replicas.

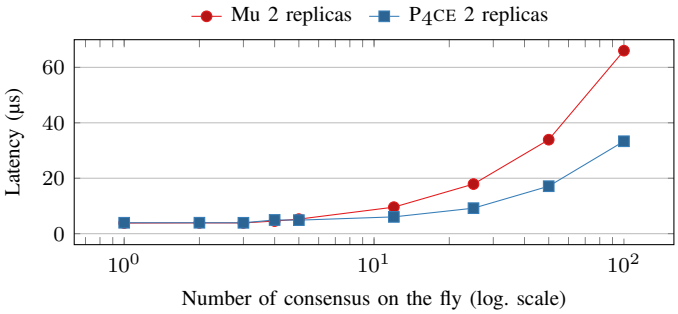

Fig. 7: Latency with 64 B requests

TABLE IV: Average fail-over times.

	Mu	P4CE
Crashed replica	0.1 ms	40.1 ms
Crashed leader	0.9 ms	40.9 ms
Crashed switch	60 ms	60 ms

grow. Mu cannot handle more than 1.2 million consensus per second (600 k with 4 replicas) and queries start accumulating when generated at a higher rate. Thanks to its lower overhead, P4CE can handle up to 2.3 million consensus per second, regardless of the number of replicas.

A similar observation can be made on the latency of short bursts of consensus operations. Both Mu and P4CE can batch queries: when the leader receives a burst of queries, it sends a burst of RDMA `write` requests to the replicas' logs instead of sending one request and waiting for its ACK before sending the next one. As a consequence, Mu and P4CE can have multiple consensus "on the fly". We measure the latency of bursts of requests, varying the size of the burst. Results are shown in Figure 7. The latency difference between P4CE and Mu increases with the number of consensus on the fly. Mu starts to become CPU-limited when handling more than 10 queries simultaneously. P4CE's latency is half that of Mu when handling bursts of 100 requests. So doing less work on the critical path of queries also improves the performance of short bursty workloads, regardless of the total throughput.

Lesson 2 : *Decoupling decision and communication is also beneficial in terms of latency, as it reduces the amount of work to be done in the critical path of the leader.*

E. Failures

We conclude our experimental evaluation by evaluating the impact of P4CE on the handling of failures, and the impact of reconfiguring the switch on the latency of view changes. We simulate failures at the replica level and leader level by killing the applications, as in the original Mu paper. We also test failures at the switch level by powering the switch off. Table IV summarizes the fail-over times.

Configuring a communication group. Configuring a communication group is done by sending a `ConnectRequest` to the switch. Since reconfigurations are infrequent, we did not try to optimize their performance. Sending a `ConnectRequest` and waiting for the switch to reconfigure its dataplane takes 40 ms on average.

Crashed replica. Mu and P4CE rely on hearbeats to check the liveness of replicas. Since the heartbeats are exchanged every 100 µs, they are not performance-critical and are not accelerated by the switch. As a consequence, detecting a crashed replica has the same latency in Mu and in P4CE. In Mu, the leader simply excludes the replica from its multicast group. In P4CE, the communication group of the switch is updated, adding a 40 ms delay to the reconfiguration.

Crashed leader. In Mu, electing a new leader mainly consists in changing the permissions of the queue pairs. The operation takes 0.9 ms on average. P4CE adds a 40 ms latency for the reconfiguration of the switch.

Crashed switch. In both Mu and P4CE, a failed switch results in packet drops, which cause the RDMA protocol to time out. The timeouts can be configured at the network card level, and need to be adapted to the network topology. In our setup, the network cards are configured to time out after 131 µs (timeout values in RDMA networks can only take discrete values of the form 4.096×2^x µs). Once a timeout is detected, both Mu and P4CE re-establish connections using a non-accelerated alternative route, which takes most of the time. Reconnecting and reconfiguring Mu takes 60 ms in both cases.

