Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols

Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan
The University of Edinburgh

1 Introduction

Online services and cloud applications replicate their datasets to remain available in the face of faults. Reliable replication protocols are deployed to maintain consistency among the replicas. This work focuses on the performance of strongly-consistent, fault-tolerant replication protocols for Get/Put Key-Value Stores deployed within the datacenter.

The performance of replication protocols has been repeatedly evaluated on various deployments over the years [1]. However, traditional protocol design and evaluation has not taken into account modern hardware. What do we mean by modern hardware, and why is it important when comparing the performance of protocols?

Over the last 10-15 years, the server-grade hardware landscape has changed drastically [8]. Servers with two or four cores per chip have given way to many-core chips with tens of cores, kernel-based 1 Gbps networking has given way to user-level networking with 10s or 100s of Gbps and finally, main memory has been scaled to 100s of GBs with 10s of Gbps worth of bandwidth. These advances challenge the conventional wisdom on protocol design in two ways.

Firstly, to benefit from the significant increase in hardware-level parallelism across compute, network, and memory, protocols must be multi-threaded. Indeed, a single-threaded protocol not only fails to utilize the available cores in a many-core system, but also the available network and memory bandwidth [30, 44].

Problematically, traditional protocol design has seldom considered threading; rather it has typically assumed that each node consists of a single serial process. For instance, a leader-based protocol specification typically assumes and often relies on the fact that the leader executes serially. Unsurprisingly, designing protocols without considering threading often results in non-scalable protocols.

The second aspect of protocol design challenged by modern hardware is the need (or the lack thereof) for optimizing
around the millisecond I/O speed. Specifically, protocols have traditionally been designed to: 1) reduce the number of messages per request and 2) avoid random memory look-ups which could result in disk accesses. Achieving these properties at the cost of thread-scalability or load balancing has been considered to be an acceptable trade-off. The reasoning is simple: in yesterday’s world, either of these actions costs milliseconds and can therefore skyrocket the request’s latency, resulting in user dissatisfaction and violations of the service-level agreements.

This is no longer the case, however. The hefty increase in main memory capacity has catalyzed the advent of in-memory databases [44, 46]; randomly accessing a memory object is now a nanosecond operation. Similarly, with modern, user-space and hardware-offloaded networking (e.g., RDMA), sending a message is a microsecond action [13]. Therefore, in the modern era, the protocol designer no longer needs to sacrifice properties such as thread-scalability or load balance in order to decrease latency.

In fact, in the modern era we argue that the opposite is true: in order to optimize latency, one should actually prioritize thread-scalability and load balance. Here is why. With networking and memory accounting for a few microseconds, the request latency does not typically exceed a few tens of microseconds on a lightly loaded system. Therefore, to ensure microsecond latency, we need only ensure that the system is not overloaded. This calls for high-throughput protocols as they are less likely to be overloaded by the target throughput. To maximize throughput, thread-scalability and load balance should be prioritized over traditional metrics such as number of messages per request. Our evaluation corroborates this hypothesis (§ 6).

**Research questions.** Thus far, we have argued that modern hardware has challenged conventional wisdom on protocol performance. How do protocols proposed in the literature perform on modern hardware? If one wishes to design a new protocol, what are the best practices one should adhere to?

In order to provide the answers we set out to evaluate and compare strongly-consistent replication protocols deployed on modern hardware over a state-of-the-art replicated Key-Value Store. Below we analyze the challenges in performing this study, how we tackle them and finally the contributions of this paper.

**A taxonomy for protocol selection (§3).** Firstly, it is neither feasible nor tractable to meaningfully compare every single proposed protocol. We must therefore select a few representative protocols that capture the design space, allowing us to extrapolate their results to the rest. To this end, we first develop a taxonomy of existing protocols, classifying them into four classes based on their operational patterns (Section 3). To understand the performance of the different classes of protocols, we carefully select ten protocols for analysis: ZAB [25], Multi-Paxos [39], CHT and multi-leader CHT [10], CRAQ [62], Derecho [26], Classic Paxos (CP) [36], All-Aboard Paxos [23], ABD [48] and Hermes [33].

