Electrode: Accelerating Distributed Protocols with eBPF

Yang Zhou* Harvard University
Zezhou Wang* Peking University
Sowmya Dharanipragada Cornell University
Minlan Yu Harvard University

Abstract

Implementing distributed protocols under a standard Linux kernel networking stack enjoys the benefits of load-aware CPU scaling, high compatibility, and robust security and isolation. However, it suffers from low performance because of excessive user-kernel crossings and kernel networking stack traversing. We present Electrode with a set of eBPF-based performance optimizations designed for distributed protocols. These optimizations get executed in the kernel before the networking stack but achieve similar functionalities as were implemented in user space (e.g., message broadcasting, collecting quorum of acknowledgments), thus avoiding the overheads incurred by user-kernel crossings and kernel networking stack traversing. We show that when applied to a classic Multi-Paxos state machine replication protocol, Electrode improves its throughput by up to 128.4% and latency by up to 41.7%.

1 Introduction

Distributed protocols such as Paxos [37] for state machine replication are important building blocks for highly-available distributed applications. For example, Google's Chubby [6] uses a variant of classic Paxos [37] and Multi-Paxos [36] to implement a highly-available lock service, powering their business-critical GFS [16] and Bigdata [7] applications. Google's globally-distributed database Spanner [8] and Microsoft's data center management tool Autopilot [22] also run Paxos protocols to maintain their high availability.

Existing high-performance implementation of distributed protocols tends to be radical and not readily-deployable. DPDK-based kernel-bypass approaches [27, 79] allow direct access to the underlying NIC hardware, but require application developers to build their own networking stack and maintain compatibility with the evolving kernel networking stack [75]. DPDK also dedicates CPU cores to busily poll the network interface for I/O competition, sacrificing CPU resources and wasting energy during low I/O loads. This is especially a problem for embedded devices [51, 60, 70] where CPU resources are rare. Other approaches co-design specialized distributed systems with niche network hardware including RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [25]. These advanced hardware devices are not widely available in today's cloud environments, and systems built on top of them tend to be difficult to design, implement, and deploy [27].

*Equal contribution

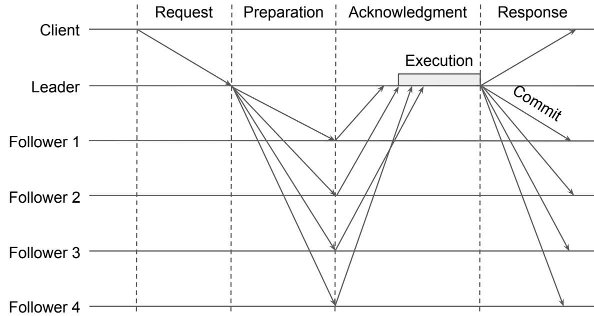
Instead, we would prefer the widely-deployed and well-maintained standard kernel networking stack that also provides load-aware CPU scaling and strong security and isolation among different applications [5, 59]. However, implementing distributed protocols under the standard kernel networking stack often gives poor performance. The root causes are the high packet processing overhead in the kernel networking stack and heavy communications in distributed protocols. Our measurement shows that over half of CPU time is spent on the kernel networking stack in a typical Paxos deployment (§2); such overhead is mainly caused by user-kernel crossings (and associated context switches) and traversing the kernel networking stack. Moreover, when using a classic leader-based Multi-Paxos protocol [43, 54] to implement state machine replication, e.g., with five replicas, processing a single request would require the leader node to send/receive *fourteen* messages in total (see Figure 1a), suffering from the kernel stack overhead fourteen times[1].

In this paper, we focus on accelerating Paxos protocols inside data centers by offloading protocol operations to the kernel via eBPF (i.e., extended Berkeley Packet Filter) [46, 49]. eBPF allows safely executing customized yet constrained functions inside the kernel at various locations. Similar to kernel bypass, the offloaded operations get executed immediately after the NIC driver receives the packet, without user-kernel crossing and kernel networking stack traversing. Unlike kernel bypass, eBPF is an OS-native mechanism such that eBPF-offloaded operations do not sacrifice security and isolation properties while amenable to load-aware CPU scaling without busy-polling.

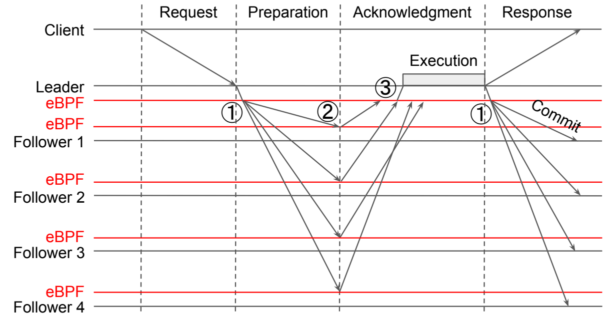
The key challenge is, given the constrained programming model of eBPF, *which parts of Paxos protocols to offload that can greatly reduce kernel stack overhead while being implementable and efficient in eBPF*. Note that the eBPF subsystem requires every offloaded function to be statically verified to guarantee kernel security, which only allows limited instructions, bounded loops, static memory allocation, etc.

Our insight is that common operations of Paxos protocols, e.g., message broadcasting and waiting on quorums, incur large kernel stack overhead, but are naturally offloadable by existing eBPF programming capacity. For example, Paxos protocols require a leader node to broadcast preparation messages to follower nodes; if implemented using multiple `sendto()` syscalls conventionally, it would incur multiple user-kernel

[1] Linux io_uring [1] can reduce user-kernel crossings, but cannot reduce kernel stack traversing (see §8 for details).



(a) The Multi-Paxos/Viewstamped Replication protocol.



(b) Electrode-accelerated Multi-Paxos/Viewstamped Replication.

Figure 1: Normal case execution of the leader-based Multi-Paxos/Viewstamped Replication protocol vs. Electrode-accelerated one with 5 replicas. Electrode offloads ①: message broadcasting (§4.1), ②: fast acknowledging (§4.2), and ③: wait-on-quorums (§4.3) to eBPF to reduce the kernel networking stack overhead.

crossings and kernel networking stack traversing. Instead, eBPF has a `bpf_clone_redirect()` [45] function that enables us to clone an in-kernel packet buffer multiple times and send them to different destinations; this eBPF-based message broadcasting only needs one user-kernel crossing and one kernel networking stack traversing. Besides broadcasting, we also utilize eBPF to reduce unnecessary wake-ups of userspace applications when waiting on quorums, and optimize how follower nodes handle preparation messages by early acknowledging before entering the kernel networking stack. The final result of these three eBPF-based optimizations is Electrode[2] (Figure 1b). When applying Electrode to a classic leader-based Multi-Paxos protocol, it achieves up to 128.4% higher throughput and 41.7% lower latency. This translates into up to 112.9% higher throughput and 19.3% lower latency for a Paxos-based transactional replicated key-value store.

Electrode has some limitations: it currently targets protocols implemented in UDP and relies on application-level retransmission to handle packet loss. This works well for Paxos protocols whose requests are usually small enough to fit into a single packet, and data center environments where packet loss is rare [28, 61].

2 Background

2.1 Consensus Protocols

Distributed protocols that coordinate and synchronize among a collection of nodes have become an indispensable part of the modern data center application stack. Storage systems in data centers replicate data for fault tolerance and availability. For instance, Berkeley-DB [55] uses a consensus protocol to replicate its logs over a set of distributed replicas. Transactional storage systems like H-Store [71] and Spanner commit their updates to multiple replicas in order to be more failure resilient. At the heart of most replication-based systems is a consensus protocol [36, 37, 43, 54] that ensures that operations execute in a consistent manner across all replicas.

[2]Electrode is a Pokémon that has a high speed score.

Here, we consider a set of nodes either functioning as clients or replicas. Clients are the users of a particular application-level service hosted by a collection of replicas. It should also be noted here that clients could often just be other servers within the same data center. Clients submit requests to one or more replicas, which triggers a round of agreement to occur. Paxos is a common protocol that is used to obtain an agreement in the presence of node and network failures.

Since applications often need to reach agreements on many client requests, servers use agreement protocols like Paxos to implement a state machine-based abstraction that requires all the replicas to process the exact same set of client requests in the same order. This log-based state machine abstraction is often optimized by the use of a leader. In a leader-based protocol, all the instances of agreement on client requests are mediated through the leader and the leader also dictates the order of the log.

In Figure 1a, we have an example of VR (Viewstamped Replication), a leader-based Multi-Paxos protocol that uses Paxos for running agreements on individual requests. The leader here is responsible for ordering all client requests by assigning sequence numbers to them, and the followers (non-leader nodes) are responsible for responding to the leader and applying all the updates in the order in which they're sequenced by the leader.

