HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services

Marios Kogias
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

Abstract

Cloud platform services must simultaneously be scalable, meet low tail latency service-level objectives, and be resilient to a combination of software, hardware, and network failures. Replication plays a fundamental role in meeting both the scalability and the fault-tolerance requirement, but is subject to opposing requirements: (1) scalability is typically achieved by relaxing consistency; (2) fault-tolerance is typically achieved through the consistent replication of state machines. Adding nodes to a system can therefore either increase performance at the expense of consistency, or increase resiliency at the expense of performance.

We propose HovercRaft, a new approach by which adding nodes increases both the resilience and the performance of general-purpose state-machine replication. We achieve this through an extension of the Raft protocol that carefully eliminates CPU and I/O bottlenecks and load balances requests.

Our implementation uses state-of-the-art kernel-bypass techniques, datacenter transport protocols, and in-network programmability to deliver up to 1 million operations/second for clusters of up to 9 nodes, linear speedup over unreplicated configuration for selected workloads, and a 4× speedup for the YCSBE-E benchmark running on Redis over an unreplicated deployment.

ACM Reference Format:
Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3342195.3387545

1 Introduction

Warehouse-scale datacenters operate at an impressive degree of availability and information consistency despite constant component failures at all layers of the network, hardware and software stacks [8], constrained by the well-known theoretical tradeoffs between consistency, availability, and partition-tolerance [12]. This is the result of the careful decomposition of modern applications into components that each exhibit well-defined consistency, scalability, and availability guarantees. Each component is further configured to match specific service-level objectives, often expressed in terms of tail latency [26, 27].

Replication plays a key role in achieving these goals across the entire spectrum of tradeoffs. First, scalability is commonly achieved by managing replicas with relaxed consistency and ordering requirements [12, 27]. This is commonly deployed as a combination of caching layers, data replication, and data sharding [27]. Second, replication is also the foundation for fault-tolerance, whether achieved through fault-tolerant hardware and process pairs [41] or more commonly in datacenters through distributed consensus protocols of the Paxos and Raft families [51, 60–63, 76, 77]. Such protocols ensure fault-tolerance through state machine replication (SMR) [91], in which a distributed system with n nodes can offer both safety and liveness guarantees in the presence of up to f node failures as long as $n \geq 2 \times f + 1$ (under some network assumptions [36]).

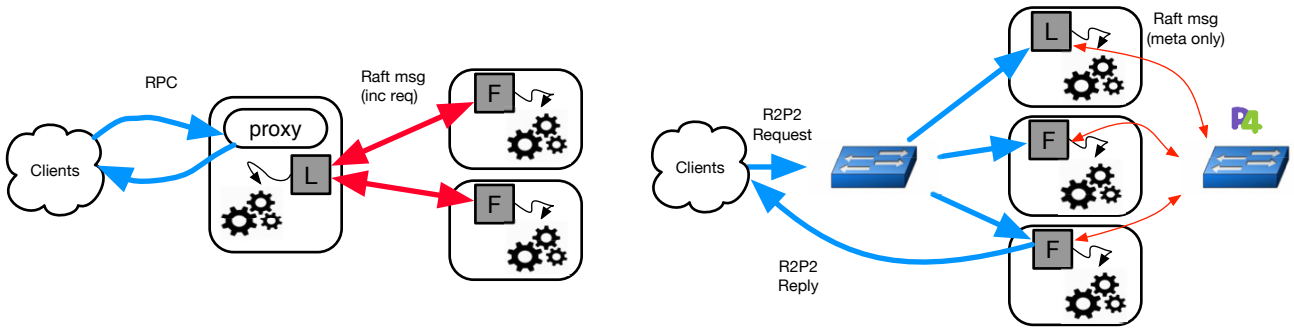
Scalability and fault-tolerance are classically opposed, even though both rely on the same design principle of replication. Adding nodes can improve scalability with relaxed consistency and can lead to very large deployments within and across datacenters (*e.g.,* content delivery networks). On the flip side, adding nodes to a consensus system can improve fault tolerance but harm performance. In practice, most deployments of SMR are limited to small cluster sizes, *e.g.,* three or five replicas [14, 20, 46] as deployments of SMR with more than a handful of nodes reduce performance and are considered too expensive [10, 46].

Figure 1a shows the leader node bottlenecks for a classic SMR deployment using Raft: (1) the leader acts as the RPC server for all clients; (2) the leader must communicate individually with each follower to replicate messages and ensure their ordering. Figure 1a also illustrates that a user-defined application that operates on the state machine must be modified to accept messages delivered by Raft (rather than use a more conventional RPC API transport).

In this work we focus on stateful datacenter applications that require fault-tolerance, low-latency, and scalability. We ask two main questions: (1) *Can we build fault-tolerant services in an application-agnostic, reusable manner, i.e., take an existing application (with deterministic behavior) and have*

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04.
https://doi.org/10.1145/3342195.3387545



(a) Scalability bottlenecks of Raft.

(b) HovercRaft, with optional in-network acceleration using a P4 ASIC.

Figure 1. Eliminating bottlenecks of SRM. Illustration on a 3-node cluster.

it transparently utilize a SMR protocol? (2) *Can we take advantage of the replication present for fault-tolerance to also improve the performance of the application?*

Figure 1b illustrates the key contributions of our system, HovercRaft, and its optional extension (HovercRaft++). We answer the first question primarily by integrating the Raft protocol [77] directly within R2P2 [58], a transport protocol specifically designed for in-network policy enforcement over remote procedure calls (RPC) inside a datacenter.

We answer the second question by first extending Raft to separate request replication from ordering and using IP multicast and in-network accelerators (*e.g.,* a P4 ASIC) to convert leader-to-multipoint interactions into point-to-point interactions. These enhancements reduce the performance degradation associated with larger clusters. We complete the answer to the second question by using R2P2's fundamental mechanism for in-network operations, which allows the destination IP address of an RPC request (as set by the client) to differ from the source IP of the reply. This allows the load balancing of client replies, as well as of the execution of read-only, totally-ordered requests. These changes reduce the I/O and CPU bottlenecks and allow HovercRaft to deliver even superior performance to an unreplicated (and therefore not fault-tolerant) deployment of the same application.

We make the following contributions:

- We integrate Raft [77], a widely used consensus algorithm, with R2P2 [58], a transport protocol specifically designed for datacenter RPCs, to offer fault-tolerance transparently to applications.
- We propose HovercRaft, a set of Raft protocol extensions that leverage the built-in features of R2P2 to systematically eliminate I/O and CPU bottlenecks associated with SMR, without changing the core of the algorithm, thus its liveness and safety guarantees.
- We further take advantage of in-network accelerators now commonly found in datacenter switches to statelessly offload low-level message processing, thus eliminating scalability bottlenecks due to cluster size.

Our implementation of Raft relies on kernel-bypass to deliver up to 1M ordered operations per second in a series of microbenchmarks on a 3-node cluster, which corresponds to a 4× improvement over the state-of-the-art [31, 66, 85]. Our implementation of HovercRaft++ delivers 1M ordered operations for clusters of up to 9 nodes. The careful elimination of CPU and IO bottlenecks allows almost linear speedup over the unreplicated configuration for selected workloads. Our evaluation of Redis running YCSB-E shows that HovercRaft can deliver up to 142k YCSB-E operations per second on a 7-node cluster in $\leq$ 500µs at the 99^{th} percentile, a 4× performance increase over the unreplicated case.

The HovercRaft codebase is opensource and can be found at `https://github.com/epfl-dcsl/hovercraft`. The rest of the paper is structured as follows: after the necessary motivation and background (§2), we present the design of HovercRaft (§3), and its optional extension using in-network accelerators (§4). We then identify the common properties of Raft and HovercRaft, as well as their subtle differences (§5). Finally, we describe our implementation of HovercRaft (§6) and evaluate it using representative applications and workloads (§7). We describe the related work (§9) and conclude (§10).

2 Motivation

2.1 Replication for Fault-Tolerance

SMR systems such as Chubby [14], Zookeeper [46], and `etcd` [33] manage the hard, centralized state at the core of large-scale distributed services, *e.g.,* Kubernetes [18] and RamCloud [81] use Raft/etcd for state management; Azure Storage [16], Borg and Omega [13], Ceph [96], and GFS [38] all have a built-in Paxos implementation to manage storage metadata and cluster state. Because of their complexity, SMR systems traditionally implement simple abstractions, such

as a key-value store or a hierarchical namespace, exposed to clients via a protocol proxy (*e.g.,* `http`).