**Odyssey: building protocols in the modern era (§4).** The second challenge is facilitating an apples-to-apples comparison that extracts maximum performance from each of these protocols on modern hardware. To overcome this challenge, we present *Odyssey*, a framework tailored towards protocol implementation for multi-threaded, RDMA-enabled, in-memory, replicated KVSes. Specifically, *Odyssey* provides the functionality to perform all the non-protocol-specific tasks, such as initializing and connecting the nodes, managing the KVS and sending/receiving RDMA messages. These tasks can account for up to 90% of the codebase for the replication protocol, requiring domain-specific knowledge in networking and KVSes. With these tasks out of the way, the developer can focus on coding solely the protocol-specific components, significantly accelerating the development process, while also producing more reliable and maintainable code. We implement all ten protocols on top of *Odyssey*.

**Comparison results (§ 6).** We answer the questions posed earlier by analyzing the results of our comparison of ten strongly-consistent replication protocols implemented over *Odyssey*. Firstly, we characterize the performance capabilities of each class of protocols along with its possible optimizations. This characterization allows us to provide an informed recommendation to those who seek to deploy an existing protocol, based on their needs. Secondly, the characterization reveals the relative importance and performance impact of properties such as thread-scalability, load balance, and the work-per-request ratio (i.e., the total cpu, network and memory resources required to complete a single request). By analyzing the effect of modern hardware on how such properties impact performance, we hope to inform the decisions of the protocol designer and steer the research community towards a more hardware-aware discussion.

**Limitations.** This work investigates the performance of strongly-consistent, fault-tolerant replication protocols for Get/Put replicated KVSes deployed within the datacenter. Note the limitations. We focus on strongly consistent protocols and not on weaker consistency models. We focus on reads and writes but not transactions. We assume a local area network and not geo-replication. Finally, we quantify the performance but not the availability guarantees of these protocols. (However, Section 3.6 discusses the qualitative impact of design decisions on availability.)

**Contributions.** Summarizing, this work presents the following contributions.

- We present a taxonomy of strongly-consistent replication protocols based on their operational patterns (§3).
- We introduce *Odyssey*, a framework that allows developers to easily design, measure and deploy replication protocols over modern hardware (§4).
• To the best of our knowledge, this paper presents the first ever implementation and evaluation of All-Aboard Paxos, CHT and CHT-multi-leader.

• Using Odyssey, we implement and evaluate ten protocols that span the design space of strongly-consistent protocols, presenting the first apples-to-apples comparison over modern hardware. Our evaluation provides a complete characterization of the replication protocol design space and reveals the impact of modern hardware on the performance of replication protocols (§6).

2 Preliminaries

Replicated Key-Value Stores. In order to remain available in the face of faults, KVSes are replicated (typically across 3 to 7 machines [25]). Note that throughout this paper the terms machines, servers, nodes and replicas are used interchangeably. We assume that clients establish connections with the replicated KVS through sessions. The order in which requests appear withina session constitutes the session order.

API. We assume that the KVS provides a Get/Put API, which we refer to as read/write. Note that writes can be conditional, i.e., they can perform an atomic read-modify-write (RMW) action on the key. Conditional writes are fundamentally harder to achieve than regular writes [21]. All of our evaluated protocols can perform conditional writes, except for multi-writer ABD [48].

Consistency. The protocols we will evaluate all enforce either one of the following two strong models: Sequential Consistency (SC) or Linearizability (lin). SC mandates that reads and writes (across all keys) from each session appear to take effect in some total order that is consistent with session order [35]. In addition to SC’s constraints, lin mandates that each request appears to take effect instantaneously at some point between its invocation and completion [22]. Note that throughout this paper we will assume the default guarantee to be lin, specifying the few cases where guarantees are downgraded to SC.