The leader is also responsible for initiating agreement by sending out a preparation message to all the other replicas. The leader then waits for a quorum of acknowledgments from all the other replicas before broadcasting a commit message to all the replicas. A successful iteration of this two-round protocol ensures that all non-failed replicas have the client's request. And the sequence number assigned by the leader determines the order in which all the replicas process this client's request. This pattern of broadcasting and waiting on quorums is common in many distributed protocols [38, 39, 80].

To gain more insights into the performance of the Multi-Paxos/VR protocol under the standard Linux kernel networking stack, we measure the CPU time breakdown of the leader node, shown in Table 1. There is 44.7% +

Function Name	Description	% CPU
`__libc_sendto()`	User function to send packets.	44.7
⊢ `sock_sendmsg()`	Kernel function to send packets.	32.2
| ⊢ `__alloc_skb()`	Allocate `sk_buff` for packets.	4.5
| ⊢ `dev_queue_xmit()`	Transmit `sk_buff`.	6.8
| ⊢ bookkeeping	For sock, IP, and UDP layers.	**20.9**
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**12.5**
`__libc_recvfrom()`	User function to recv packets.	11.8
⊢ `sock_recvmsg()`	Kernel function to recv packets.	5.7
⊢ user-kernel crossing	Interrupt, mode switching, etc.	**6.1**

Table 1: CPU time breakdown for the leader node when running the Multi-Paxos/Viewstamped Replication protocol with 5 replicas. See §7 for measurement setup.

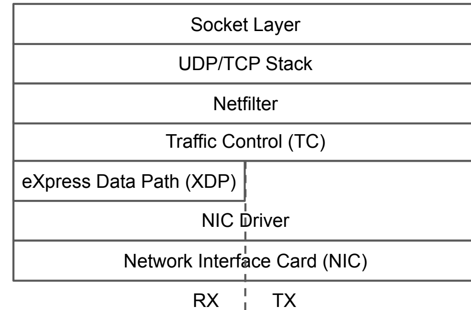


Figure 2: Linux kernel networking stacks and eBPF XDP/TC hooks.

$11.8\% = 56.5\%$ of time spent on the `__libc_sendto()` and `__libc_recvfrom()` functions, while $20.9\% + 12.5\% + 6.1\% = 39.5\%$ of time spent on user-kernel crossing and kernel networking stack bookkeeping. These numbers concrete our previous motivations that implementing distributed protocols under kernel networking stack incurs significant overhead on user-kernel crossings and kernel stack traversing (while eBPF can potentially save them).

2.2 eBPF and Hooks

BPF (i.e., Berkeley Packet Filter) [49] enables user-space applications to customize packet filtering in the kernel. A BPF program, written in some predicates on packet fields, is triggered by the kernel event that a packet arrives at a NIC driver. Once triggered, the BPF program will run inside a kernel virtual machine with limited registers and scratch memory, and a reduced instruction set [49]. For example, the well-known *tcpdump* [20] command-line packet analyzer is based on BPF.

eBPF extends the BPF by increasing the number of registers and adding stack memory. The increased number of registers and stack memory enable the eBPF program to execute more complex operations—the developers can use a C-like language to express customized operations. This C-like code is compiled into an eBPF bytecode by the Clang/LLVM toolchain and runs inside the kernel virtual machine via just-in-time compilation.

eBPF also introduces various powerful in-kernel data structures called *eBPF maps*, which, paired with various helper functions, are used to store and maintain states across multiple triggering of eBPF programs. Example eBPF maps include array, per-CPU arrays, queues, stacks, and hashMaps [46]. These maps are also used to communicate among different eBPF programs and between eBPF programs and user-space processes. Each eBPF map can be identified by a `map_path` through the file system, e.g., `/sys/fs/bpf/<map_name>`, and user-space processes can access a map based on its path.

The kernel events that can trigger eBPF programs are called *eBPF hooks*. There are many hooks existing in Linux kernels and various device drivers, such as hooks in NIC drivers right after it receives a packet. User-space applications can attach eBPF programs to these eBPF hooks to customize the handling of corresponding kernel events.

Constrained programming model: An eBPF program needs to go through strict verification by an in-kernel eBPF verifier before attaching to an eBPF hook and running inside the kernel. The verification process does a static sanity check to make sure the eBPF program does not have out-of-bounds memory access (i.e., safety) and will always terminate (i.e., liveness). The verifier basically enumerates all possible cases of every conditional branch and loop to make sure every execution path meets the safety and liveness requirements. Because the verification tends to be time-consuming, each eBPF program can only contain up to 1 million instructions. For a larger eBPF program, the developer needs to split it into multiple smaller eBPF programs and uses *tail calls* to let one eBPF program call another one in a continuation manner.

Because of the strict verification process, dynamical memory allocation is not supported in eBPF programs; instead, eBPF programs can only rely on eBPF maps with capacity *specified statically* to maintain in-kernel states.

Due to these limitations, eBPF is commonly used in kernel tracing, profiling, and monitoring [3, 63] and L2-L4 low-level packet processing such as load balancing [14].

XDP (eXpress Data Path) [21, 64] technique implements an in-kernel eBPF hook that enables attached eBPF programs to process RX packets directly out of the NIC driver (Figure 2). Such processing gets triggered before any `sk_buff` [31] allocation or entering software socket queues, thus bypassing any higher-level networking stacks (e.g., UDP, TCP, Socket). XDP-based packet processing normally achieves comparable throughput and latency as DPDK-based kernel-bypass packet processing [21].

TC (Traffic Control) [47] is another important layer/hook which locates right after the XDP (Figure 2). In the TC layer, the `sk_buff` data structure has already been allocated by the kernel networking stack, thus the performance of TC-based packet processing will be slightly worse than XDP. However, the TC hook allows attached eBPF programs to process both RX and TX packets and manipulate the packet `sk_buff`. For

example, one can clone the `sk_buff` for a TX packet and thus implements packet broadcasting in the TC layer.

3 Electrode Overview

Electrode is a framework for offloading Paxos protocols under kernel networking stack to in-kernel eBPF programs to reduce user-kernel crossings and kernel networking stack traversing. Electrode has two goals in designing its eBPF offloads: 1) largely reducing kernel stack overhead to improve performance, and 2) carefully partitioning user- and kernel-space functionalities to keep offloads implementable and efficient inside the eBPF subsystem.

To achieve the first goal, Electrode carefully extracts generic and performance-critical fast-path operations from Paxos protocols to offload to the eBPF. As shown in Figure 1b, Electrode offloads message broadcasting (§4.1), fast acknowledging (§4.2), and wait-on-quorums (§4.3). These operations, if purely implemented in the user space, would involve many user-kernel crossings and kernel stack traversing, causing significant kernel stack overhead as shown in §2. Once implemented in the eBPF, message broadcasting allows the leader node to efficiently send preparation and commit messages to multiple follower nodes, by cloning and sending packets in the kernel; fast acknowledging enables follower nodes to buffer preparation messages in the kernel, and quickly respond to the leader node without involving user-space processes; wait-on-quorums lets the leader node eBPF program wait for a quorum number of acknowledgments from follower nodes, and only notify user-space processes once. Moreover, to simplify how user-space applications use these eBPF-based accelerations, Electrode further designs a set of user-space APIs (Table 2). Each API corresponds to one operation that Electrode offloads to the eBPF, and is used to invoke the offloaded function or retrieve eBPF processing results.

To achieve the second goal, Electrode keeps complicated slow-path operations of Paxos protocols in the user space. Specifically, Electrode leaves the procedures of failure recovery and handling message loss/reordering (i.e., gap agreement) to user-space applications, using similar mechanisms as VR [43] and NOPaxos [40]. These procedures involve accessing dynamic ranges of memory, which is hard to implement in eBPF under the static verification (see §8 for details).

Overall, Electrode has the following workflow: first, user-space applications attach eBPF programs to various hook locations corresponding to a network interface; then, user-space applications use Electrode APIs to invoke eBPF-offloaded functions or retrieve eBPF processing results; finally, the eBPF programs intercept and process target packets in the kernel without going through the networking stack or user-space applications (i.e., Paxos protocols in our case). Electrode targets accelerating the handling of messages that can fit into one ethernet packet (i.e., up to 9KB for jumbo frames). This is well-suited for locks, barriers, and configuration parameters [25, 78] that Paxos protocols commonly maintain. Non-target packets still go through the stack and reach user-space applications, without impacting applications' other operations or protocol semantics.

Finally, we note that Electrode does not aim to offload every operation of Paxos protocols to the eBPF, because of eBPF's constrained programming model vs. the diverse set of operations that Paxos protocols and related services could have. For example, currently, Electrode does not offload client-facing request/response handling. There are two reasons: 1) Paxos clients normally serialize/deserialize their requests using widely-used libraries such as protocol buffers [19]; however, parsing or constructing protocol buffers is difficult in eBPF, because it involves complex pointer arithmetics and conditional branches which cannot easily pass the eBPF verifier. 2) client-facing requests/responses are normally embedded into application-level services like the Chubby lock service [6], but it is hard and inefficient to implement them in eBPF because of the strict eBPF verifier and the lack of dynamic memory allocation. We discuss more on Electrode's offloading decisions in §8.