Complex and stateful applications, such as databases, typically depend on application-specific solutions. For example, they replicate state via cluster configurations (*e.g.,* active-passive database pairs) that can suffer from split-brain scenarios or require manual intervention in case of failures [4]. These stateful applications come with critical performance advantages as they reduce communication roundtrips and enable complex, atomic, transactional operations on state.

The separation of concerns between stateless applications and stateful services exposing simple abstractions comes with extra cost and complexity, due to the marshalling of RPCs, multiple roundtrips due to abstraction mismatches, and added network latency [1]. Approaches such as Splinter [59] and Redis user-defined modules [89] help bridge the gap through application-specific abstractions and richer semantics, making a harder case for fault-tolerance, though.

As we will show, we advocate for transport protocols specifically designed for RPCs with integrated SMR functionality to support applications with rich abstractions.

2.1.1 State Machine Replication.

At the core of all fault-tolerant systems is a variant of a consensus algorithm that is used to implement distributed state-machine replication [91]. State machine replication guarantees linearizability, namely all replicas receive and apply the updates to the state machine in the same order. Consequently, all replicas in the same group behave as a single machine. This field has been the subject of extensive academic and industrial research both from a theoretical [36, 60, 91] and systems [14, 46, 62, 77] point of view.

Most consensus algorithms can be split into two phases: the *leader-election* phase, where the participating nodes vote to elect a leader node; and the *normal operation*. Once a leader is elected, the leader is in charge of selecting one of the client requests to be executed and notifies the rest of the nodes about its choice. Once the majority of nodes are aware of this choice, the leader can commit and announce the committed decision, and the request can be executed.

Although the terminology (*e.g.,* acceptors and learners in Paxos vs. followers in Raft) and the specific semantics change across different implementations, the key point is that consensus algorithms generally operate in a 2-roundtrip communication scheme. In the first round-trip the leader announces the intention to execute an operation, and in the second it announces that the operation is committed, given the necessary majority.

For the rest of the paper we will focus on Raft [77] and use Raft terminology. Raft is a consensus algorithm that depends on a strong leader and exposes the abstraction of a replicated log. The leader receives client requests, puts them in its log, thus guaranteeing a total order, and replicates those to the follower through an `append_entries` request. The followers append them to their logs and notify the leader. Then, the leader can execute the operation and reply to the client. The above interaction is summarized in Figure 2. The leader notifies the followers for its committed log index in the next communication round. The choice of Raft is crucial for HovercRaft's design (§3).

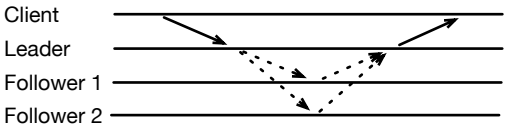


Figure 2. Basic communication pattern to execute a client request in a fault-tolerant group of 3 servers. Solid arrows refer to messages from and to the client based on the service-specific API, while dotted arrows refer to SMR messages.

2.1.2 SMR Bottlenecks.

We now focus on the normal case of operation and we identify bottlenecks that might arise based on the interactions between the clients and the fault-tolerant group of servers, as well as the consensus communication pattern inside the fault-tolerant group specifically in the case of Raft.

Leader IO bottleneck for request replication: The leader is in charge of replicating requests to the followers. Once those requests start increasing in size or the number of followers increases, the transmission bandwidth of the leader's NIC will become the bottleneck, thus limiting throughput.

Leader IO bottleneck for client replies: The leader must also reply to all clients. Depending on the reply sizes, the leader transmission bandwidth can again become a bottleneck, especially in combination with the replication traffic.

Leader CPU bottleneck for running requests: Since the leader is expected to reply to the client, it must run all client requests, even read-only ones. However, this can lead to a CPU bottleneck in the leader if the read-only operations are expensive. This, for example, can be the case in systems supporting mixed OLTP and OLAP workloads [88].

Leader packet processing rate: The leader must send requests and receive replies from the majority of the followers in order to make forward progress. Increasing quorum sizes increases the packet processing requirements at the leader. This can both limit the throughput of committed client requests if the IOPS becomes a bottleneck, and increase latency before hitting the IOPS bottleneck.

2.2 Replication for Scalability

Data replication, either in the form of caching or load balancing between replicated servers, is used to improve the latency and throughput of data accesses. Replication, though,

improves scalability by trading off either availability or consistency, as the CAP theorem suggests [12, 27]. In such replicated systems, concurrent accesses and component failures lead to anomalies that must be handled either at the application layer or by the end-user herself [27].

Replication is therefore the fundamental mechanism used to offer either scalability or fault-tolerance, but not both at the same time. Replication for scalability requires compromises in the consistency model and can at best offer high-availability [3].

2.3 Low-latency intra-datacenter interactions

The core of datacenter communications relies on short µs-scale RPCs [2, 37, 58, 71]. In current datacenters, using standard networking hardware technologies, any two computers can communicate in $\leq 10\mu s$ at the hardware level between any two NIC. Such time budget includes the latencies associated with PCI, DMA, copper transmission, optical transmission, and multiple hops through cut-through switches.

Recent advances in system software eliminate most software overheads by bypassing the traditional networking stacks. These technologies include kernel-bypass development kits [29], user-level networking stacks [49, 68, 78], stacks that bypass the POSIX layer [44], protected dataplanes [9, 84, 87], and microkernel-based approaches [56, 69]. Such stacks support either standard networking protocols such as TCP or UDP, protocols specifically designed to accelerate datacenter RPC interactions [42, 58], or leverage RDMA to directly access and manipulate data [30, 69, 70, 81].

Such proximity fundamentally changes assumptions about the cost of sending messages from applications (clients) to the nodes of a replicated system, as well as between the nodes themselves. Approaches such as [58] show that increasing the number of round-trips can lead to better tail-latency due to better scheduling.

At the same time, the introduction of new non-volatile memory technologies [79] reduces the access latencies, increases throughput [48], and eliminates the traditional bottleneck associated with persistent storage, especially when combined with low-latency networking [28, 97]. In particular, the combination opens up new opportunities for SMR applications previously presumed to be storage-bound.

3 Design

Our goal is to provide a systematic and application-agnostic way of building fault-tolerant, µs-scale, datacenter services, with similar or better performance to the unreplicated ones. We achieve this by integrating the consensus directly within the RPC layer (§3.1) and through a set of extensions to the Raft consensus protocol that do not modify the core algorithm, but only go after its CPU and IO bottlenecks. We call

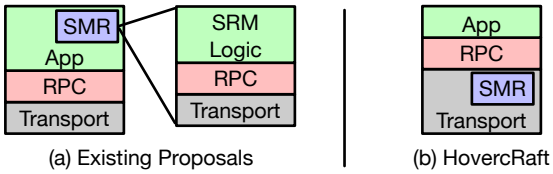


Figure 3. HovercRaft proposal in terms of layering compared to existing approaches.

the Raft version with the performance extensions HovercRaft. Figure 1b summarizes our design, and in the rest of the section we analyze each design choice.

3.1 SMR-aware RPC layer

Clients interact with stateful servers through RPCs to change or query the internal server state, making RPCs the natural place to implement a systematic, application-agnostic mechanism for fault-tolerance. In a standard design of a fault-tolerant service, clients interact with the nodes of a cluster through a standard RPC library that is oblivious to SMR. For example, `etcd` uses gRPC [42] for this interaction. This protocol layering requires the SMR leader, which receives the initial client RPC, to decode it, re-encode it within the SMR protocol for insertion into the consensus log, and finally forward it to the followers. The selection of the endpoint for the client RPC can be done via a stateless load balancer which hides the internal IP addresses of the cluster but does not improve performance since all client requests have to go through the leader, *e.g.*, the `etcd gateway` [34].

We advocate, instead, to incorporate consensus directly within an RPC library or a transport protocol, which has RPC semantics (*e.g.*, gRPC [42] or R2P2 [58]) and to provide fault-tolerance at the RPC granularity. Specifically, the SMR layer becomes part of the RPC layer which forwards RPC requests to the application layer only after those requests have been totally ordered and committed by the leader. Doing so can transform any existing RPC service with deterministic behavior into a fault-tolerant one with no code modifications.

We chose to work with R2P2 [58] and incorporate Raft [77] within the RPC processing logic. We chose R2P2 as it is a transport protocol specifically designed for datacenter RPCs that targets in-network policy enforcement and decouples the initial request target from the replier. This design choice in R2P2 is crucial for RPC-level load balancing implemented at its core.

Figure 3 describes the proposed design in terms of layering. Unlike previous approaches that incorporate SRM libraries at the application layer, HovercRaft incorporates SMR within the same transport protocol that also exposes request-response pair semantics, making the solution application agnostic.

Our choice of Raft, instead of a Paxos variant, is driven by Raft's strong leader, and the in-order commit mechanism.