3 A Taxonomy of Replication Protocols

This section serves two purposes. First, we present a taxonomy of strongly-consistent replication protocols. The taxonomy will not only inform our choice of protocols to implement and evaluate, but will also enable us to generalize the results of each protocol to its respective class. Second, we describe the operation of various protocols, providing the background material necessary for the rest of this paper. Before diving into the taxonomy we first offer three remarks on the protocols and the corresponding jargon.

Remarks. Firstly, note that a lot of the protocols that we discuss can also execute transactions. However, this work will view them solely through the lens of the read/write API, explaining how each protocol performs a read and a write to keys stored in the replicated KVS.

Secondly, note that the problem of performing a conditional write in an environment where machines can fail and network/processing delays are unbounded is equivalent to asynchronous consensus [21]. This is why some of the protocols we are studying are known under the umbrella of “consensus protocols”. However, in this work we cast a wider net, investigating the sensitivity of performance to relaxing the fault model or to downgrading the API from conditional writes to plain writes. For that reason we refer to the protocols discussed in this paper with the general term “strongly-consistent replication protocols”.

Finally, note that throughout this paper, when we refer to a “local read”, we refer to an operation that is performed by a machine that knows it is in the configuration and hence reads from its local KVS.

3.1 Taxonomy

Our taxonomy is split into four quadrants as shown in Table 1 based on two operational patterns: 1) leader-based (L) vs. decentralized (D) and 2) total order (TO) vs. per-key order (PKO). Consequently, there are four resulting classes of protocols:

1. LTO: leader-based total order
2. LPKO: leader-based per-key order
3. DTO: decentralized total order
4. DPKO: decentralized per-key order

Table 1: Taxonomy (implemented protocols are in bold)

<table>
<thead>
<tr>
<th>Leader-based</th>
<th>Decentralized (Leaderless)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Paxos [39], ZAB [25, 57], CR [53], EPaxos [51], Atlas [14], ABD [23]</td>
<td>CRUS [54], CRUS-Paxos [59], Atlas [14], ABD [23]</td>
</tr>
<tr>
<td>CHT [10], W-Paxos [3], Primary-backup [3], CRAQ [32]</td>
<td>CHT [10], W-Paxos [3], Primary-backup [3], CRAQ [32]</td>
</tr>
</tbody>
</table>

Total order implies that protocols create a total order of all writes across all keys and apply them to the KVS in that order. In contrast, per-key order mandates that protocols only enforce a total order of writes at a per-key basis. Note that this does not affect the consistency guarantees; in both cases, protocols can offer lin. Leader-based protocols utilize a single node (i.e., a leader) to enforce the ordering of the writes, while decentralized protocols achieve the same effect in a distributed manner.

Why choose these two axes to categorize protocols? We hypothesize that from a performance perspective, protocols must optimize for three metrics: 1) thread-scalability: the protocol’s ability to scale with more threads, 2) load-balance: whether the work required to complete a request is evenly distributed among all nodes and 3) the work-per-request
ratio: the total cpu, network and memory resources required to complete a single request.

The classification is derived from the above three metrics. Specifically, total order protocols—with or without a leader—struggle to achieve thread-scalability because applying writes in order requires coordination between the threads. Leader-based protocols struggle to achieve load balance as the leader tends to carry out most of the work required to execute a write. Both techniques (leader and total order) help reduce the work-per-request ratio as they provide an easy way to serialize writes. Conversely, protocols that are both per-key and leaderless tend to require a higher work-per-request ratio because the protocols must do additional work to serialize writes in a distributed manner. We will substantiate these claims in our evaluation section (§6).