4 Electrode Designs

4.1 Message Broadcasting in TC

In Paxos protocols, one-to-all message broadcasting is widely used. For example, 1) the leader node sends preparation messages to all follower nodes, and 2) (after receiving enough acknowledgments from followers) the leader node sends commit messages to all follower nodes.

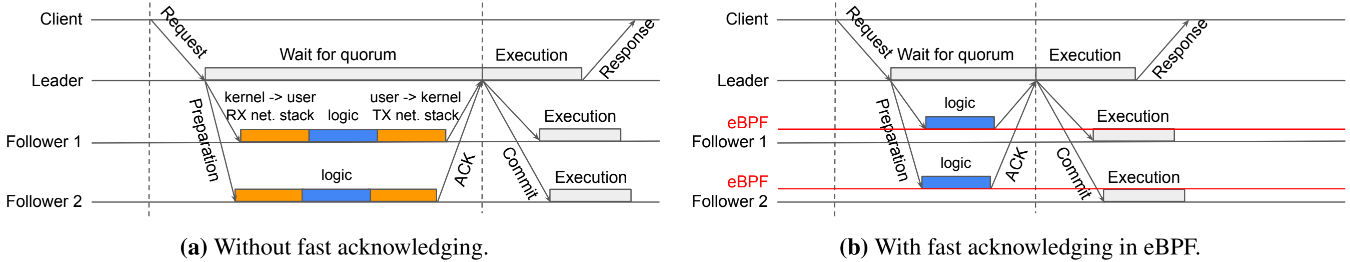
To implement the above message broadcasting, the most common way is sending the same message multiple times in the user space to different destinations. However, the overhead (i.e., user-kernel crossing and kernel networking stack traversing) of this implementation on the leader node increases linearly as the number of followers increases, while the overhead on each follower node remains constant. Thus, the leader node essentially becomes the system bottleneck, e.g., Table 1 has shown that 44.7% of CPU time is spent on sending messages on the leader node.

An alternative implementation is to use IP multicast [42, 68, 77]. However, IP multicast normally requires support from the underlying network switches (e.g., storing a large number of multicast group-table entries for the whole network topology) [68, 77] or considerable modifications of the Linux networking stack [42].

Electrode approach: Electrode provides a flexible host-based broadcasting solution by utilizing eBPF on the TC hook. Here, we require the eBPF program that implements broadcasting operations to attach to the TC hook, because only the TC hook can intercept and process outgoing packets (§2.2). After attaching the eBPF program, user-space applications can call the `elec_broadcast()` function shown in Table 2 with specified `sock_fd`, message, and a list of destination IPs to broadcast the message to these destinations through the socket.

Function Name	Arguments	Output	Description
elec_broadcast	sock_fd, message, {dst_ips}	status	Broadcasts <message> to all destinations through <sock_fd>
elec_poll_message	map_path	messages	Polls buffered messages from an eBPF-maintained in-kernel ring buffer identified by <map_path>
elec_check_quorum	received_message	bool	Checks if <received_message> (acknowledgment) indicates quorum reaching

Table 2: Electrode user-space APIs.



(a) Without fast acknowledging.

(b) With fast acknowledging in eBPF.

Figure 3: Fast acknowledging in eBPF reduces Paxos request latency. This example follows Figure 1, but omits followers 3 and 4 for brevity.

Under the hood, the eBPF program makes clones of the message packet using the `bpf_clone_redirect()` [45] helper function, modifies the destination addresses of cloned packets accordingly, and sends these packets out. The benefit of cloning packets and broadcasting in the kernel compared with sending the same message multiple times in the user space is that we only need to cross the user-kernel boundary and traverse the UDP and socket layer once.

Handling message loss: Electrode relies on application-level timeout and retransmission to handle message loss, similar to modern RPC-based applications [13, 69]. Specifically, if the leader node does not receive a response after a certain time of sending a request, it will resend the request; once a request experiences several timeouts, the leader node will mark the destination node as dead and start Paxos failure recovery. An alternative approach to handling message loss is doing retransmission in the kernel, which could save user-kernel context switching overheads, but such savings become marginal as packet loss happens rarely in data centers [28, 61]; it would also involve complex message buffer management in kernel/eBPF, hurting performance.

4.2 Fast Acknowledging in XDP

As shown in Figure 3a, a significant portion of Paxos request latency comes from the round-trip delay between the leader node and follower nodes. Note that the ACK messages in this figure mean Paxos protocol acknowledgments, not TCP acknowledgments. For Paxos protocols under the kernel networking stack, this round-trip delay includes not only physical propagation and transmission delay, but also the delay caused by the kernel networking stack (i.e., user-kernel crossing and networking stack traversing). As the fabric latency of nowadays data center network reaches a few tens of microseconds [48] or sub-ten microseconds [18, 27], the latency of the kernel networking stack, which is also around sub-ten microseconds [59], becomes non-negligible.

Electrode approach to reducing the Paxos request latency is to optimize the preparation handling in follower nodes by directly buffering the preparation messages into an in-kernel log and early acknowledging to the leader node. At the same time, user-space applications asynchronously poll and consume the buffered messages from the log, using the `elec_poll_message()` function shown in Table 2. Under the hood, the function calls a corresponding eBPF syscall to poll messages in batches, amortizing kernel crossing overhead. This asynchrony does not break the correctness of Paxos protocols because as long as a preparation message gets buffered into the log, it will be eventually processed by the user-space Paxos protocols, and the message processing order has been specified by the sequence number assigned by the leader node. Figure 3b shows that this approach removes *two* user-kernel crossings and networking stack traversing from the critical path of the Paxos request.

Note that not every preparation message can be handled using fast acknowledging; in some non-critical path cases (e.g., message loss/reordering, and node failure) where the eBPF program cannot handle because of its constrained programming model, our eBPF program can detect them and directly forward preparation messages to user-space Paxos protocols (detailed in §6).

In-kernel log implementation: The in-kernel log temporally stores incoming early-acknowledged preparation messages, which are polled and consumed by user-space applications concurrently. To implement this in-kernel log, we use a special eBPF map named `BPF_MAP_TYPE_RINGBUF` [30] (introduced from Linux kernel 5.8). This map implements an efficient multi-producer single-consumer (MPSC) ring buffer using shared memory and a lightweight spinlock, where we can have multiple writers in eBPF and one reader in user space. Based on our measurement, the time of pushing a preparation message into the ring buffer is roughly equal to memcpying this message, in cases without any lock contention. Note

that the in-kernel ring buffer also has a fixed size, because eBPF does not support dynamic memory allocation; in case it becomes full, the eBPF program can detect them and directly forward preparation messages to user-space applications.

4.3 Wait-on-Quorums in TC + XDP

Another common operation in Paxos protocols is the leader node waiting for a quorum number of acknowledgments (ACKs) from follower nodes (i.e., wait-on-quorums). Assume there are $2f+1$ replicas including one leader node and $2f$ follower nodes. In most Paxos protocols, once the leader collects f ACKs from different follower nodes, the Paxos request is considered *committed*.

Conventionally, wait-on-quorums is implemented by the user-space applications that receive all ACKs and count towards the quorum number. However, each acknowledgment handling incurs the overhead of the user-kernel crossing and traversing the kernel networking layer. The total overhead of handling all ACKs is linear to the number of follower replicas (i.e., $2f$). Moreover, among these $2f$ ACKs, only the first f ones are required to commit a Paxos request.

Electrode approach: Electrode moves the leader-side wait-on-quorums operations to the eBPF, requiring only one user-kernel crossing and one networking stack traversing. Electrode maintains an array of bitsets (and other metadata) in eBPF, each of which indicates whether a Paxos request has reached the quorum. Electrode only forwards ACK messages that indicate reaching the quorum to the user-space applications, while dropping others. Electrode maps each Paxos request to a specific bitset by using the unique increasing sequence number assigned by the leader node (§2). Note that we use the bitset instead of a counter to check if the quorum gets reached; this is because a timed-out preparation request could cause duplicate ACK messages from follower nodes, and we want to avoid double counting.

Electrode maintains the bitset setting and clearing (i.e., zeroing out) operations through two eBPF programs hooked at TC and XDP layers, respectively. The TC-hooked eBPF program intercepts each outgoing preparation message and clears the indexed bitset, while the XDP-hooked eBPF program intercepts each incoming ACK message from follower nodes and sets the bit corresponding to the follower node's index in replicas.

As shown in Listing 1, the `tc_ebpf` function/program intercepts each outgoing preparation message and clears a specific bitset indexed by the sequence number in each message. Line 6 checks if it is the first time to intercept a preparation message corresponding to this Paxos request, by comparing the `seq` stored along this bitset and the `seq` extracted from the message; if so, it updates the stored `seq` in the array and clears the bitset that may have been used by previous Paxos requests (line 17-18).