Lesson 3 : *Enabling in-network replication increases the time it takes to handle crashed replicas and view changes because the switch needs to be reconfigured. However, this added delay (*40 ms*) is negligible in most real-world scenarios where crashes are infrequent. Note that the reconfiguration of the switch could also be done asynchronously:* P4CE *could manually replicate packets while the switch is reconfiguring, and then use in-network replication once the switch is reconfigured. In that case, Mu and* P4CE *would have identical fail-over times.*

VI. Related Work

The availability of programmable switches on the market and their increasing availability continues to attract the attention of academia and industry, opening the pathway to novel and more efficient network protocol designs [16].

a) Network hardware acceleration: Network hardware acceleration has been used to accelerate Paxos, and other consensus protocols [17]. NetPaxos [18] motivated the use of switches to accelerate the roles of the original Paxos protocol. P4xos [19] (and its ancestor [20]) implemented these ideas, using multiple Tofino switches, each playing a specific role (one switch is a proposer, another the acceptor, *etc*.). P4xos implements the decision of these roles inside the switch, and packets are forwarded between switches as they would be forwarded between standard servers playing the Paxos roles. P4xos requires a specific network topology with multiple programmable switches, which makes it hard to deploy outside specially-designed clusters. The large amount of messages exchanged between switches (and actual servers sitting behind the switches) means that P4xos neither minimizes latency, nor maximizes the usage of network links. For instance, the latency of P4xos exceeds 100 µs at 100 k consensus/s (compared to 33 µs when executing 2 million consensus/s in P4CE).

Waverunner [21] proposes to accelerate a replication protocol by offloading the fast path of the protocol in FPGA cards located in the leader and the replicas. Waverunner accelerates the protocols in hardware, but, like Mu [1], it divides the network capacity by the number of replicas. With P4CE, we propose a complementary approach: we propose to offload the RDMA packet replication inside the network by leveraging a programmable switch. Combining a programmable switch, as in P4CE, with FPGA cards, as in Waverunner, would further increase performance, but evaluating the combination requires specific hardware which is not at our disposal.

FastCast [22] proposes to use a switch to perform efficient multicast of IPv4 messages. NOPaxos [23] simulated the use of programmable switches to further tag the multicasted messages with a sequence number, in order to replace consensus with network ordering. HovercRaft [24] uses programmable switches to speed up the multicasting of R2P2 messages. P4CE builds on these ideas and, crucially, enhances multicasting with in-network aggregation of acknowledgments to allow for transparently-replicated RDMA connections. Transparent replication of RDMA connections is needed to run the whole *communication* part of the consensus in the network, and is key to the high throughput and low latency of P4CE.

Belocchi *et al*. [25] have proposed to use SmartNICs to accelerate Paxos. Hyperloop [26] uses SmartNICs to offload the handling of persisting transactions on multiple replicas to the network card. These approaches are useful to reduce CPU overheads on the leader, but do not eliminate the bottleneck caused by the leader's network link.

NetLR [27] implements a leaderless replication protocol in a Tofino. Because NetLR implements a custom two-sided protocol, it is roughly 100x slower than P4CE, which leverages on-sided RDMA access.

Programmable switches have also been used to accelerate various workloads: from machine learning [28], [29], [30], [31], [32], map reduce tasks [33], software-defined network functions [34], future Internet architecture (SCION [35]), *etc*. P4SC [36] is a framework to implement service function chains into P4-enabled devices and execute common network functions (*i.e*., NAT, firewalling, L2/L3 forwarding, load-balancing) at line speed in the data plane. Wang *et al*. [37] propose an IoT framework to aggregate small network packets into large ones and to disaggregate them later. TEA [38] extends the limited memory of programmable switches by using the DRAM of remote servers by initiating RDMA connections from the switch itself. Rather than such routing-level services, P4CE is optimized to support efficient replication protocols.

b) Reducing bandwidth requirements of the leader: Multiple protocols have been proposed to reduce the network capacity needed by leaders in consensus protocols. Most of these papers rely on replicas forwarding packets between each other, usually following a ring topology [5], [6]. These protocols offer suboptimal latency, and require active replication actions from each replica, adding CPU overheads on the replicas, or requiring the use of SmartNICs.