3.2 Leader-based & Total Order (LTO)

Protocols such as ZAB [25], Multi-Paxos [39] and Raft [54] serialize all writes at the leader node, creating the total order. The leader executes the writes by proposing them to the rest of the nodes (dubbed followers), typically in two broadcast rounds: a propose round to which followers respond with an acknowledgement (ack), and a commit round. All nodes must apply committed writes in their total order.

**Reads.** A write is guaranteed to propagate to only a majority of nodes. The leader is the only node that is guaranteed to be in that majority, and thus the only node guaranteed to know of the latest committed write for any key. As such, the leader can always read locally. Followers must send their reads to the leader, querying it for the latest value.

There are two possible relaxations that allow local reads in follower nodes, too. The first relaxation is to simply forego linearizability, conceding that reads may not return the latest write. This is tolerable for LTO protocols, because if writes are totally ordered, this relaxation downgrades consistency guarantees only mildly to Sequential Consistency [40]. ZAB subscribes to this practice.

The second relaxation that allows followers to read locally is to ensure that every write reaches all followers. Note that there is a downside in requiring that all writes propagate to all nodes: even if one node fails, all writes block. We elaborate in Section 3.6.

**Choices.** To represent LTO, we implement three protocols: CHT, CRAQ and a variant of CHT with multiple leaders, dubbed CHT-multi-ldr. CHT represents the typical LTO protocol, CRAQ captures the CR optimization for load balancing writes and finally, CHT-multi-ldr captures the optimization of denoting all nodes as leaders of a partition of the key space. All three protocols read locally.

3.3 Leader-based & Per-key Order (LPKO)

Protocols in this class use the leader node to only serialize writes to the same key. Specifically, all writes are steered to the leader node, which simply ensures that writes to the same key are applied in the same order by all replicas. A typical example of this class is the CHT [10] protocol, where the leader executes writes in two rounds as described in the total order class. There are two possible optimizations protocols can employ.

The first is exemplified by Chain Replication (CR) [63]. In CR, the leader does not broadcast the writes to the followers; rather the nodes are organized in a chain, through which writes propagate from the head of the chain to its tail. The head node acts as the leader in that all writes have to be steered to it so that it serializes them. In our evaluation, we will see how this approach significantly—but not entirely—alleviates the load balance problem.

The second optimization also tackles load balance, by denoting that all nodes are leaders for a subset of the keys. For example, for a 5-node deployment the key space is partitioned five ways, where each node is denoted leader for only one of the partitions. Notably, this is possible in LPKO—but not LTO—because the leader need not enforce an order across all writes.

**Reads.** LPKO protocols can execute lin reads in the same manner as LTO protocols. When writes propagate to a majority of nodes, reads have to be propagated to the leader. When writes are guaranteed to propagate to all followers, reads can execute locally in all nodes. CHT and CRAQ [62], an optimized variant of CR, both subscribe to this approach.

Finally, note that the option to propagate writes to a majority of nodes but execute reads locally by downgrading consistency to SC (discussed for LTO) is not available for per-key order protocols. Reading locally in this case would result in very weak guarantees (i.e., Eventual Consistency [64]).

**Choices.** To represent LPKO, we implement three protocols: CHT, CRAQ and a variant of CHT with multiple leaders, dubbed CHT-multi-ldr. CHT represents the typical LPKO protocol, CRAQ captures the CR optimization for load balancing writes and finally, CHT-multi-ldr captures the optimization of denoting all nodes as leaders of a partition of the key space. All three protocols read locally.

3.4 Decentralized Total Order (DTO)

In DTO protocols, the total order of writes is not created in a central location. Rather, there is typically a predetermined static allocation of write-ids to nodes. For example, all nodes know that the writes 0 to N − 1 will be proposed and coordinated by node-0, the next N writes (i.e., N to 2N − 1) will be proposed by node-1 and so on. Therefore, each node can calculate the place of each write in the total order based on its own node-id, without synchronizing with any other node. Then, the node broadcasts its writes along with their place in the total order. Typically a commit message is broadcast after gathering acks from a majority of the nodes. Crucially, all nodes must apply the writes in the prescribed total order. Derecho [26], AllConcur [56] and Mencius [49], all belong to the DTO class.
Reads. Reads can be executed by allocating slots in the total order, similarly to writes. Local reads are also possible, either by downgrading consistency guarantees to SC (similarly to LTO), or by enforcing that all writes will propagate to all nodes.