The `xdp_ebpf` program intercepts each incoming ACK message, updates the indexed bitset, drops most of the ACK packets, and only forwards packets to user-space applications that indicate reaching quorum or array overflow (explained in the next paragraph). Lines 17-18 check if this bitset corresponds to the `seq` in the ACK message, and set the proper bitset bit if so. Line 19 further checks if this ACK message reaches the quorum: if so, lines 20-21 will mark the packet as quorum-reaching and forward it to user-space applications; otherwise, line 22 just drops the packet. Once the user-space applications receive a quorum-reaching packet—checked by calling the `elec_check_quorum()` function shown in Table 2, it can directly consider this Paxos request as committed.

```
1  # Processing outgoing preparation message
2  # pkt: the packet of the message
3  # seq: unique increasing sequence number (from pkt)
4  def tc_ebpf(pkt, seq):
5      idx = seq % array_length
6      if array[idx].seq != seq
7          array[idx].seq = seq
8          array[idx].bitset.clear()
9      forward(pkt) # to follower node
10
11 # Processing incoming ACK message
12 # pkt   : the packet of the message
13 # seq   : unique increasing sequence number (from pkt)
14 # node_i: follower node index (from pkt)
15 def xdp_ebpf(pkt, seq, node_i):
16     idx = seq % array_length
17     if array[idx].seq == seq
18         array[idx].bitset.set(node_i)
19         if array[idx].bitset.count() == f
20             pkt.mark_quorum_reach(true)
21             forward(pkt) # to user-space application
22         else: drop(pkt)
23     else: # bitset overwritten by tc_ebpf
24         pkt.mark_quorum_reach(false)
25         forward(pkt)
```

Listing 1: Maintaining the fixed-length bitset array to achieve wait-on-quorums in eBPF. Each bitset operation is also protected by a spinlock; we omit it here for simplicity.

Handling array overflow: In some cases, a bitset might be overwritten by the `tc_ebpf` because of the fixed size of the bitset array. `xdp_ebpf` detects such array overflow in lines 17&23; once detected, lines 24-25 will mark the packet as *not*-quorum-reaching and forward it to user-space applications. Once the user-space applications receive a not-quorum-reaching packet, it resends the preparation messages to all follower nodes and waits for ACKs again. In practice, the leader node could limit the number of in-flight preparations while provisioning a large bitset array, such that the array overflow does not normally happen.

RSS: Electrode supports RSS (Receive-Side Scaling) which distributes incoming packets to different NIC queues and CPU cores. Specifically, Electrode has two receive-side optimizations: fast acknowledging and wait-on-quorums. For fast acknowledging, the eBPF programs in the follower node could maintain separate in-kernel ring buffers on different cores to avoid synchronization overhead during log append-

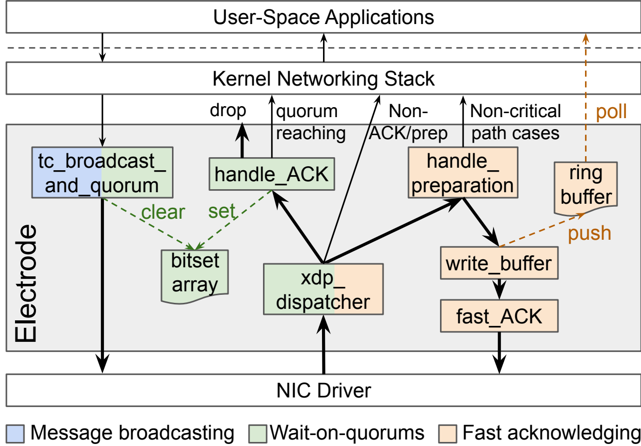


Figure 4: eBPF program structure of Electrode. The thickness of solid lines indicates traffic volume (the thicker, the higher).

ing, and use spinlocks to synchronize accesses to small shared in-kernel states (e.g., `ebpf_seq` in §6); the user-space applications asynchronously pull messages from all ring buffers, and process messages following the order specified by their embedded sequence numbers. For wait-on-quorums, the eBPF programs in the leader node could use atomic instructions to count how many ACKs it has received and check if the quorum is reached.

5 Electrode Implementation

Electrode is prototyped with six eBPF programs written in a restricted C language, and we utilize the Clang/LLVM toolchain for compiling source code to eBPF bytecode. These eBPF programs consist of 500 lines of C code in total. Application developers can also customize their own eBPF programs based on needs, e.g., only processing packets with a specific source port like [25]. Our prototype does not implement the RSS handling yet.

Figure 4 shows the structure of the six eBPF programs. One program can transfer its control flow to the next program via the eBPF tail call. We break the implementation into these six programs because of 1) avoiding breaking the instruction limits in the eBPF verifier (§2.2), and 2) modularity. In the following, we describe each program in detail.

- `tc_broadcast_and_quorum`: This program intercepts outgoing preparation messages. It implements the message broadcasting mechanism (§4.1) and the `tc_ebpf` function in Listing 1 for wait-on-quorums (§4.3). For broadcasting, we generate multiple clones of the preparation packets using the `bpf_clone_redirect()` [45] helper function.
- `xdp_dispatcher`: This program checks the types of incoming messages and calls corresponding message handlers. It only intercepts the ACK (only received on the leader node) and preparation (only received on follower nodes) messages, and calls the corresponding `handle_ACK` and `handle_preparation` programs. It directly forwards other types of messages to user-space applications.
- `handle_ACK`: This program implements the `xdp_ebpf` function in Listing 1 for wait-on-quorums (§4.3). In common cases, it drops most ACK messages, and only forwards the quorum-reaching ACK messages to user-space applications.
- `handle_preparation`: This program implements various checks to detect non-critical path cases where it should forward messages to user-space applications (§4.2). In normal cases (mostly), it will call `write_buffer` to begin `fast_ACK`.
- `write_buffer`: This program stores message/packet data into an in-kernel log for user-space applications to poll and consume. As mentioned earlier, We use the eBPF ring buffer [30] to implement the log data structure. This program then calls the `fast_ACK` program.
- `fast_ACK`: This program reuses and modifies the received packet buffer to create an ACK packet and sent it out. This requires swapping the src-dst IP addresses and filling the corresponding fields of the ACK message.

6 Apply Electrode to Multi-Paxos

Optimizing throughput: We apply the eBPF-based message broadcasting (§4.1) and wait-on-quorums (§4.3) mechanisms to the leader node in the Multi-Paxos protocol. This implies two throughput optimizations: 1) when the leader node sends out preparation messages to follower nodes, it relies on eBPF to broadcast these messages instead of sending them one by one; and 2) when the leader node is waiting for a quorum number of ACK messages from follower nodes, it only needs to process the quorum-reaching ACK message while the other ACK messages are pruned/dropped by the eBPF program. These two optimizations largely reduce the number of user-kernel crossings and kernel networking stack traversing, thus alleviating the CPU bottleneck on the leader node and improving system throughput.

Optimizing latency: We apply the eBPF-based fast acknowledging mechanism (§4.2) to each follower node in the Multi-Paxos protocol. In normal cases (e.g., without packet loss/reordering, and all nodes are alive), the preparation messages from the leader node are quickly buffered and acknowledged by the eBPF program in the follower nodes, bypassing both the kernel networking stack and the user-space Multi-Paxos protocol. This reduces the commit latency of each Multi-Paxos request by twice the time of user-kernel crossing and kernel networking stack traversing.

Detecting non-critical path cases in fast acknowledging: As mentioned in §4.2, there are some non-critical path cases in fast acknowledging where the eBPF program must detect them and forward the incoming packets to the user-space Paxos protocols. To understand why non-critical path cases happen and how to detect them, we first elaborate on the Multi-Paxos/VR protocol shown in §2, following the literature [43]. In the Multi-Paxos protocol, the leader node assigns

each Multi-Paxos request a unique and strictly increasing sequence number, `seq`. Each replica including both the leader node and follower nodes maintains locally a `view` number, a `status`, and its last observed `seq`; each message sent by a replica will piggyback these three variables. The `view` number indicates which (leader) election epoch this replica is in; the `status` indicates if this replica is during a leader election (`status_viewchange`), recovering (`status_recovering`), or normal state (`status_normal`). This protocol requires a follower node to only process a preparation message if the node is in the normal state, and the message has a matched `view` and strictly increasing `seq`; otherwise, the follower node needs to drop the message, or execute a complex view-change or state-transfer procedure [43, 54]. Therefore, the non-critical path cases for Multi-Paxos are:

1. the follower is during a leader election or recovering,
2. the follower receives a message with an unmatched `view` that is either (a) stale or (b) newer,
3. the follower receives a message with a non-strictly-increasing `seq` caused by message (a) loss/reordering or (b) duplication.