In the original Paxos algorithm [60] there can be a different leader in each consensus round, thus limiting the potential for optimization due to centralized choice. Unlike Paxos, Raft is a consensus algorithm that depends on a strong leader in charge of ordering and replicating client requests across all followers, with a global view of all the nodes participating in the fault-tolerant group. Our goal is to take advantage of the global cluster view at the Raft leader in conjunction with R2P2's load balancing capabilities to go after the SMR bottlenecks. Although the above requirement is also satisfied by Multi-Paxos [61], Raft's in-order commit logic significantly simplifies the design of in-network acceleration for HovercRaft. We specifically target scalability bottlenecks, and we note that our design does not improve upon Raft's two network roundtrip approach, unlike 1-roundtrip proposals [66].

3.2 Separating RPC Replication from Ordering

The first Raft bottleneck of §2.1.2 is the leader IO transmission bottleneck due to request replication. With n nodes in the cluster, a mean RPC request size of S_{req}, and a link capacity of C, Raft cannot serve more than $C/((n-1)*S_{req})$ requests per second.

HovercRaft achieves fixed-cost SMR independent of the RPC request size. We solve the bottleneck by separating replication from ordering and leverage IP multicast to replicate the requests to all nodes. Instead of targeting a specific server, clients inside the datacenter send requests to a multicast group that includes the leader and the followers, or a middlebox that is in charge of assigning the correct multicast IP. All nodes in the group receive client requests without the leader having to send them individually to each of the follower nodes.

Obviously, IP-multicast does not guarantee ordering or delivery. As with Raft, the leader decides the order of client requests. R2P2 provides a way to uniquely identify an RPC based on a 3-tuple *(req_id, src_port, src_ip)* and HovercRaft relies on this metadata built into the protocol. Upon reception of an RPC request, the leader immediately adds it to its log, while followers insert the RPC into a set of unordered requests. The leader then communicates fixed-size request metadata to the followers. Followers retrieve the RPC requests from the unordered set based on the request metadata and add them in their log. Therefore, in the common case, when no packet loss occurs, the leader is only in charge of ordering requests and not data replication.

3.3 Load balancing replies

The second bottleneck mentioned in §2.1.2 refers to the cost of replying to clients from the leader. We observe that replying to the client can be load balanced between the leader and the followers as long as the followers keep up with the request execution. By doing so, the I/O bottleneck due to client replies could expand from C/S_{reply} RPS to almost $n * C/Sreply$ RPS, where C is again the link capacity and

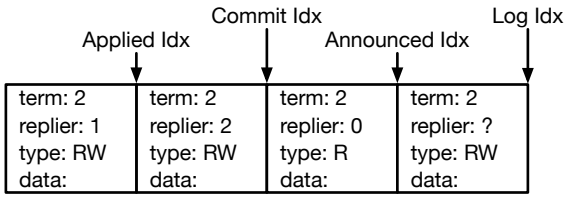


Figure 4. A view of the leader log. Each log entry includes the term, the designated replier, a pointer to the client request, and whether it is read-only. The replier is set only for the entries up until the `announced_idx`.

S_{reply} is the reply size. For example, a server with a 10G NIC can serve up to approximately 400k requests per second, when the replies are 2 MTUs (we assume an MTU of 1500 bytes). Load balancing those replies among 3 replicas can improve application throughput by 3×. It also reduces the leader CPU load due to network protocol processing.

The leader can decide which node replies to the client for which request, given its global view of the cluster. According to the semantics of the vanilla Raft algorithm, a leader sends an `append_entries` message to the followers that includes the client requests and the leader commit index, so that followers can update their own commit index. There is no explicit commit message per log entry. As a result, the leader has to designate the replier when announcing the request order to the followers, and not after committing them.

HovercRaft extends the information stored in the Raft log with a `replier` field. The leader sets the `replier` field immutably for each entry before sending the particular entry to any follower for the first time. When a log entry is later committed, each Raft node can run the RPC, but only the one with the matching replier identifier replies to the client. For this, we use a built-in feature of R2P2 that allows the source IP address of the RPC reply to differ from the destination IP address of the request.

The load balancing of replies creates a window of uncertainty between the `append_entries` request that communicates the designated replier, and the commit point, during which the designated replier can fail, and as a result the client will not receive a reply. Note that introducing this window does not affect the correctness of the SMR algorithm, as it is consistent with Raft's semantics that do not guarantee exactly-once execution, and can lead to missed replies. Note also that clients may receive their replies in a different order than in the log; this is not, however, different than Raft.

3.4 Bounded Queues

We rely on another design idea introduced in R2P2, *bounded queues*, to minimize the potential visible impact of a particular node failure to clients to at most B lost replies. R2P2's *Join-Bounded-Shortest-Queue* was designed as a load balancing policy to mitigate tail latency across stateless servers [58].

Join-Bounded-Shortest-Queue splits queueing between a large centralized queue and distributed bounded queues, one per server. The rational behind this policy is to delay delegating requests to a specific server queue in anticipation of better scheduling, approaching the performance of the optimal single queue. HovercRaft delays the reply node assignment to bound the number of lost replies in the case of node failure.

Specifically, HovercRaft caps the quantity of announced entries from the leader's log relatively to the applied index, thus bounding the amount of assigned but not applied operations. Figure 4 describes the different indices on the log: (1) the leader inserts entries at `log_idx`, the head of the log without determining yet which node will send the reply; (2) the leader selects the node in charge of replying to the client and updates the `announced_idx` accordingly; (3) the `commit_idx` represents the point upon which consensus has been reached; (4) the `applied_idx` represents the point upon which operations have been applied to the state machine. Each follower also has its own set of `applied_idx`, `commit_idx`, and `log_idx` indices on its local log. `Announced_idx` is relevant only for the leader. Followers communicate their `applied_idx` to the leader as part of the `append_entries` reply.

For every node there is a bounded queue of reply assignments to that node between its `applied_idx` and the leader's `announced_idx`. The leader respects the invariant of the bounded queue at node selection time, *i.e.,* when moving `announce_idx`: nodes with too many operations left to apply are not eligible for receiving additional work. This obviously includes the case when the node has failed and its `applied_idx` does not progress. Thus, choosing nodes based on the bounded queues minimizes the risk of selecting a failed node and eventually losing client replies.

We note that respecting the bounded queue invariant never affects liveness. When no node is eligible for designated replier, the leader simply waits either for the application in the leader node to make progress and selects itself as a replier, or an `append_entries` reply from a follower that will make this follower eligible to reply.

3.5 Load balancing Read-only Operations

The third bottleneck of §2.1.2 is the leader CPU bottleneck. We observe that many RPCs only query the state machine but do not modify it. Such read-only requests still need to be placed in the Raft log and ordered to guarantee strong consistency, but do not need to be executed by all nodes. The load balancing design of §3.3 therefore naturally extends to the CPU for read-only operations, and can increase the global CPU capacity of the system. Clients tag their requests as read-only as part of the metadata of the R2P2 RPC. This information is also kept in the Raft log and propagated to the followers (see Figure 4). It follows that all requests remain totally ordered by Raft, but only the designated replier node executes a read-only query.

Read leases [40], which are an alternative solution for read-only requests, were initially proposed for Paxos [17] and also used in Chubby [14] and Spanner [20]. In this approach, read-only operations run on the leader without running consensus for them. However, this increases the CPU load and traffic on the leader. Megastore [5] grants leases to every replica for different read operations, but requires communicating with every replica to perform a write request. Quorum leases [73] also load balance read-only requests among different nodes, but assume an application-specific way of detecting read-write dependencies and do not match our application agnostic requirements. Finally, there is also the choice of not ordering read operations, acknowledging the risk of returning stale replies [32]

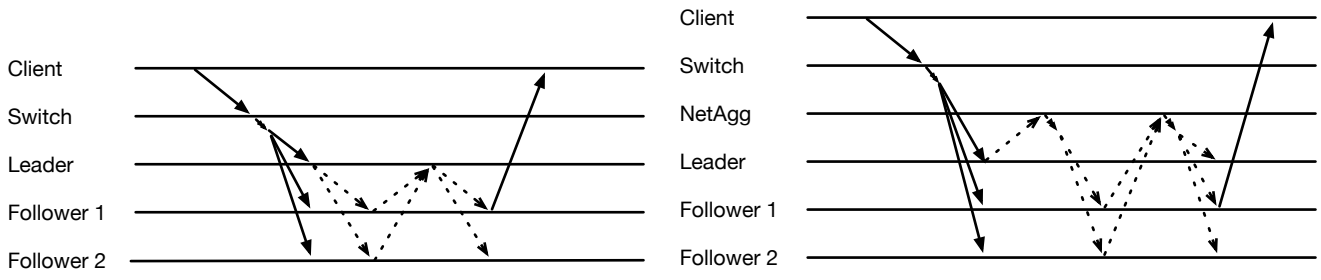
3.6 Join-Bounded-Shortest-Queue

Bounded queues are necessary to limit the lost client replies up to the queue bound, but can also help improve the end-to-end request latency, especially in cases where read-only requests have high service time variability. Once the leader decides to announce more log entries, identifies the eligible followers, and has to select which follower will reply to the client. One option is random choice among eligible nodes. Another one is to leverage R2P2's *Join-Bounded* ***Shortest*** *Queue (JBSQ)* policy [58].