c) Accelerating workloads using RDMA: Using RDMA is key to the low latency and high throughput of P4CE. P4CE follows the logic of Mu [1], which uses RDMA permissions to ensure that, at a given time, a single leader can write to the replicas' logs. Other RDMA-accelerated consensus protocols have been proposed. Similarly to Mu, APUS [2] pushes messages in replicas' logs using RDMA `write` commands. Replicas acknowledge having received the messages by sending other RDMA `write` commands. Velos [3] leverages RDMA compare-and-swap operations to allow consensus to be achieved even when multiple competing leaders try to write to the logs of the replicas. Sift [4] separates replicas in CPU nodes that store non-persistent state and storage nodes, modified using RDMA. In this paper, we focused on improving Mu [1] because it outperforms the other protocols, but the shared memory interface we implemented at the switch level can be used to improve the performance of all these protocols by transparently multicasting and aggregating RDMA operations and their results.

d) Changing and extending RDMA: RDMA multicast has been implemented in software libraries [39], [40]. These libraries extend the standard RDMA libraries to make it

possible to send data to multiple replicas. In RDMC [39], multicasting is done in multiple hops and using multiple point-to-point requests: the leader manually replicates data on a set of replicas that then re-send the data to other replicas, allowing dissemination over a large group of replicas. In RamCast [40], the leader coordinates replicas to give the illusion of an atomic broadcast. None of these software implementations offer latency-optimal or throughput-optimal replication because they are limited by the network capacity and CPU of the servers doing manual replication of the data.

Researchers also explored the possibility of adding support for multicast directly in RDMA, most notably by allowing replicas to accept and process the exact same RDMA packet [41], which would allow a passive optical cross-connect fabric to do the duplication of packets. Such extensions would simplify the design of the scattering performed by P4CE, but not the aggregation of ACKs.

VII. Conclusion

We have proposed the first RDMA-based consensus protocol able to achieve consensus in a single round trip at line speed. We have shown that decoupling decision and communication brings significant throughput increase, while reducing the latency of reaching a consensus.

The idea of P4CE is deceptively simple, but yields important performance gains in practice. Our results demonstrate that P4CE allows consensus to scale across multiple replicas, regardless of the size of the exchanged values. We demonstrate up to $4\times$ better throughput and lower latency than the state-of-the-art consensus protocols.

References

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi, "Microsecond consensus for microsecond applications," in *OSDI*. USENIX Association, 2020, pp. 599–616.

[2] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: fast and scalable paxos on RDMA," in *SoCC*. ACM, 2017, pp. 94–107.

[3] R. Guerraoui, A. Murat, and A. Xygkis, "Velos: One-sided paxos for RDMA applications," *CoRR*, vol. abs/2106.08676, 2021.

[4] M. Kazhamiaka, B. N. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee, "Sift: resource-efficient consensus with RDMA," in *CoNEXT*. ACM, 2019, pp. 260–271.

[5] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 5:1–5:32, 2010.

[6] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: Devouring the communication bottlenecks in distributed consensus," in *SIGMOD Conference*. ACM, 2021, pp. 235–247.

[7] InfiniBand Trade Association, *InfiniBand Architecture Specification Volume 1*, 2020.

[8] Broadcom Inc. (2022) RDMA over Converged Ethernet feature in Ethernet NIC controllers. [Online]. Available: https://techdocs.broadcom.com/us/en/storage-and-ethernet-connectivity/ethernet-nic-controllers/bcm957xxx/1-0/introduction/features_27/rdma-over-converged-ethernet--roce.html

[9] Intel Corporation. (2021) Which Intel Ethernet network adapters support iWARP and RoCE v2? [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000031905/ethernet-products/700-series-controllers-up-to-40gbe.html

[10] NVIDIA Corporation. (2022) ConnectX SmartNICs. [Online]. Available: https://www.nvidia.com/en-us/networking/ethernet-adapters/

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[12] Intel Corporation. (2022) Intel Tofino intelligent fabric processors. [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf

[13] The P4.org Architecture Working Group, *$P4_{16}$ Portable Switch Architecture (PSA)*, 2018. [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.1.0.pdf

[14] Intel Corporation, "Open Tofino," 2023. [Online]. Available: https://github.com/barefootnetworks/open-tofino

[15] P. Biondi *et al.* (2023) Scapy. [Online]. Available: https://scapy.net/

[16] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *HotNets*. ACM, 2017, pp. 150–156.

[17] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *NSDI*. USENIX Association, 2016, pp. 425–438.

[18] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: consensus at network speed," in *SOSR*. ACM, 2015, pp. 5:1–5:7.