These cases only happen when replicas fail or join, or messages get lost/reordered, which is less common in data centers [27, 61].

To detect these non-critical path cases in eBPF, we maintain an `ebpf_status`, an `ebpf_view`, and an `ebpf_seq` variable in the eBPF program using the eBPF map. In particular, these three variables can be updated by the user-space Multi-Paxos protocols to reflect the current protocol state. Listing 2 shows the detection pseudocode. Line 5 detects case 1, and line 6 detects case 2(a); for these two cases, the eBPF program needs to drop the packet. Line 7 detects cases 2(b) and 3(a), and forwards the packet to the user space to execute the view-change or state-transfer procedure. For case 3(b), i.e., `msg_seq` < `ebpf_seq` + 1, the eBPF program function replies an ACK (line 11), because it could be a re-transmitted preparation message due to timeout.

Handling the cases 2(a)&3(a) in fast acknowledging is tricky, because it (i.e., forwarding packets to the user space for processing) involves the concurrency between the user-space protocols and the kernel-space eBPF program, while eBPF only supports map-based communication *but not synchronization* between the user and kernel. Our approach is to let the user-space protocols *detach* the eBPF program from the hook while executing the view-change or state-transfer procedure. Specifically, once a user-space protocol receives a preparation message corresponding to the case 2(a) or 3(a), it detaches the eBPF program, then it finishes the view-change or state-transfer procedure, next it updates the `ebpf_status`, `ebpf_view`, and `ebpf_seq` properly, and finally it reattaches the eBPF program. This guarantees the cases 2(a)&3(a) are exclusively handled by the user-space protocol, avoiding the synchronization between the user and kernel. An alternative approach to achieving the same effect as eBPF detach-reattach

```
1  # pkt     : the packet of the preparation message
2  # msg_view: view piggybacked by the pkt
3  # msg_seq : unique increasing sequence number (from pkt)
4  def detect_non_crit_path_cases(pkt, msg_view, msg_seq):
5      if (ebpf_status != status_normal): drop(pkt)
6      if (msg_view < ebpf_view): drop(pkt)
7      if (msg_view > ebpf_view or msg_seq > ebpf_seq + 1):
8          forward(pkt)
9      if (msg_seq == ebpf_seq + 1):
10         append_log(++ebpf_seq, pkt)
11     reply_ack(pkt)
```

Listing 2: Detecting non-critical path cases during fast acknowledging for Multi-Paxos. Assume the protocol works in a single core, in line with prior Paxos work [40, 44, 61].

is to use an eBPF map with a branch testing before any Electrode logic. The first packet in the non-critical path can update this map atomically and let all following packets directly go to the user-space application (i.e., closing Electrode optimizations); later, the user-space application can update this map to reopen Electrode optimizations.

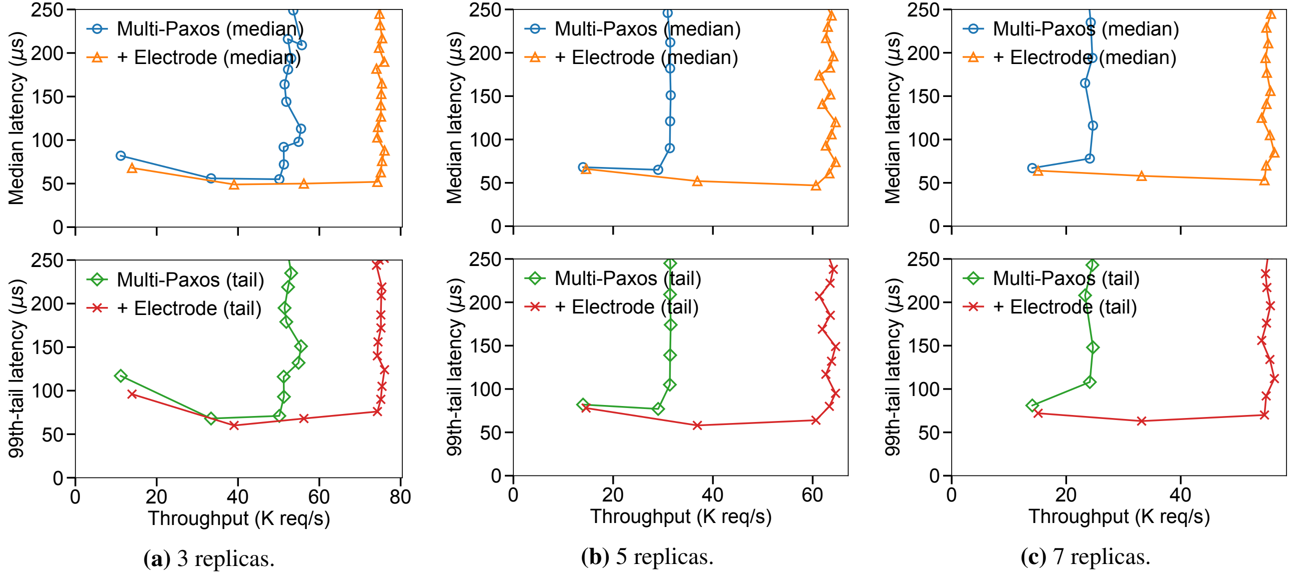
There are a few caveats: 1) After the user-space protocol detaches the eBPF program, it needs to poll the in-kernel ring buffer again, in case the eBPF program still appends a few messages to the ring buffer before detaching. Note that the eBPF map can outlive the eBPF program, as long as the user-space process holds a reference to it, because its lifetime is managed through reference counting [50]. 2) While the user-space protocol is setting the `ebpf_seq` value and is about to reattach the eBPF program, some preparation packets might just pass the eBPF hook location but have not been processed by the user-space protocol, e.g., queued in the socket layer. In this case, the user-space protocol actually has set a smaller `ebpf_seq` value in the map; once the eBPF program gets reattached, it will trigger more case 3(a) (lines 7&8). Our solution to this problem is: after the user-space protocol finishes the view-change or state-transfer procedure, it first sends a `stop_sending_preparation` message to the leader node to stop it from sending preparation messages, then it polls the socket to drain and process any queued packet, next it sets the proper `ebpf_seq` value, finally it sends a `resume_sending_preparation` message to the leader node to resume sending preparation messages, and reattaches the eBPF program. These two messages should be sent using reliable transport like TCP to handle packet loss.

Generalizability: Electrode's eBPF-based optimizations are generic to many more distributed protocols, which normally consist of broadcasting and wait-on-quorums operations. More discussions can be found in Appendix A.

7 Evaluation

This section answers the following questions:

1. How do Electrode and each optimization improve the performance of the Multi-Paxos protocol (§7.1 and §7.2)?



(a) 3 replicas.

(b) 5 replicas.

(c) 7 replicas.

Figure 5: Performance comparison of the Multi-Paxos protocol vs. Electrode-accelerated one with different numbers of replicas.

2. How does Electrode improve the performance of real-world Paxos-based applications (§7.3)?
3. How does Electrode save kernel stack overhead (§7.4)?
4. How does Electrode compare to kernel-bypassing (§7.5)?

Testbed setup: We use eight `xl170` servers from CloudLab [12], each of which has a ten-core Intel E5-2640v4 CPU at 2.4 Ghz, 64GB memory, and a Mellanox ConnectX-4 25 Gbps NIC. Each server runs an unmodified Ubuntu 20.04 OS with kernel v5.8.0. All servers are connected using a two-level topology: five Mellanox 2410 as rack switches (each connecting to forty `xl170` servers) and one Mellanox 2700 as the spine switch. One server is dedicated as the client server that generates Paxos requests, and other servers run the Paxos protocol with 3/5/7-replica configurations. By default, we configure each server to use one core for interrupt processing and another core for Paxos processing, following the performance optimizations in [41]. We disable irqbalance to avoid out-of-order packet deliveries as much as possible (which would hurt Paxos performance), in line with prior Paxos work [40, 44, 61]. Unlike prior Paxos work [32, 40, 61], we *do not* use IP multicast which requires specialized support from the network (§4.1).

Measurement methodology: The client server runs multiple Paxos/application clients, and each client sends Paxos/application requests in either a closed-loop or open-loop manner. In closed-loop experiments, each client sends the next request once it receives the response of the last request; we vary the number of clients and measure the corresponding throughput, and median and 99th-percentile tail latency, in line with prior Paxos work [40, 44, 61]. In open-loop experiments, each client sends requests one by one at a specific time interval, such that the overall request rate reaches a specified value; we use enough clients (i.e., they could saturate the Paxos servers), specify different request rates, and measure the corresponding CPU utilization of each replica node.

Comparisons: We use the Multi-Paxos/VR protocol implementation in the SpecPaxos [61] open-sourced code [35] as the baseline, and optimize it using Electrode. We also run a transactional replicated key-value store similar to the one in SpecPaxos [61] atop the baseline Multi-Paxos protocol and Electrode-accelerated Multi-Paxos protocol. All implementation uses the standard UDP stack and socket layer from the Linux kernel.