Choices. To represent DTO, we implement and evaluate Derecho. In order to get the upper bound of the DTO class, we implement the Derecho variant that executes reads locally, downgrading consistency guarantees to SC.

3.5 Decentralized Per-key Order (DPKO)

In the fourth and final quadrant, DPKO protocols agree on a per-key order of writes in a distributed manner. There is no central leader—rather any node can propose and coordinate a write. The most prominent example is Classic Paxos (CP) [36]. Traditionally, CP has been regarded simply as a way to perform leader election so that Multi-Paxos can start executing. However, recent proposals [20, 58, 59] have used CP to reach consensus on which node should be the next to perform a write at a per key basis.

Notably, CP extracts a steep price: it requires three broadcast rounds to complete (propose, accept and commit [23]), each of which contains considerably more metadata than any other protocol we have discussed, while responding to a propose or accept is also very complicated, as there are various possible responses, depending on the state of other conflicting ongoing writes. Finally, depending on conflicts, CP may have to retry an unbounded number of times [17].

The source of CP’s overhead stems from the combination of three constrains: 1) conflicting writes may be concurrently executing at all times and 2) it is impossible to guarantee that a message will always be delivered to all nodes and 3) writes are conditional (i.e., RMWs). Relaxing any of the constraints will significantly simplify the problem. Consequently, there are three approaches to optimize CP, one for each constraint. The first approach is exemplified by protocols such as EPaxos [51], Atlas [14] and All-aboard Paxos [23], which provide a fast path, where consensus can be achieved after two broadcast rounds (accept and commit), in the absence of conflicts, using CP as the fallback option when conflicts do occur.

The second approach is presented by Hermes [33], which, similarly to CR and CHT, enforces that a message will always be delivered to all nodes. With this guarantee, performing a write can be done in two lightweight broadcast rounds which are roughly equivalent to accept and commit.

Finally, the third approach downgrades the API, offering plain writes instead of conditional writes. Multi-writer ABD [48] is a variant of the ABD protocol [5] that exemplifies this approach. From now on, we refer to multi-writer ABD simply as ABD. A write in ABD requires two broadcast rounds that must reach a majority of nodes.

Reads. In DPKO protocols that do not guarantee that a write reaches all nodes, there is no master copy to read from. Therefore, to get the most recently committed write, a read must consult a majority of nodes [11]. The reads should then perform a second round to ensure that the write is committed to a majority of nodes, so that subsequent reads can also observe it. We refer to this as the ABD-read as it was first proposed in the original ABD protocol [5]. Notably, if writes are guaranteed to reach all nodes, reads can be performed locally.

Choices. To represent DPKO we implement and evaluate four protocols: CP, All-aboard, Hermes and ABD. CP will provide a baseline. All-aboard shows the limit of CP while maintaining its availability guarantees. Hermes will show us the performance gains possible when writes reach all nodes. ABD will showcase the performance difference between conditional and regular writes.

Notably, instead of All-aboard, we could have selected EPaxos [51] (or its most recent variant, Atlas [14]). EPaxos requires that nodes respond to accept messages with recent conflicting commands. This requires memory, compute and network resources to store, retrieve, reply and transmit an unbounded number of conflicting writes. In contrast, All-aboard is a zero-cost optimization. Specifically, All-aboard leverages the Flexible Paxos [24] theorem to shave off the first round (propose) and significantly reduce the size of the commit round, without incurring a counterweight cost. The complete specification of our All-aboard implementation over CP can be found in [19].