[19] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a network service," *IEEE/ACM Trans. Netw.*, vol. 28, no. 4, pp. 1726–1738, 2020.

[20] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, 2016.

[21] M. Alimadadi, H. Mai, S. Cho, M. Ferdman, P. A. Milder, and S. Mu, "Waverunner: An elegant approach to hardware acceleration of state machine replication," in *NSDI*. USENIX Association, 2023, pp. 357–374.

[22] G. Berthou and V. Quéma, "FastCast: A throughput- and latency-efficient total order broadcast protocol," in *Middleware*, ser. Lecture Notes in Computer Science, vol. 8275. Springer, 2013, pp. 1–20.

[23] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *OSDI*. USENIX Association, 2016, pp. 467–483.

[24] M. Kogias and E. Bugnion, "HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *EuroSys*. ACM, 2020, pp. 25:1–25:17.

[25] G. Belocchi, V. Cardellini, A. Cammarano, and G. Bianchi, "Paxos in the NIC: hardware acceleration of distributed consensus protocols," in *DRCN*. IEEE, 2020, pp. 1–6.

[26] D. Kim, A. S. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *SIGCOMM*. ACM, 2018, pp. 297–312.

[27] G. Kim and W. Lee, "In-network leaderless replication for distributed data stores," *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1337–1349, 2022.

[28] A. Feng, D. Dong, F. Lei, J. Ma, E. Yu, and R. Wang, "In-network aggregation for data center networks: A survey," *Comput. Commun.*, vol. 198, pp. 63–76, 2023.

[29] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *NSDI*. USENIX Association, 2021, pp. 785–808.

[30] S. Liu, Q. Wang, J. Zhang, Q. Lin, Y. Liu, M. Xu, R. C. C. Cheung, and J. He, "Netreduce: Rdma-compatible in-network reduction for distributed DNN training acceleration," *CoRR*, vol. abs/2009.09736, 2020.

[31] C. Jia, J. Liu, X. Jin, H. Lin, H. An, W. Han, Z. Wu, and M. Chi, "Improving the performance of distributed tensorflow with RDMA," *Int. J. Parallel Program.*, vol. 46, no. 4, pp. 674–685, 2018.

[32] S. Liu, Q. Wang, J. Zhang, W. Wu, Q. Lin, Y. Liu, M. Xu, M. Canini, R. C. C. Cheung, and J. He, "In-network aggregation with transport transparency for distributed training," in *ASPLOS (3)*. ACM, 2023, pp. 376–391.

[33] G. Chen, G. Zeng, and L. Chen, "P4COM: in-network computation with programmable switches," *CoRR*, vol. abs/2107.13694, 2021.

[34] K. Zhang, D. Zhuo, and A. Krishnamurthy, "Gallium: Automated software middlebox offloading to programmable switches," in *SIGCOMM*. ACM, 2020, pp. 283–295.

[35] J. de Ruiter and C. Schutijser, "Next-generation internet at terabit speed: SCION in P4," in *CoNEXT*. ACM, 2021, pp. 119–125.

[36] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4SC: towards high-performance service function chain implementation on the p4-capable device," in *IM*. IFIP, 2019, pp. 1–9.

[37] S. Wang, C. Wu, Y. Lin, and C. Huang, "High-speed data-plane packet aggregation and disaggregation by P4 switches," *J. Netw. Comput. Appl.*, vol. 142, pp. 98–110, 2019.

[38] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "TEA: enabling state-intensive network functions on programmable switches," in *SIGCOMM*. ACM, 2020, pp. 90–106.

[39] J. Behrens, S. Jha, K. Birman, and E. Tremel, "RDMC: A reliable RDMA multicast for large objects," in *DSN*. IEEE Computer Society, 2018, pp. 71–82.

[40] L. H. Le, M. Eslahi-Kelorazi, P. R. Coelho, and F. Pedone, "RamCast: RDMA-based atomic multicast," in *Middleware*. ACM, 2021, pp. 172–184.

[41] K.-W. Leong, Z. Li, and Y. L. Liu, "Reliable multicast using remote direct memory access (RDMA) over a passive optical cross-connect fabric enhanced with wavelength division multiplexing (WDM)," *APSIPA Transactions on Signal and Information Processing*, vol. 8, p. e25, 2019.