7.1 Overall Results

Figure 5a, 5b, and 5c show the performance comparison of the Multi-Paxos protocol and the Electrode-accelerated one when using 3, 5, and 7 replicas, respectively. In each figure, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report throughput and median and 99th-percentile tail latency. All curves eventually hit a "hockey stick" in their median or tail latency growth when the system reaches its maximum throughput.

Throughput: the Electrode-accelerated Multi-Paxos protocol achieves 34.9%, 104.8%, and 128.4% higher maximum throughput than the original Multi-Paxos protocol under 3, 5, and 7 replicas, respectively. The large throughput improvements benefit from the eBPF-based broadcasting and wait-on-quorums which reduce the kernel stack overhead significantly on the leader node. With more replicas, the improvement becomes more significant. This is because, for each Multi-Paxos request, the leader node will send more preparation and commit messages, and handle more ACK messages; thus the eBPF-based broadcasting and wait-on-quorums can save more user-kernel crossings and kernel networking stack traversing.

Latency: the Electrode-accelerated Multi-Paxos protocol achieves 12.5%, 20.0%, and 25.6% lower median latency than the original Multi-Paxos protocol with 2 clients (before

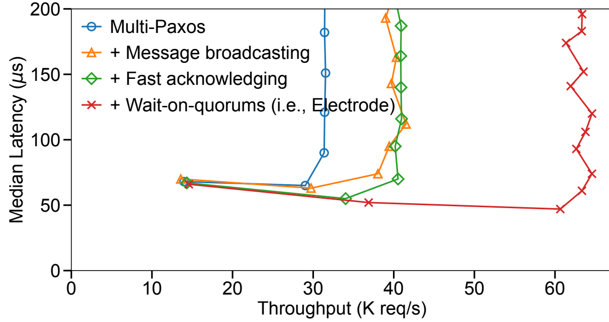
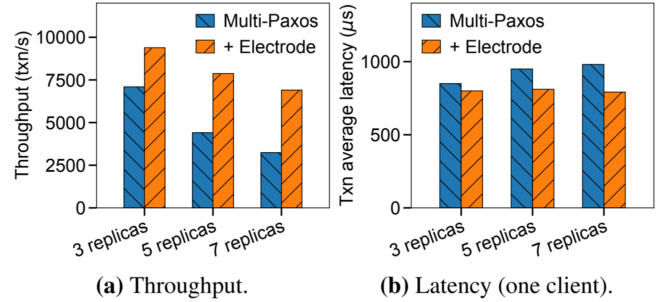


Figure 6: Performance impact of different optimizations for Electrode-accelerated Multi-Paxos protocol (with 5 replicas).



(a) Throughput. **(b)** Latency (one client).

Figure 7: Performance comparison of a transactional key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one.

the "hockey stick") under 3, 5, and 7 replicas, respectively; the corresponding tail latency is 11.8%, 24.7%, and 41.7% lower. The latency reduction mostly comes from the fast acknowledging in the follower nodes, which, for each Multi-Paxos request, saves the time of two user-kernel crossings, two kernel networking stack traversing, and one wake-up of the user-space process. With more replicas, the latency reduction becomes larger. This is because the fast acknowledging bypasses user-space process scheduling and avoids unpredictable scheduling delays [48] by the OS; for the original Multi-Paxos, with more follower nodes, such unpredictable scheduling delays would raise the chance of follower nodes straggling, thus increasing commit latency. Besides, for Multi-Paxos under 3/5 replicas and Electrode under 7 replicas, their latency curves first decline a bit and arrive at the lowest point, then rise and reach the "hockey stick". This is because, under lower throughput, the Linux scheduler would schedule the Paxos process off the CPU more frequently, while under higher throughput, the Paxos process is mostly scheduled on the CPU.

7.2 Performance Gain Breakdown

Figure 6 shows the performance impact of different optimizations for the Electrode-accelerated Multi-Paxos protocol with 5 replicas. Similar to §7.1, we vary the number of clients sending Multi-Paxos requests in a closed-loop manner, and report the throughput and latency. eBPF-based message broadcasting improves the maximum throughput of the Multi-Paxos protocol by 31.7%; fast acknowledging further reduces the median latency by 4.3%-12.7% (before the "hockey stick"); finally, wait-on-quorums improves the maximum throughput by 57.7%. Overall, we find that the two throughput optimizations (i.e., eBPF-based message broadcasting and wait-on-quorums) have almost no impact on the median latency, while the latency optimization (i.e., fast acknowledging) does not nearly impact maximum throughput. This division of labor demonstrates good modularity of each optimization design in Electrode, and each design can be independently used to accelerate more distributed protocols as shown in Table 4.

7.3 Application Performance

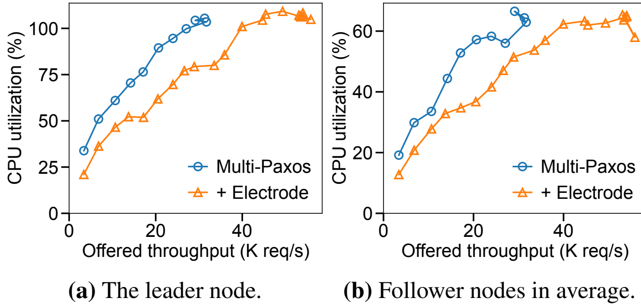
To demonstrate how Electrode can bring benefits to real-world Paxos-based applications, we run a transactional replicated key-value store (similar to the one in SpecPaxos [61]) atop the Multi-Paxos protocol and Electrode-accelerated one. This key-value store supports serializable transactions using two-phase commit and optimistic concurrency control (OCC). Clients use `BEGIN_TXN`, `COMMIT_TXN`, `ABORT_TXN`, `SET`, and `GET` operations to express transactions. We use a synthetic workload derived from the Retwis application [56]—an open-source Twitter clone. This workload consists of four types of transactions with different ratios, and each one issues different numbers of `GET` and `PUT` operations. The workload details can be found in Table 2 of [80]. We vary the number of clients that execute transactions in a closed-loop manner, and measure the maximum throughput these clients can achieve and the average latency under one client.

Figure 7a and 7b shows the maximum throughput and average latency of the key-value store atop the Multi-Paxos protocol vs. Electrode-accelerated one under different numbers of replicas, respectively. Overall, Electrode improves the key-value store throughput by 32.3%-112.9% and latency by 5.9%-19.3%. The improvement becomes larger with more replicas, due to the similar reasons described in §7.1. The latency of the key-value store atop the original Multi-Paxos gradually increases with more replicas, while Electrode-accelerated one's remains relatively stable, because the former is more vulnerable to follower nodes straggling (§7.1).

7.4 CPU Utilization

One design goal of Electrode is to reduce the kernel networking stack overhead (§3) when implementing Paxos protocols. Thus, in this subsection, we study the impact of Electrode on CPU utilizations, which indicates how much kernel stack overhead gets reduced.

Figure 8a and 8b show the CPU utilization of the leader node and follower nodes, respectively, for the Multi-Paxos protocol and Electrode-accelerated one with different offered throughput. The experiments are done in an open-loop manner to control the offered throughput when measuring CPU utilization. The CPU utilization covers both the core handling



(a) The leader node. **(b)** Follower nodes in average.

Figure 8: CPU utilization comparison of the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

interrupts and the core running Paxos. With higher offered throughput, the CPU utilization gradually increases, demonstrating the load-aware CPU scaling provided by the kernel networking stack (§1). We note that for DPDK-based Multi-Paxos protocol implementation, the CPU utilization would be always 100% because DPDK busily polls the network interface. Overall, Electrode reduces the CPU utilization by 22.7%-38.0% on the leader node and 16.0%-35.7% on the follower nodes, benefiting from the reduced user-kernel crossings and kernel stack traversing.

7.5 Comparison with Kernel-Bypassing

Electrode still handles client-facing requests/responses and initiates message broadcasting using the Linux kernel networking stack (§3); thus, it will achieve lower performance than pure kernel-bypassing approaches. This subsection compares the performance of Electrode with a kernel-bypassing baseline, aiming to reveal the performance upper bound of kernel-based approaches and identify the possible improvements for future work.

We choose Caladan [15] and use its high-performance DPDK-based UDP stack to implement our kernel-bypassing baseline. Similar to Caladan, our baseline dedicates one CPU core for packet polling and another core for running the Paxos protocol. We also configure the Caladan runtime to never idle the Paxos core even under low request load.