3.6 The Impact on Availability

In this section, we discuss the implications of protocol design choices on the availability guarantees.

CP, All-aboard and ABD offer the highest level of availability guarantees. Specifically, they assume the possibility of: 1) non-Byzantine machine and network failures; and 2) unbounded delays in both processing and networking. Under these assumptions, as long as $N/2 + 1$ nodes remain alive, responsive and connected, these three protocols will operate without interruption, i.e., they will remain available. The rest of the protocols that we have selected make design choices that downgrade these availability guarantees.

Table 2: A summary of the availability guarantees of the ten protocols, with up to $f$ failures (with $2f + 1$ nodes).
Specifically, leader-based protocols (ZAB, MP, CRAQ, CHT and CHT-multi-ldr) will block if the leader becomes unresponsive. Similarly, assuming that writes always reach all nodes (as in Hermes, CRAQ, CHT, and CHT-multi-ldr) results in blocking if any node becomes unresponsive. Note that assuming that writes reach all nodes is a prerequisite for linearizable local reads. Therefore, local reads can only be implemented at the expense of availability. Finally, Derecho assumes that every node makes use of their pre-allocated slots in the total order in a timely manner. If any node is slow to broadcast new writes, then all nodes will block. Table 2 provides a brief summary of the availability guarantees of the ten protocols.

In all the above cases, a failure causes blocking for the duration of a predefined time-out. Expending this time-out will trigger a recovery action (e.g., leader election, reconfiguration etc.). Once recovery is complete, operation can resume. The unavailability period is the sum of the length of the time-out plus the latency of the recovery action.

This work provides a detailed performance analysis of replication protocols without delving into the nuances of availability. However, having pointed to the choices that come at the expense of availability, we enable the operator to select (or design) the protocol that best fits their needs.

4 Odyssey

In this section, we describe Odyssey, a framework that allows developers to easily design, measure and deploy replication protocols over modern hardware. Specifically, Odyssey contains libraries to perform, among other things, the following: create and pin software threads, initialize and interface with the KVS, initialize RDMA data structures, exchange RDMA metadata to connect the servers, send and receive RDMA messages, initialize and use the RDMA multicast primitive, detect failures and maintain the configuration, specify and implement the read/write API (or create traces for benchmarking) and finally measure the performance of the system.

All ten of our protocols are implemented over Odyssey. Therefore, describing Odyssey serves a dual purpose: presenting implementation details of our evaluated protocols and describing how Odyssey can be used by the community to design and deploy new protocols.

In the rest of this section we first describe the utility of Odyssey (§4.1), and then focus on its three basic components: the threading model (§4.2), the Key-Value Store layer (§4.3) and the networking layer (§4.4).

4.1 Utility of Odyssey

The utility of Odyssey is twofold. Firstly, for the purposes of this paper, it allows us to compare strongly-consistent replication protocols over modern hardware. Secondly, once open-sourced, Odyssey can be used to develop new (or old) protocols over modern hardware. Below, we elaborate on why Odyssey is necessary to achieve either of these goals.

Protocol comparison. Odyssey facilitates an apples-to-apples comparison between strongly-consistent replication protocols over modern hardware: all our protocols use the same threading model, underlying KVS and networking patterns and optimizations. However, it is not enough for the comparison to be fair; it must also be meaningful. For that, protocols must be able to stress modern hardware to its limits. Only then will the protocol inefficiencies be exposed. For instance, Figure 1a, orders our ten protocols by their single-threaded performance; this order changes drastically when multi-threading them in Figure 1b. This is because multi-threading stresses the hardware, which in turn exposes protocol pathologies. The need to stress the hardware necessitates a framework, such as Odyssey, that targets multi-threaded, RDMA-enabled, in-memory KVSes.