In HovercRaft, the leader maintains the queue depth of requests to be executed on each node. This counter is increased every time the leader assigns a request and decremented when followers reply to the leader. The JBSQ policy load balances the requests based on the known queue depths, as this is known to improve tail-latency compared to a random selection [58], by choosing the shortest among the bounded queues. The queue of a follower that is assigned to run a long read-only request will fill up while the node is busy serving the request and that will prevent the leader from assigning more work to that node. Preferring the other less loaded nodes is expected to improve the tail-latency in cases of high service-time variability, compared to the naive random assignment policy.

3.7 Communication in HovercRaft

Figure 5a summarizes the logic and the communication pattern described in the above design. A client sends requests to a pre-defined multicast group. Requests get replicated to all nodes based on multicast functionality existing in commodity switches. The leader orders requests and sends `append_entries` with request metadata to followers, identifying the request type (read-only or read-write) and delegating client replies; in this particular case follower 1 will reply. The followers reply to the leader and the leader commits the request. At the next `append_entries` request the followers are notified about the leader's current commit index, execute the



(a) HovercRaft **(b)** HovercRaft++

Figure 5. The communication patterns in our design resulting from separating replication from ordering and using in-network fan-in/fan-out. Solid arrows refer to application requests and replies. Dotted arrows refer to SMR messages.

client request, and follower 1 replies to the client. The above communication pattern increases latency in the unloaded case (2.5 RTTs), but can lead to significant throughput benefits.

4 HovercRaft++

The last bottleneck of § 2.1.2 is the packet processing overhead at the leader that becomes worse as the number of followers increases. The leader has to send `append_entries` requests to each follower independently and receive replies for each request. Communication delays inside the datacenter, though, are short and usually predictable, so it is likely that all followers make progress at the same pace. Based on that observation, we tackle the packet processing bottleneck using in-network programmability. Specifically, we design and implement an in-network aggregator, based on a P4-enabled programmable switch, that will handle the fan-out and fan-in of the `append_entries` requests and replies. This in-network accelerator should be viewed as part of the leader. A P4 switch runs the aggregation logic at line rate. So, we offload some of the leader's packet processing duties, thus reducing the leader's CPU pressure, on a hardware appliance specifically designed for packet IO. Our goal is to achieve fixed-cost SMR in the non-failure case independently of the number of followers for small cluster sizes.

Figure 5b describes our proposed design. Similarly, to HovercRaft's communication pattern (Figure 5a), a client sends a request to the multicast group and this request gets replicated to all nodes. However, in the case of in-network acceleration (HovercRaft++), the leader, instead of communicating with each individual follower, sends only one `append_entries` request to the network aggregator. This request includes the metadata to implement the aforementioned load balancing logic. The network aggregator then forwards this request to the equivalent multicast group excluding the leader. The followers reply back to the aggregator and the aggregator keeps track of the per-follower replies, without forwarding them to the leader. Once the majority of the followers have successfully replied, the aggregator multicasts an `AGG_COMMIT` to all the nodes in the group, announcing the new commit index. Based on that message, the delegate follower (follower 1) replies back to the client.

MSGs\System	Raft	HovercRaft	HovercRaft++
Rx msgs	1+(N-1)	1+(N-1)	1+1
Tx msgs	(N-1)+1	(N-1) + 1/N	1+1/N

Table 1. Comparison of Rx and Tx message overheads for the leader in Raft, HovercRaft, and HovercRaft++ for the non-failure case. N is the number of nodes.

Table 1 summarizes the communication complexity at the leader in Tx and Rx messages for the different approaches for a cluster with N nodes (N-1 followers and a leader). In the case of Raft, the leader receives the client request, sends N-1 `append_entries` requests to the followers, receives N-1 `append_entries` replies, and sends the reply to the client. In the case of HovercRaft, the leader receives the client request, sends N-1 `append_entries` requests (smaller than in the previous case) to the followers, receives N-1 `append_entries` replies, and approximately sends only 1/N replies to the client because of replies load balancing. Finally, in the case of HovercRaft++, the leader receives the client request, sends only one `append_entries` request to the aggregator that multicasts it to the followers, the aggregator collects the quorum and sends one `append_entries` reply to the leader. Similarly with the previous case, the leader approximately sends only 1/N replies to the client.

5 HovercRaft vs Raft

HovercRaft does not modify the core of the Raft algorithm but instead goes after its bottlenecks and implements optimizations to bypass them. Those optimizations are only in effect in the non-failure mode of operation. HovercRaft falls back to vanilla Raft whenever a failure, *e.g.,* failed leader, is detected. As a result, HovercRaft provides exactly the same

linearizability guarantees as Raft. It assumes the same failure mode, and guarantees safety and liveness with $2f + 1$ nodes, where up to f nodes can fail.

Raft's correctness depends heavily on the strong leader and its election process. HovercRaft's modifications to the Raft logic only affect the normal operation after a leader is elected. As a result, the correctness of the leader election is not challenged.

The rest of this session discusses the modifications introduced by HovercRaft, how they affect the consensus logic, and how HovercRaft handles failures.

Separating replication and ordering: In HovercRaft, all nodes receive client requests through the multicast group and the leader is in charge of ordering. Client requests are placed in the Raft log as they are received only in the leader. Followers keep those requests in a list of un-ordered requests waiting for an `append_entries` request.

Followers index unordered requests based on R2P2 unique 3-tuple *(req_id, src_ip, src_port)*. Clients are responsible to ensure the uniqueness of the metadata identifiers. This is not a problem in practice given the large R2P2 metadata namespace. The leader can also include a hash of the request body to avoid cases of metadata collision. Therefore, there is a unique mapping between metadata in the *append_entries* message and the requests in the followers unordered list.

HovercRaft does not assume reliable multicast [45, 83]. Consequently, there might be cases in which client requests do not reach all the nodes. Followers detect such cases when processing an `append_entries` message and do not find the equivalent client request in their unordered set. We introduced a new `recovery_request` message type. Followers use this request to ask for a missing client request from the leader or any other follower that might have potentially received it. Once a follower retrieves a missing request, it adds it to the unordered set, waiting for the next `append_entries` request from the leader to order it properly.

The inverse case, in which the followers received a client request but the leader did not, does not require changes to the algorithm. The followers periodically garbage collect client requests in their unordered set that linger, based on a specific timeout. Early garbage collection does not affect the correctness of the algorithm and will unnecessarily trigger the recover mechanism described above.

Bounded Queues: Bounding the amount of committed but unapplied requests does not affect the number of lost client requests in case of a leader failure. Followers have also received the client requests already placed in the failed leader log, but not yet announced. When a leader fails, the new leader will remove the received client requests from its unordered set, add them to its log in some order and start sending `append_entries` announcing their order.

Load Balancing Client Replies: Raft does not guarantee exactly-once RPC semantics but instead only at-most-once RPC semantics [92]. It only guarantees linearizability of operations, leaving the client outside the algorithmic logic. Consequently, the client reply can be lost or the leader can crash after committing a log entry and before replying to the client. Guaranteeing exactly-once RPC semantics is outside the scope of Raft and projects like RIFL [64] solve the problem of implementing exactly-once semantics on top of infrastructures providing at-most-once RPC semantics.

HovercRaft's ability to load balance replies introduces this window of uncertainty between the point of replier choice by the leader and the actual client reply. This is consistent with Raft's at-most-once RPC semantics, does not affect correctness, and should be considered equivalent to the cases of missing replies in vanilla Raft.

Load Balancing Read-Only Operations: Read-only operations do not modify the state machine but still need to be ordered to guarantee strong consistency. It is safe to execute them only in the designated replier, only after they have been committed. The application (client-side or server-side) is responsible to correctly identify which operations are read-only, as to avoid a catastrophic inconsistency.