Table 3 compares the latency and throughput of kernel-based Multi-Paxos and the kernel-bypassing one. To exclude the latency incurred by the client-side kernel stack, we tested all three Paxos implementations with a request generator implemented using Caladan. Electrode achieves 1.4-1.6x lower latency and 2.0x higher throughput than vanilla Linux, but it still has 2.2x higher latency and 2.4x lower throughput compared to pure kernel-bypassing. The performance gap between Electrode and kernel-bypassing exists, because there are still substantial Paxos messages going through the kernel networking stack in Electrode. In particular, our profiling shows that, on the leader node, around 59.5% CPU time is spent on `__libc_sendto()` caused by frequent `dev_queue_xmit()` and `sk_buff` clones. Although eBPF-based broadcasting reduces a significant number of user-kernel crossings and sock-

	Lowest median/99p latency	Maximum throughput
Vanilla Linux	59/69 μs	32 K req/s
Electrode	38/49 μs	65 K req/s
Kernel-bypassing	17/22 μs	154 K req/s

Table 3: Performance comparison of kernel-based Multi-Paxos vs. kernel-bypassing one (with 5 replicas).

/UDP/IP layer traversing, it cannot fundamentally optimize how the Linux kernel manages NICs and packet buffers. Finally, we note that Electrode's goal is to provide generic eBPF-based accelerations for distributed protocol implementations that stick to kernel networking stacks because of compatibility, security, isolation, and elastic CPU scaling.

An additional evaluation regarding how the interrupt coalescing feature of modern NICs impacts Electrode is in Appendix B.

8 Discussion and Future Work

Electrode's offloading decisions: Electrode decides to leave four components of the Multi-Paxos protocol to the user space: 1) failure recovery, 2) handling packet loss and reordering, 3) handling client-facing requests/responses, and 4) executing application-specific operations after reaching the consensus. The first two components involve complex operations on the log, e.g., scanning the log and sending inconsistent entries to other replicas, and inserting missing log entries received from others. These operations require accessing dynamic ranges of log entries, which would fail the eBPF static verification. The last two involve complex serialization/deserialization and application-level operations (see §3). We note that it is possible to offload these four components into eBPF by modifying the kernel eBPF subsystem or verifier—we leave this as future work.

How to improve the eBPF subsystem for offloading? Verifying memory accesses more smartly could make more application operations offloadable. The current eBPF verifier only allows accessing static ranges of memory, which hinders many applications with complex memory accessing behaviors. Another useful construct in eBPF would be dynamic memory allocation, which could ease the maintenance of more advanced data structures in eBPF. To avoid memory leaks, a possible solution could be enforcing Rust-style single-owner memory semantics.

io_uring [1] was recently introduced into the Linux kernel to support efficient batching of asynchronous I/Os via shared memory between the user and kernel space, thus reducing the overhead of frequent user-kernel crossings. Therefore, when implementing Paxos protocols using io_uring, it can help reduce the overhead of message broadcasting, which accounts for 12.5% of CPU time based on Table 1. However, each preparation and ACK message still goes through the

full Linux networking stack and wakes up user-space applications, incurring significant overhead; Electrode can be used together with io_uring to reduce such overhead. A recent work XRP [82] shares a similar view regarding io_uring.

Electrode on shared environments: Electrode requires attaching eBPF programs to the network interface, which then processes every packet accordingly. However, multiple Electrode applications might share the same NIC and attach different eBPF programs that might interfere with each other. We can use the SR-IOV (Single Root IO Virtualization) feature that is widely available in modern NICs [2, 9] to avoid such interference. SR-IOV virtualizes a physical network interface into multiple virtualized ones; the Electrode eBPF program can be attached to only one virtualized interface, without impacting others (e.g., used by non-Paxos applications). Besides SR-IOV, Electrode can also check the port numbers of incoming packets in eBPF, and only execute optimizations if the port numbers belong to target Paxos applications.

Accelerating leader-less consensus protocols using eBPF: Electrode targets at leader-based consensus protocols such as Paxos [37] and its variants [36, 43, 54], because they are the most-used ones by modern distributed applications [6, 8, 22]. Electrode's eBPF-based optimizations could also be applied to leader-less consensus protocols, e.g., EPaxos [52], Mencius [4], SD-Paxos [81], etc. For example, replicas in EPaxos could acknowledge preparation messages earlier in an eBPF program before entering the kernel networking stack, thus reducing latency. We leave the exploration of applying Electrode to leader-less consensus protocols as future work.

9 Related Work

Kernel-bypass and hardware offloading: Overheads of the monolithic kernel networking stack have spurred various attempts to design new kernel-bypassed networking stacks like mTCP [24], eRPC [27], Demikernel [79] and more [15, 29, 33, 48, 57, 67], which attempt to eliminate the kernel from the I/O datapath. But all of these solutions are not backward compatible with solutions that already use the standard kernel networking stack, and they incur more costs in terms of CPU cycles and energy during low I/O loads due to busy-polling. Electrode attempts to leverage eBPF to unclog some of the bottlenecks in the kernel networking stack for distributed protocols without completely having to shift to kernel-bypassed stacks.

Similarly, network offload solutions attempt to offload I/O-intensive operations to specialized hardware, e.g., RDMA [11, 28, 76], FPGA [23], SmartNICs [66], and programmable switches [10, 25]. But they come with limited interfaces for programmability and need custom hardware to be installed.

Co-designing distributed systems with networks: There have been attempts to optimize distributed systems by co-designing them with data center networks for improved performance. SpecPaxos [61] attempts to leverage the natural order of packet delivery in data centers to optimize the ordering of messages needed for state machine replication. NoPaxos [40] uses in-network switches to sequence packets for a similar purpose. Eris [39] further applies in-network sequencing to distributed transactions to avoid coordination overhead. These are orthogonal ways to optimize distributed systems and can be used in conjunction with Electrode.

Distributed protocols in data centers: Data centers have a variety of distributed protocols that are deployed for fault tolerance and data consistency. These include replication protocols like Mencius [4], EPaxos [52], chain replication [74], SDPaxos [81], and transaction protocols like TAPIR [80] and Meerkat [72]. Since many distributed protocols share similar patterns of communication like broadcasting and quorum responses, Electrode can be applied to speed up these distributed protocols as well.

eBPF applications: For a long time, eBPF was only used for packet filtering [49], monitoring [3, 63], and load balancing [14] because of its restricted programming model. Now, it is shown to be able to offload small yet critical operations to improve application performance. CCP [53] mentions that it may be possible to leverage the JIT feature of eBPF to gather datapath's congestion measurements for congestion control. BMC [17] uses eBPF to implement an in-kernel cache to accelerate UDP-based Memcached GET requests and achieves significant throughput improvement. Syrup [26] uses eBPF maps to share incoming request information across OS, networking stacks, and application runtimes to enable user-defined scheduling. SPRIGHT [65] employs fast eBPF-based packet forwarding to accelerate sidecar proxies in serverless computing. XRP [82] offloads storage functions (e.g., B-tree lookups) into the kernel using eBPF to reduce kernel storage stack overhead. SynCord [58] leverages eBPF to inject workload-specific and hardware-aware kernel lock policies specified by application developers. Electrode further demonstrates that eBPF can be used to accelerate distributed protocols under the kernel networking stack.

10 Conclusion

Electrode is a system that accelerates distributed protocols using safe in-kernel eBPF-based packet processing before the networking stack. Electrode retains the benefits of using the standard Linux networking stack (e.g., good maintenance, elastic CPU scaling, security, and isolation), while optimizing the performance-critical operations of distributed protocols (e.g., broadcasting, and wait-on-quorums) in a non-intrusive manner. When applying Electrode to a classic Multi-Paxos protocol, we achieve up to 128.4% higher throughput and 41.7% lower latency. We believe that the designs of eBPF-based optimizations in Electrode can motivate more research on improving networked application performance while maintaining the standard Linux networking stack.

Electrode code is available at `https://github.com/Electrode-NSDI23/Electrode`.

Acknowledgments

We thank our shepherd Adam Belay and the anonymous reviewers for their insightful comments. We thank Cloudlab [12] for providing us with the development and evaluation infrastructure. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by the Google PhD Fellowship.

References

[1] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[2] NVIDIA Corporation affiliates. Single Root IO Virtualization (SR-IOV) for Mellanox NICs. https://docs.nvidia.com/networking/pages/viewpage.action?pageId=43718746.

[3] The Cilium Authors. Cilium: eBPF-Based Networking, Observability, Security. https://cilium.io/.

[4] Catalonia-Spain Barcelona. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of USENIX OSDI*, 2008.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[6] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of USENIX OSDI*, pages 335–350, 2006.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[8] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[9] Intel Corporation. Single Root IO Virtualization (SR-IOV) for Intel NICs. https://www.intel.com/content/www/us/en/support/articles/000005722/ethernet-products.html.

[10] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at Network Speed. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 1–7, 2015.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of ACM SOSP*, pages 54–70, 2015.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[13] Facebook. Facebook's Branch of Apache Thrift, Including a New C++ Server. https://github.com/facebook/fbthrift/blob/main/thrift/doc/cpp/cpp2.md#options.