Development of new protocols. The second purpose of Odyssey is to accelerate the development and deployment of replication protocols over modern hardware. Note that in most of our protocols 80 to 90% of the codebase is devoted to tasks such as setting up and using the KVS and the RDMA networking. The challenge is that, while orthogonal to protocol design, these tasks require intimate domain-specific knowledge.

To get a taste of what this knowledge entails, let us look at a specific example of a commonly occurring error when using RDMA. Assume that an RDMA message that appears to have been transmitted is never received. Also assume the developer is wise enough to check the hardware counters and detects that req_cqe_error has been incremented. In that case, the developer must know from experience that the most likely cause for this error is attempting to send a message from a memory location that has not been registered with the NIC. Absent that intimate knowledge of the RDMA universe, the developer would have to make due with the manual’s enigmatic explanation, that a “completion queue event has completed with an error” [61].

Odyssey frees the developer from all that cumbersome complexity allowing them to focus solely on the protocol. Under the hood, Odyssey uses best practices and optimizations from different domains to maximize performance.

To get a better sense of Odyssey’s utility, let us consider a concrete example in the form of Hermes over Odyssey. Was development accelerated? It took one developer less than 2 working days to develop and test our Odyssey-based Hermes. Did Odyssey practices help performance? Our Odyssey-based Hermes enjoys a 20% increase in write throughput, compared to the open-sourced version. We attribute the increase to Odyssey’s smart messages (explained in Section 4.4.3).
4.2 Odyssey Threading model
Multi-threading is a necessary step to harness the inherent parallelism in modern hardware. Here we describe how it is implemented in Odyssey.

Odyssey sets up a number of threads called workers and a number of threads called clients. Clients establish connections with the workers through sessions. Each session represents an entity (e.g., an external client, or an application thread), which issues requests (reads and writes) to the system. Each worker is typically responsible for a number of sessions. Workers are independent from each other: a worker completes each request in isolation and reports completion to the corresponding client. The order in which requests appear within a session constitutes the session order. Requests are always executed in session order.

This execution model allows Odyssey to uncover all available parallelism across unrelated requests, i.e., request-level parallelism. This is necessary in order to take advantage of the ample parallelism in today’s modern hardware. Specifically, an Odyssey-based protocol may be working on thousands of request at any given moment, by uncovering the thread-level parallelism across worker threads, and the session-level parallelism within a worker thread (as every worker is typically responsible for multiple sessions).

Developer effort. Threads are spawned and pinned transparently to the developer. The developer specifies how many workers and clients are required and provides details on the system’s resources, so Odyssey knows how to pin the threads.

4.3 Odyssey Key-Value Store
Odyssey sets up an in-memory KVS in each node, leveraging the memory capabilities of modern hardware. The KVS is largely based on MICA [46], (as found in [32]), a state-of-the-art in-memory KVS tailored for high performance. We enhance MICA with sequence locks (seqlocks) [34] to allow for concurrency control. Seqlocks allow reads to execute in a lock-free manner; writers must spin on the lock variable.

The challenge in providing a KVS as a library is that different protocols may have different requirements from the metadata stored along with each key. Some protocols may simply wish to read/write the value, but other protocols may require to read/write additional metadata. For example, when executing CP, upon receiving a propose message we may need to transition the state of the key to proposed.

Developer effort. Odyssey allows the developer to specify their own data structure to be stored in the value of a key-value pair. Furthermore, the developer must also specify the necessary handlers to process application-specific requests to the KVS. These handlers can be registered with Odyssey to be called on receiving a message.

4.4 Odyssey Networking
The third core component of Odyssey is its networking layer which allows it to leverage modern RDMA-enabled networks. In this section, we first provide an overview of the networking decisions and the effort required by the developer to use the Odyssey networking library (§4.4.1). Then we look at generic optimizations that are enabled by default (§4.4.2), and finally we describe two useful pieces of functionality that the developer can leverage: smart messages (§4.4.3) and hardware multicast (§4.4.4).