In-network Aggregation: The aggregator should be viewed as part of the leader that undertakes leader tasks in the non-failure case. In HovercRaft++, followers reply to the aggregator only when `append_entries` requests succeed. When an `append_entries` fails, *e.g.,* due to wrong previous entry, followers talk directly to the leader, bypassing the aggregator. When the leader receives a failure reply for an `append_entries` request, it uses point-to-point communication with this follower until it recovers. In the meantime, this follower receives the multicast `append_entries` requests from the aggregator and keeps them in order to identify when to stop the recovery process with the leader, and continue with using the messages from the aggregator instead.

If the in-network aggregator fails, the followers stop receiving requests from the leader. This will trigger a new election process, in which the aggregator does not participate. Once a new leader is elected based on the vanilla Raft election process, the new leader has to identify whether to use the in-network aggregator or not. The leader switches to HovercRaft++ once it has confirmed the liveness of the in-network aggregator. The state in the aggregator is flushed after every new leader election.

Model-checking the correctness of HovercRaft++ using TLA+ [93] is left for future work.

6 Implementation

We implemented HovercRaft and HovercRaft++ based on the open-source version of R2P2 [39], and a production-grade, open-source implementation of Raft [15], which is

thoroughly tested for correctness, and used in Intel's distributed object store [25]. We built the network aggregator in $P4_{14}$ and ran it in a Tofino ASIC [7].

To guarantee timely replies, we dedicate a thread to network processing and a thread to run the application logic. The networking thread is in charge of receiving client requests and running the R2P2 and consensus logic. The application thread is in charge of running the application and replying to the client if necessary. In our implementation on top of DPDK [29], we configure 1 RX queue for the networking thread and 2 TX queues, one for each thread. The networking thread polls the RX queue, while the application thread polls for changes in the `commit_idx` and applies the newly committed entries.

6.1 R2P2 protocol extensions

We extended R2P2 to integrate consensus in its RPC processing logic. R2P2's RPC-tailored semantics and its RPC-aware design choice are a perfect fit to achieve our goal.

The R2P2 header includes two relevant fields for our implementation. The first one is the `POLICY` field, initially used to define load balancing policies. We extended the semantics of this field with two new policies. Clients use those fields to tag requests that must be totally ordered for strong consistency. Specifically, requests that read and modify the state machine should be marked with `REPLICATED_REQ`, and requests that only read with `REPLICATED_REQ_R`. Marking requests that require strong consistency allow servers in the fault-tolerance group to serve also other requests that are not replicated, with the probability of stale data, similarly to `etcd` [33]. Those non-replicated requests can also be load balanced based on the techniques described in [58].

The second relevant field is the message type. Given that Raft itself depends on RPCs, Raft RPCs are also on top of R2P2. We added two more message types, one for Raft requests and one for Raft responses, as to separate them from the client ones since they have to be handled by the consensus logic in R2P2. These fields are also used by the network aggregator to specially handle Raft requests and replies.

6.2 Raft extensions

We added minimal modifications to the Raft implementation initially for high throughput and low-latency and then to support HovercRaft. Specifically, we switched from the periodic application of the log, to eager application the moment the entries are committed. Then, we extended the log entry with two new fields to include the replier identifier and the entry's type (read-only/read-write). Finally, without modifying the Raft code, we extended the `append_entries` reply to include the applied index, necessary for the bounded queues (§3.4) and load balancing.

6.3 Multicast Flow Control and Recovery

Raft and HovercRaft differ noticeably in one particular area. In vanilla Raft, the leader is the only one receiving client requests and is in charge of both ordering and replicating them; it is the only bottleneck in the system. Therefore, dropping client requests at the leader is a form of implicit flow control. HovercRaft, however, leverages multicasting to replicate client requests, which implies that, under high load, different requests will be dropped for the leader and the followers.

As a consequence, HovercRaft has to implement flow control to guarantee forward progress under high load or bursts of load that lead to dropped multicast client messages. One way of dealing with the problem is to let the leader and the followers drop requests independently under high load, and rely on the recovery mechanism to create back-pressure. However, this would lead to poor performance.

Instead, we leverage the R2P2's `FEEDBACK` mechanism, that is designed to be repurposed according to the application needs, to limit the number of outstanding client requests in the system. For example, the R2P2 request router used `FEEDBACK` messages to implement the `JBSQ` scheduling policy. HovercRaft and HovercRaft++ use `FEEDBACK` messages to implement a coarse-grained flow control mechanism for multicast traffic.

Specifically, instead of letting clients send requests to a multicast IP, we use a middlebox, *e.g.,* a programmable switch, that counts the number of requests in the system and switches the destination IP to the multicast IP of the fault-tolerance group. Every time a node sends back a reply to the client, it also sends a `FEEDBACK` message to the flow-control middlebox, to decrement the counter of requests. When the number of requests in the system reaches a certain threshold, instead of multicasting, the middlebox sends a NACK back to the client for every new client request arriving, thus preventing throughput collapse in the system.

6.4 Aggregator implementation

Our in-network aggregator is implemented as part of a Tofino programmable switch, but it is an IP connected device that can be placed anywhere inside the data center. The aggregator maintains only soft state that is flushed on every new leader election.

The aggregator needs to keep per-follower state and the current commit index. For this purpose, we use P4 registers that can be read and modified in the dataplane. For each node the aggregator keeps its current log index and the number of completed requests necessary for load balancing. The current log index information in the aggregator is effectively the match index kept in the Raft leader, thus the aggregator should be seen as an extension of the leader and not as a standalone entity. Our implementation uses two P4 registers for each follower, and each follower is handled in a different Tofino pipeline stage.

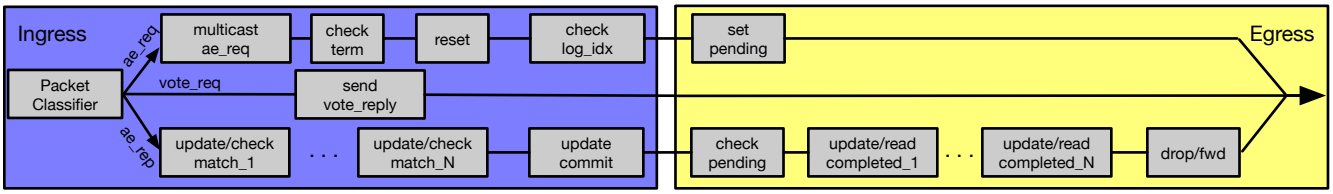


Figure 6. The in-network aggregator pipeline handling `append_entries` requests and replies

Figure 6 illustrates our implementation with the Tofino's ingress and egress pipelines. When receiving `append_entries` requests, the aggregator only has to forward the packet with a modified destination IP address set to the appropriate multicast group. That multicast group includes all nodes except for the sender. The aggregator keeps track of the current term. Receiving an `append_entries` request with a higher term will lead to the aggregator flushing its internal state.

Processing `append_entries` replies is more challenging since the aggregator needs to decide whether the log should be committed up until a certain point and multicast the `AGG_COMMIT` message to all the nodes. The `AGG_COMMIT` message should include the committed index and the number of completed requests per node. To achieve this in one pass through the dataplane, we keep the match_idx registers in the ingress pipeline and the completed_count registers in the egress pipeline. When an `append_entries` reply arrives at the aggregator, the aggregator updates the match index of the sender node, counts the number of nodes that have received up until this log index to determine whether it should commit or not, and sets its decision in per packet metadata. All replies go through egress processing, to at least update the register holding the completed requests. If the aggregator decides in ingress that the commit index is increased, it compiles an `AGG_COMMIT` reply that includes the completed requests of the followers and multicasts it to all nodes. Otherwise, it drops the reply in egress.

There are cases where the leader announces up to the same log index, which is already committed, in its `append_entries` request. This might happen either because there are no new client requests, or because a message between the aggregator and the leader was lost. In this case, the aggregator needs to forward the request to the followers to prevent a new leader election and send an `AGG_COMMIT` for an already committed log index. If the aggregator detects the same log index as in a previous `append_entries` request, it keeps track of it, and sends an `AGG_COMMIT` for the next `append_entries` reply it receives, even if the commit index is not increased. (`check_log_idx`, `set_pending` and `check_pending` stages)

The aggregator does not participate in the leader election, but it should be able to notify the new leader that it is up and ready to serve requests. Thus, the new leader, after being elected, contacts the aggregator sending a `vote_request` message. If the aggregator is up, it replies with a `vote_reply`. Note that this `vote_reply` does not count for the leader election.

Finally, Figure 6 illustrates the logic split between the ingress and egress pipelines required to meet the timing restriction of the ASIC: each stage of the pipeline can access only one register. With the Tofino v1 ASIC , HovercRaft++ can accommodate up to 9 nodes, with full line-rate processing, and without requiring any packet recirculation.