[14] Facebook. Katran: A High-Performance Layer 4 Load Balancer. https://github.com/facebookincubator/katran.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of ACM SOSP*, pages 29–43, 2003.

[17] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[18] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan MG Wassel, Zhehua Wu, Sunghwan Yoo, et al. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of USENIX NSDI*, pages 1249–1266, 2022.

[19] Google. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[20] The Tcpdump Group. tcpdump. https://www.tcpdump.org/.

[21] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David

Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of ACM CoNEXT*, pages 54–66, 2018.

[22] Michael Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[23] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[24] 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 *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of USENIX NSDI*, pages 35–49, 2018.

[26] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[27] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX OSDI*, pages 185–201, 2016.

[29] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[30] The Linux kernel development community. BPF Ring Buffer. https://www.kernel.org/doc/html/latest/bpf/ringbuf.html.

[31] The Linux kernel development community. struct sk_buff. https://docs.kernel.org/networking/skbuff.html.

[32] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of EuroSys*, pages 1–17, 2020.

[33] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs First-Class Datacenter Citizens. In *Proceedings of USENIX ATC*, pages 863–880, 2019.

[34] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[35] UW Systems Lab. Speculative Paxos Open Source. https://github.com/UWSysLab/specpaxos.

[36] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[37] Leslie Lamport. The Part-Time Parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[38] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and Primary-Backup Replication. In *Proceedings of ACM PODC*, pages 312–313, 2009.

[39] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[40] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI*, pages 467–483, 2016.

[41] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of ACM SoCC*, pages 1–14, 2014.

[42] John C Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of IEEE INFOCOM*, volume 96. Citeseer, 1996.

[43] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. 2012.

[44] Xuhao Luo, Weihai Shen, Shuai Mu, and Tianyin Xu. DepFast: Orchestrating Code of Quorum Systems. In *Proceedings of USENIX ATC*, pages 557–574, 2022.

[45] Linux Programmer's Manual. bpf-helpers(7). https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[46] Linux Programmer's Manual. bpf(2). https://man7.org/linux/man-pages/man2/bpf.2.html.

[47] Linux Programmer's Manual. tc-bpf(8). https://man7.org/linux/man-pages/man8/tc-bpf.8.html.

[48] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[49] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[50] Paul E McKenney. Overview of Linux-Kernel Reference Counting. *N2167*, pages 07–0027, 2007.

[51] Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine Consensus in Wireless Ad hoc Networks. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 537–546. IEEE, 2010.

[52] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of ACM SOSP*, pages 358–372, 2013.

[53] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[54] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of ACM PODC*, pages 8–17, 1988.

[55] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *Proceedings of USENIX ATC, FREENIX Track*, pages 183–191, 1999.

[56] VMware Inc. or its affiliates. Spring Data Redis - Retwis-J. https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[58] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[59] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[60] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. Paxos Made Wireless: Consensus in the Air. In *EWSN*, pages 1–12, 2019.

[61] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of USENIX NSDI*, pages 43–57, 2015.

[62] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of Interrupt Coalescence on Network Measurements. In *International Workshop on Passive and Active Network Measurement*, pages 247–256. Springer, 2004.

[63] The IO Visor Project. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.

[64] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[65] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[66] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of ACM SOSP*, pages 740–755, 2021.

[67] ScyllaDB. SeaStar High Performance Server-Side Application Framework. https://github.com/scylladb/seastar.

[68] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source Routed Multicast for Public Clouds. In *Proceedings of ACM SIGCOMM*, pages 458–471. 2019.

[69] Gráinne Sheerin. gRPC and Deadlines. https://grpc.io/blog/deadlines/.

[70] Alberto Spina, Julie McCann, Michael Breza, and Anandha Gopalan. *Reliable Distributed Consensus for Low-Power Multi-Hop Networks*. PhD thesis, Master's thesis, Imperial College London, 2019.

[71] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland.

The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of VLDB*, page 1150–1160. VLDB Endowment, 2007.

[72] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr Sharma, Arvind Krishnamurthy, Dan RK Ports, and Irene Zhang. Meerkat: Multicore-Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of EuroSys*, pages 1–14, 2020.

[73] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. Optimizing Storage Performance with Calibrated Interrupts. *ACM Transactions on Storage (TOS)*, 18(1):1–32, 2022.

[74] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of USENIX OSDI*, volume 4, 2004.

[75] Ed. W. Eddy. RFC 9293: Transmission Control Protocol (TCP). https://datatracker.ietf.org/doc/html/rfc9293.

[76] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better! In *Proceedings of USENIX OSDI*, pages 233–251, 2018.

[77] IJsbrand Wijnands, E Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. RFC 8279: Multicast Using Bit Index Explicit Replication (BIER). https://www.rfc-editor.org/rfc/rfc8279.

[78] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of ACM SIGCOMM*, pages 126–138, 2020.

[79] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[80] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.

[81] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *Proceedings of ACM SoCC*, pages 68–81, 2018.

[82] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

Types	Protocols	Applying message broadcasting	Applying fast acknowledging	Applying wait-on-quorums
Replication	Primary-backup	The primary broadcasts requests to backups.	Each backup buffers messages in the kernel and quickly responds to the primary.	The primary waits for responses from all backups.
	Chain	None	Each replica (except for the last one) buffers write requests in the kernel and forwards them to the next replica.	None
Concurrency control	Two-phase locking	A transaction coordinator broadcasts `LOCK` and `UNLOCK` requests to all shards.	Each shard maintains a lock table in the kernel and directly handles lock acquiring and releasing.	A transaction coordinator waits for responses from all shards.
	OCC	None	Each shard checks in the kernel if the committing transaction's timestamp conflicts with all other running ones.	None
Atomic commitment	Two-phase commit	A transaction coordinator broadcasts `PREPARE` and `COMMIT` requests to all shards.	Each shard buffers `PREPARE` messages in the kernel and responds to the coordinator, and handles `COMMIT` requests by polling the buffered messages.	A transaction coordinator waits for responses from all shards

Table 4: Applying Electrode to more distributed protocols.

APPENDIX

A Electrode Generalizability

Table 4 summarizes how the classic replication, concurrency control, and atomic commitment protocols can leverage Electrode optimizations. For example, the primary-back replication, two-phase locking, and two-phase commit protocols follow the pattern of sending requests to multiple nodes and waiting for a quorum number of responses; thus they naturally fit well with the eBPF-based message broadcasting and wait-on-quorums. Together with the above protocols, the chain replication [74] and opportunistic concurrency control (OCC) [34] protocols include some critical-yet-simple operations like storing messages in memory, maintaining a lock table, and checking timestamp conflicts; these operations are also suitable for offloading to the eBPF following the fast acknowledging mechanism.

B Impact of Interrupt Coalescing

During benchmarking, we noticed that the interrupt coalescing [62] (IC) feature of modern NICs has a big impact on the measured performance. In IC, after an incoming packet triggers an interrupt, the kernel networking stack waits until a threshold of packets arrives or a timeout gets triggered, aiming to amortize the interrupt cost. In our scenarios, we find it significantly hurts latency and performance predictability in our settings; similar results are also reported in [73]. Thus, in all our experiments, we disable IC by default.

Figure 9 shows the performance impact of IC on the Multi-Paxos protocol and Electrode-accelerated one, by varying the number of open-loop clients. With IC, load-latency curves become unpredictable with two "hockey stick"s. The second "hockey stick" is because the extremely high load triggers coalescing/batching much more packets in one interrupt. Overall, IC does not nearly impact the maximum throughput for the Multi-Paxos protocol and Electrode-accelerated one, but it increases the latency by 57.4%-129.2% and 9.1%-246.8% with 1-3 clients (before the first "hockey stick"). Moreover,

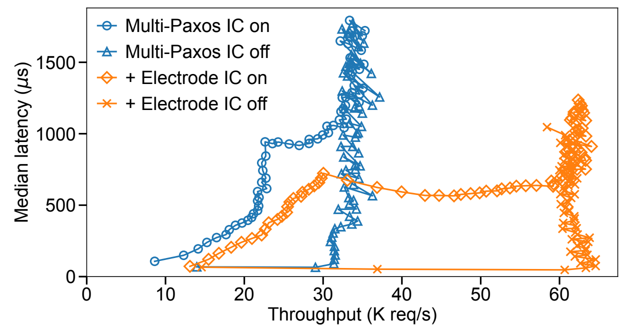


Figure 9: Performance impact of interrupt coalescing (IC) on the Multi-Paxos protocol vs. Electrode-accelerated one (with 5 replicas).

enabling IC decreases the one-client throughput by 38.3% and 10.1% for the original Multi-Paxos and Electrode-accelerated one, respectively.

Electrode performance with IC: Electrode accelerates the maximum throughput of the Multi-Paxos protocol by 81.4% and latency by 32.7% with 1 client (before the first "hockey stick") when IC is on.