7 Evaluation

We evaluate HovercRaft and HovercRaft++ with the primary goal of showing the benefits of load balancing and in-network aggregation for datacenter SMR.

Our infrastructure is a of a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads), connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. All machines are configured with Intel x520 10GbE NICs (82599EB chipset).

All experiments use the Lancet open-source load generator, which generates an open-loop Poisson arrival process, relies on hardware timestamping for accurate RPC measurements, and reports accurately the 99^{th} percentile tail latency [57]. Lancet supports R2P2.

Our experiments compare four different system setups, all on top of DPDK:

(1) the unreplicated service (`UnRep`) is not fault-tolerant as the state-machine is not replicated. Clients interact with that single server using R2P2. This setup is expected to have the lowest latency, but it is not fault-tolerant.

(2) Our port of the vanilla Raft algorithm [15] on R2P2 and DPDK (`VanillaRaft`), which directly integrates the SMR layer within the RPC layer. This setup incorporates our design contributions from §3.1, but no protocol contributions.

(3) HovercRaft (`HovercRaft`), which incorporates protocol extensions to separate request replication from ordering (§3.2), and the ability to load-balance replies (§3.3) and read-only operations (§3.5).

(4) HovercRaft++ (`HovercRaft++`), which leverages in-network aggregation to offload protocol processing (§4). This last configuration leverages, in addition to the hardware above, a Barefoot Tofino ASIC that runs within an Edgecore Wedge100BF-32X accelerator connected to the

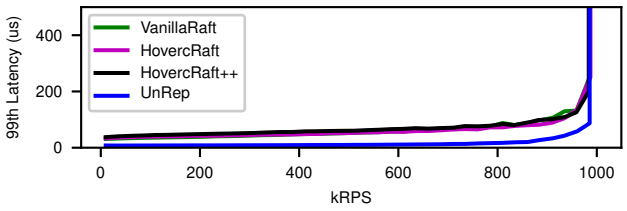


Figure 7. Tail latency vs. throughput for a fixed service time $S = 1\mu s$ workload with 24-byte requests and 8-byte replies on N=3 node cluster.

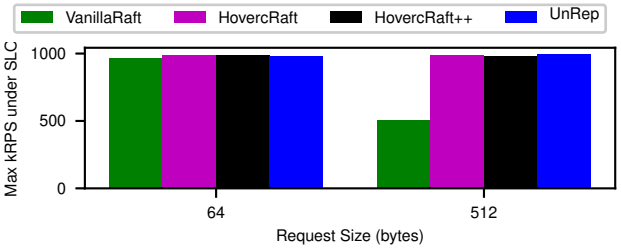


Figure 8. Achieved throughput under a 500µs SLO for a fixed service time service workload with $S = 1\mu s$, 8-byte replies and different request sizes.

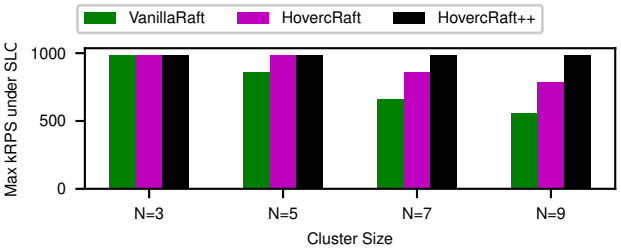


Figure 9. Achieved throughput under a 500µs SLO for a workload with fixed service time of $S = 1\mu s$, 24-byte requests and 8-byte replies for different cluster sizes.

Quanta switch via a 40Gbps link. We use the same Barefoot switch as a flow-control middlebox.

We use a combination of synthetic micro-benchmarks and a real-world application. Synthetic microbenchmarks depend on a synthetic service with configurable CPU service execution time, request, and reply sizes. Requests to this service can be either read-only or read-write. This methodology is used to determine protocol overheads in the presence of known upper bounds in terms of either CPU or I/O, and therefore to exercise the bottlenecks independently. For example, we run most experiments with a service time $S = 1\mu s$, which obviously limits the throughput to $\leq 1000KRPS$. Similarly, we run some experiments with 6KB replies, which limits throughput to $\sim\leq 200KRPS$ per 10GbE link.

The evaluation answers the following questions and quantifies the benefits of our design decisions:

1. what is the overhead of turning an RPC service into a fault-tolerant one with SMR implemented directly within the RPC layer in a 3-node cluster? (§7.1)
2. how is HovercRaft's and HovercRaft++'s performance affected by the client request size? (§7.1)
3. how do HovercRaft and HovercRaft++ scale with an increased number of nodes? (§7.2)
4. how does HovercRaft load balance replies and optimize read-only operations? (§7.3)
5. how does HovercRaft behave in the presence of failures? (§7.4)
6. how well does HovercRaft perform in practice with a production-grade application (Redis) and an industry-standard benchmark (YCSB-E)? (§7.5)

7.1 One million SMR operations per second

We first characterize all four setups on a 3-node cluster using a microbenchmark with a tiny service time ($S = 1\mu s$), minimum request size (24B) and minimum reply size (8B). There are no read-only operations to be load balanced. For this baseline experiment, we also explicitly disable the load balancing of client replies offered by HovercRaft, as to focus on protocol overheads.

Figure 7 shows the latency versus throughput curve for the four setups. We observe that there is a latency offset between the fault-tolerant configurations and the unreplicated case that comes from the extra round-trip required to achieve consensus. Nevertheless, that offset remains small and never exceeds 68µs, even for throughputs as high as $950KRPS$. Also, note that our experiment infrastructure depends on rather old hardware. Newer hardware, such as in [54], is expected to reduce this offset.

Figure 7 also shows that all four setups achieve close to the maximum possible throughput (1M RPS) under the 500µs SLO. This is a significant result as it outperforms other software-based, state-of-the-art approaches that either depend on kernel networking, such as NOPaxos [66] (by a factor of 4×), implement consensus inside the kernel, such as Kernel Paxos [31] (by a factor of 5×), or depend on RDMA [85] (by a factor of 4×). The difference is explained by our kernel-bypassed DPDK-based implementation and the leaner RPC protocol (R2P2) with its direct Raft integration.

Figure 8 adjusts the first experiment by setting the request size to 64B and 512B (compared to 24B in the first experiment) and reports the achieved client throughput in requests per second under the 500µs SLO. We observe that HovercRaft and HovercRaft++ are unaffected by the request size as they rely on multicast for request replication. However, the `VanillaRaft` configuration is sensitive to request size, with the throughput under SLO reduced by 2% and 48% for 64B and 512B sized-requests, respectively, vs 24B requests for the baseline experiment.

7.2 Scaling Cluster Sizes Without Regret

We now scale the cluster size to 5, 7, and 9 nodes *i.e.,* clusters that can tolerate 2, 3, and 4 failures, for the same experiment as the baseline in §7.1.

Figure 9 shows the achieved throughput under the 500µs SLO. In the 3-node cluster the three configurations are equivalent. The differences become obvious with larger clusters, with `VanillaRaft` most severely affected (−43% for 9 nodes). HovercRaft is unaffected up to 5 nodes, but shows a reduction with 7 and 9 nodes as the leader has to communicate independently with every follower. HovercRaft++ benefits from in-network aggregation and its performance is independent of the cluster size: the communication overhead at the leader is always the same for any number of nodes, since the replication and aggregation are performed at the P4 switch, which operates at line rate.

7.3 Scaling to Improve Performance

In the previous experiments, we showed how HovercRaft++ outperforms the other configurations even without considering its load balancing benefits. We now enable the load balancing mechanism of HovercRaft with bounded queues of up to 128 pending requests and increase the reply size to 6KB; all other parameters remain the same as the baseline. All SMR operations execute on all nodes.

Figure 10 plots the latency as a function of the achieved throughput for the unreplicated case and HovercRaft++ with 3 and 5 nodes. As expected, the unreplicated setup hits an IO-bottlenecked at ~200*KRPS*, which corresponds to a fully utilized 10G link. Running on 3 and 5 nodes, increases the capacity of the system by almost 3× and 5×, since it is an IO-bottlenecked workload and all followers reply to clients.

We now study specifically the CPU load balancing mechanisms in our design. The purpose of this experiment is to study the impact of service time variability and scheduling disciplines on performance. For this, we assume that 75% of operations are read-only. We use the baseline configuration for request and reply sizes (which is free of I/O bottlenecks), and increase the CPU service time to an average of $S = 10\mu s$. We switch from the fixed service time distribution to a bimodal distribution, in which 10% of the requests are 10x longer than the rest. Based on these parameters, the unreplicated service is expected to reach close to 100k RPS, while HovercRaft++ on a 3-node cluster will be close to 200k RPS if perfect load balancing is achieved.

Figure 11 shows the 99th percentile tail-latency as a function of the achieved throughput for the unreplicated and replicated cases. For HovercRaft++, we consider two load balancing policies, `RANDOM` and `JBSQ`, with bounded queues of 32, due to the longer service time. We observe that load balancing the read-only operations increases the CPU capacity of the system for a 57% throughput improvement under the 500µs SLO. Also, the benefit of `JBSQ` over `RANDOM` becomes obvious. `JBSQ` allows HovercRaft++ to deliver lower latency, and therefore higher throughput under SLO, since it load balances read-only requests better by avoiding overloaded followers. We expect the gap between the 2 curves to increase with the number of nodes in the cluster given the more opportunities for careful load balancing.

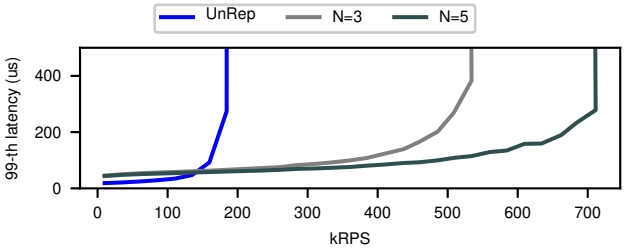


Figure 10. Latency versus throughput for $S = 1\mu s$ fixed service time, 24-byte requests, and 6kB replies for different cluster sizes.

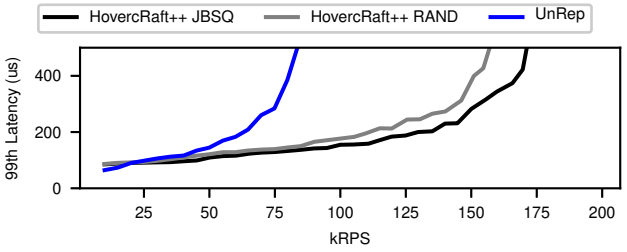


Figure 11. Latency versus throughput for a $S = 10\mu s$ service time with bimodal distribution on a 3-node cluster. Requests are 24-bytes, replies are 8-bytes, and 75% of operations are read-only.

7.4 Loadbalancing in the presence of failures

HovercRaft and HovercRaft++ enable pushing the achieved throughput load beyond the capacity of a single node by leveraging the existing redundancy. In the presence of failures, though, the capacity of the fault-tolerant system reduces. In such cases, HovercRaft and HovercRaft++ should manage the failure and gracefully degrade their performance.

In the next experiment we study how HovercRaft++ behaves when the leader fails. We use the same setup as in the previous experiment with the bimodal distribution of $\bar{S} = 10\mu s$ and 75% read-only operations. Note that the capacity of the system with this request mix is 200k RPS with 3 nodes, but drops to 160k RPS with 2 nodes. We load the system with a fixed load of 165 kRPS which is below maximum capacity for the 3-node case, but above the maximum capacity for the 2-node setup. We configure flow control to allow up to 1000 client requests in the system. We measure the

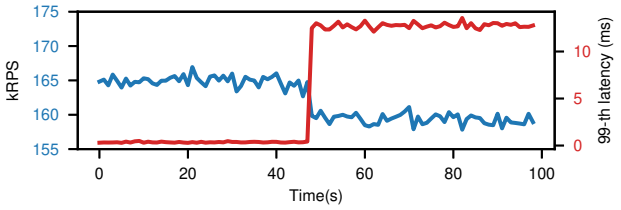


Figure 12. 99-th percentile latency and throughput as a function of time in the presence of failures a leader failure for a HovercRaft++ 3-node cluster and a bimodal distribution of $\bar{S} = 10\mu s$ with 75% read-only operations.

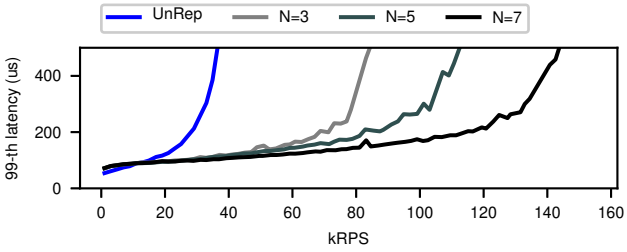


Figure 13. Latency vs throughput for YCSB-E(95% SCAN 5% INSERT) on Redis with our custom module to support YCSBE-E operations.

latency and throughput every second. At some point in time we kill the leader and we study how the system behaves.

Figure 12 plots the 99-th percentile latency and throughput as a function of time. We observe that before the leader failure the system serves 165k RPS under low latency. When the leader fails, one of the followers takes over and the system operates with 2 nodes. Because of bounded queues the new leader does not assign work to old leader. Throughput drops to the system capacity of 160k RPS. The flow control mechanism drops approximately 5k RPS maintaining the number of requests in the system below 1000, leading to increased latency, but avoiding collapse.

7.5 YCSB-E on Redis

Finally, we evaluate our design on a real-world application that requires generic SMR functionality for fault-tolerance. We run Redis [89] with the YCSB-E [19] workload.

YCSB-E is a cloud workload that consists of `SCAN` and `INSERT` operations, in a 95:5 ratio, modelling threaded conversations. It assumes 1kB records, with 10 fields of 100 bytes each. `INSERT` requests add a new record and they have to be ordered since they are parts of a conversation. `SCAN` requests query the last posts in a conversation, and they also need to be ordered for correctness, but they are read-only operations, thus they can be load balanced. We set the maximum number of elements to be returned in a `SCAN` request to 10.

Redis is an in-memory data store that supports basic data-structures and operations on them, such as lists, hashmaps, and sets. We chose Redis because it can be easily extended through user-defined modules [90]. Through Redis modules, users can define their own operations that manipulate Redis data-structures. We leverage the feature to implement the `SCAN` and `INSERT` operations of YCSB-E as single Redis operations that are guaranteed to execute within an isolated transaction. Given the support for arbitrary SMR operations, turning Redis to fault-tolerant requires an application agnostic approach similar to HovercRaft and HovercRaft++.

We ported Redis to use R2P2 instead of TCP for client operations. Beyond this protocol change, running the fault-tolerant version of Redis via `VanillaRaft`, HovercRaft, or HovercRaft++ required *no* code modifications, showing the benefits of transport layer support for SMR.

Figure 13 plots the 99^{th} percentile latency as a function of the achieved throughput for the unreplicated case and the cluster configurations for HovercRaft++ with 3, 5, and 7 nodes. YCSB-E on Redis is a CPU-bound. read-mostly workload. We observe that SMR has only a very moderate negative impact on tail latency at low loads (up to $10KRPS$), but that HovercRaft++'s ability to leverage data replication present in SMR substantially increases the achieved throughput under the 500µs SLO. In the 7-node cluster, Redis can execute $142k$ YCSB-E operations per second under SLO, while guaranteeing full state machine replication and ordering of all operations, for a speedup of 4× over the unreplicated case. This speedup of 4× is consistent with the upper bound predicted by Amdahl's law given the relative cost of `SCAN` and `INSERT`, and the fact that only `SCAN`s can be load balanced.

8 Discussion

Programmable Switches and Consistency: HovercRaft++ is not the first system to leverage programmable switches for state machine replication. However, the use of programmable switches in HovercRaft++ differs when compared to the previous work, and we believe our proposal could be used in other distributed systems mechanisms, such as byzantine fault tolerance and primary backup.

We split the proposals of using programmable switches in SMR in three main categories. The first category includes systems that use P4 switches as **sequencers**. NOPaxos [66] and Harmonia [100] take advantage of in-network compute to assign sequence numbers to client requests under very low latency and high throughput. Despite its simplicity, one drawback in this approach is handling switch failures. After a sequencer failure, the new sequencer has to make sure that it respects the request order from the previous sequencer. Thus, these systems depend on a second fault-tolerant group at the level of the network controller that maintains an epoch number, that increases at every sequencer failure, and is used to bootstrap the new sequencer.

The second category includes systems that **fully offload** the implementation of an SMR algorithm on a programmable switch. For example, Paxos made switch-y [22] runs a Paxos coordinator and acceptor inside the P4 dataplane. Although such proposals can significantly improve performance, they suffer from the P4 dataplane limitations, such using fixed-size small values.

HovercRaft++ **partially offloads** leader duties to the programmable switch by using it as a very efficient packet processor. The programmable switch deals with Raft's fan-out and the fan-in communication patterns, while maintaining only soft state. If a switch fails, a new switch can take over starting from an empty state, thus bypassing the problems in the first category. Also, using a software-based leader running in a server offers a lot of flexibility to the system, unlike the proposals from the second category.

HovercRaft and High Speed Networks: HovercRaft and HovercRaft++ go after both IO and CPU bottlenecks in SRM. Although the IO bottlenecks might become less important with the advent of faster networks, *e.g.,* 40G and 100G, the read-only operation load balancing and in-network fan-out/fan-in management employed in HovercRaft++ focus only on CPU bottlenecks and remain relevant despite bandwidth abundance. Especially, HovercRaft++ will become much more beneficial in those high speed networks since it offloads packet IO to the programmable switch and exposes a fixed overhead to the leader independent of the cluster size.

9 Related work

SMR: Consensus and state machine replication have been widely studied both from a theoretic [51, 60–63, 76, 77], and a systems point of view. Systems such as Spanner [20], Zookeeper [46], Chubby [14], etcd [33] depend on those algorithms and are widely deployed serving millions of users. Researchers have optimized consensus systems to offer SMR in a WAN environment [67, 72], inside the datacenter [66, 86], implemented within the kernel [31], using RDMA fabrics [85, 95], or on top of FPGAs [47]. Similar to NOPaxos [66] and Speculative Paxos [86], HovercRaft focuses on fault tolerance inside the datacenter assuming lossy Ethernet fabrics (rather than RDMA). Despite leaderless approaches such as Mencious [67] and EPaxos [72] deal with leader bottlenecks, they lack a global cluster view, unlike HovercRaft, thus reducing the load balancing potentials.

Scaling read-only operations: Read leases [40] proposed to optimize for read-only operations in SMR and have been used either in the form of master leases in Spanner [20], and Chubby [14], or in the form of read quorums [73]. Those approaches either overload the leader or assume application-specific knowledge. HovercRaft implements load balancing of linearizable, read-only operations in an application-agnostic manner.

µs-scale computing: Exposing the hardware potential of µs-scale interactions within a datacenter to applications requires a new approach. This approach includes datacenter-specific operating systems [9, 53, 80, 84, 87, 98], user-level networking stacks [49, 54, 56], and transport protocols [58, 71]. HovercRaft builds on the R2P2 paradigm of pushing support for RPC to the transport layer [58] by extending it to offer fault-tolerance.

Networking protocol design and implementation: The separation of request data and metadata for ordering is also used in previous systems [6, 11, 35, 52]. UDP has been proposed to increase scalability of datacenter RPCs [75]. Bounded batching has been proposed to increase throughput [9] and reduce tail latency [58]. In-network programmability and programmable switches have been used for fault-tolerance, for sequencing in NOPaxos [66], to accelerate Vertical Paxos in NetChain [50], or implementing the entire Paxos algorithm [22, 24, 94]. Such approaches either do not assume switch failures, or rely on another fault-tolerant group for state management. HovercRaft incorporates these ideas into the domain of SMR, and only stores soft-state in the network, which is recreated at every leader election.

Alternative fault-tolerance models: Fault-tolerance can be offered at the system call layer (Crane [21]), at the level of memory operations [23], in transactional databases [65, 74, 99], key-value stores [82], or for multi-threaded applications (Rex [43] and Eve [55]). HovercRaft provides fault-tolerance at the RPC layer in an application-agnostic manner and without code modifications.

10 Conclusion

We showed that replication can simultaneously improve both fault-tolerance and performance. Through the careful implementation of HovercRaft using modern kernel-bypass techniques and appropriate datacenter transport protocols, we first show that SMR is suitable for µs-scale computing, delivering 1 million ordered operations per second. Through the additional use of multicast, in-network accelerators, and load balancing, we tackle SMR's CPU and I/O bottlenecks and enable the deployment of fault-tolerant applications in an application-agnostic manner.

11 Acknowledgements

We would like to thank the anonymous PC and shadow-PC reviewers, our shepherd Marco Canini, and Jim Larus, for their valuable feedback on the paper. We would like to thank Antoine Albertelli for his initial work on the topic. This work is part of the Microsoft Swiss JRC TTL-MSR project. Marios Kogias is supported by an IBM PhD Fellowship.

References

[1] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of The 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, pages 113–119, 2019.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[3] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[4] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, 2014.

[5] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, 2011.

[6] Mahesh Balakrishnan, Ken Birman, and Amar Phanishayee. PLATO: Predictive Latency-Aware Total Ordering. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 175–188, 2006.

[7] Barefoot Networks. Tofino product brief. https://barefootnetworks.com/products/brief-tofino/, 2018.

[8] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[9] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[10] Carlos Eduardo Benevides Bezerra, Le Long Hoang, and Fernando Pedone. Strong Consistency at Scale. *IEEE Data Eng. Bull.*, 39(1):93–103, 2016.

[11] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[12] Eric A. Brewer. Pushing the CAP: Strategies for Consistency and Availability. *IEEE Computer*, 45(2):23–29, 2012.

[13] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[14] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–350, 2006.

[15] C implementation of raft. https://github.com/willemt/raft.

[16] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[17] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

[18] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2017.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[21] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 105–120, 2015.

[22] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *Computer Communication Review*, 46(2):18–24, 2016.

[23] Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. Consensus for Non-volatile Main Memory. In *Proceedings of the 26th IEEE International Conference on Network Protocols (ICNP)*, pages 406–411, 2018.

[24] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: consensus at network speed. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 5:1–5:7, 2015.

[25] Daos object store. https://github.com/daos-stack.

[26] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[28] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Commun. ACM*, 62(6):54–62, 2019.

[29] Data plane development kit. http://www.dpdk.org/.

[30] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[31] Emanuele Giuseppe Esposito, Paulo R. Coelho, and Fernando Pedone. Kernel Paxos. In *Proceedings of the 37th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 231–240, 2018.

[32] etcd ordering guarantees. https://github.com/etcd-io/etcd/blob/master/Documentation/learning/api_guarantees.md.

[33] `etcd`: Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd.

[34] `etcd` Documentation Guide: What is etcd gateway? https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/gateway.md.

[35] Pascal Felber and André Schiper. Optimistic Active Replication. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 333–341, 2001.

[36] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[37] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[38] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[39] R2p2 source code. https://github.com/epfl-dcsl/r2p2.

[40] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, 1989.

[41] Jim Gray. Fault Tolerance in Tandem Systems. In *Proceedings of the 1985 International Workshop on High-Performance Transaction Systems (HTPS)*, 1985.

[42] gRPC. http://www.grpc.io/.

[43] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of the 2014 EuroSys Conference*, pages 11:1–11:14, 2014.

[44] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 135–148, 2012.

[45] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of the ACM SIGCOMM 1995 Conference*, pages 328–341, 1995.

[46] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[47] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 425–438, 2016.

[48] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[49] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[50] 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 the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[51] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[52] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 222–230, 1991.

[53] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[54] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[55] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 237–250, 2012.

[56] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the 2019 EuroSys Conference*, pages 24:1–24:16, 2019.

[57] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 881–896, 2019.

[58] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[59] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 627–643, 2018.

[60] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

[62] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[63] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 312–313, 2009.

[64] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[65] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[66] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 467–483, 2016.

[67] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, pages 369–384, 2008.

[68] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 175–186, 2014.

[69] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[70] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[71] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[72] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[73] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 22:1–22:13, 2014.

[74] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 517–532, 2016.

[75] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[76] Brian M. Oki and Barbara Liskov. Viewstamped Replication: A General Primary Copy. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

[77] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[78] Open OnLoad. http://www.openonload.org/.

[79] Intel optane. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[80] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019.

[81] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[82] Seo Jin Park and John K. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–64, 2019.

[83] Sanjoy Paul, Krishan K. Sabnani, John C.-H. Lin, and Supratik Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, 1997.

[84] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[85] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118, 2015.

[86] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, 2015.

[87] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[88] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, pages 36–45, 2013.

[89] Redis. https://redis.io/.

[90] Redis modules. https://redis.io/topics/modules-intro.

[91] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[92] B. H. Tay and Akkihebbal L. Ananda. A Survey of Remote Procedure Calls. *Operating Systems Review*, 24(3):68–79, 1990.

[93] The tla+ homepage. https://lamport.azurewebsites.net/tla/tla.html.

[94] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The Case For In-Network Computing On Demand. In *Proceedings of the 2019 EuroSys Conference*, pages 21:1–21:16, 2019.

[95] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*, pages 94–107, 2017.

[96] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 307–320, 2006.

[97] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologie (FAST)*, pages 221–234, 2019.

[98] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of The 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, pages 73–80, 2019.

[99] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

[100] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan R. K. Ports, Ion Stoica, and Xin Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. *PVLDB*, 13(3):376–389, 2019.