4.4.1 Networking Overview. Odyssey adopts the Remote Procedure Call (RPC) paradigm over UD Sends. Researchers have extensively proven that this paradigm comprises the most efficient and practical design point for modern RDMA-capable networks [29–32]. Below we provide an overview of how the networking layer is initialized and how it can be used to exchange messages.

Developer effort – initialization. The developer must specify the number and the nature of the logical message flows they require. In RDMA parlance each flow corresponds to one queue pair (QP), i.e., a send and a receive queue. For instance, consider Hermes where a write requires two broadcast rounds: invalidations (invs) and validations (vals). Each worker in each node sets up three QPs: 1) to send and receive invs, 2) to send and receive acks (for the invs) and 3) to send and receive vals. Splitting the communication in message flows is the responsibility of the developer. To create the QP for each message flow, the developer simply calls an Odyssey function, passing details about the nature of the QP.

Developer effort – send and receive. For each QP, Odyssey maintains a send-FIFO and a receive-FIFO. Sending requires that the developer first inserts messages in the send-FIFO via an Odyssey insert function; later they can call a send function to trigger the sending of all inserted messages. To receive messages, the developer need only call an Odyssey function that polls the receive-FIFO. Notably, the developer can specify and register handlers to be called when calling any one of the Odyssey functions. Therefore, the Odyssey polling function will deliver the incoming messages, if any, to the developer-specified handler.

4.4.2 Optimizations. Let us now overview the networking optimizations that are employed by default in Odyssey. Firstly, we limit each worker to communicate with only a single worker in every remote machine. This restriction has been shown to substantially increase performance by reducing the pressure on NIC’s hardware (caches and TLB) caused by networking metadata [18].

Furthermore, Odyssey will always batch messages in the same network packet when given the opportunity. Batch- ing more than doubles the performance when messages are small [18] by amortizing all costs associated with sending
a single packet (i.e., the packet header, DMA transactions, computation in the CPU, NIC and switch etc.).

Finally, we carefully implement low-level, well-established RDMA practices such as doorbell batching, inlining and batched selective signaling. We refer the reader to [6, 30] for more details on these optimizations.

4.4.3 Smart Messages. In this section, we describe Odyssey’s smart messages, i.e., an implementation of acknowledgements (dubbed smart-acks) and commit messages (dubbed smart-coms) that can be readily used by the developer.

Smart-acks. A smart-ack acknowledges receiving multiple messages with a fixed-size payload as long as the received messages have consecutive ids. Specifically, a smart-ack specifies 1) the first message-id it acks and 2) the number of consecutive message-ids it acks.

We call them “smart” because instead of sending an ack message for every received message, they batch multiple acks while keeping the payload fixed. The batching is opportunistic, that is, it never waits to fill a quota. In practice however, smart-acks always carry a batch because batching is used in all messages, and thus there is always a batch of messages to be acked.

Smart-coms. The idea is the same: smart-coms commit multiple writes with a fixed payload, as long as the writes have consecutive ids. Notably, smart-coms and smart-acks have great synergy, as commits are often sent after receiving acks.

Developer effort. The developer needs to make sure that messages are tagged with monotonically increasing ids. In return, they avoid the effort of implementing acks and commits. Instead, they need only call the Odyssey functions to create and send the smart messages.

We have found smart messages to be extremely useful: we have smart-acks in all ten of our protocols, and smart-coms in six of them. Besides boosting performance, smart messages significantly accelerate the time to build a protocol.

4.4.4 Hardware Multicast. Most replication protocols require broadcasting messages in order to communicate a new write to all replicas. Broadcasts are implemented in Odyssey through unicasts. However, Infiniband switches can perform a hardware-assisted multicast [7], where the sender transmits a single packet and the switch then replicates it and propagates it to all recipients. A packet always specifies the multicast-group-id that it must be transmitted to. To receive a multicast, nodes must register in the corresponding multicast group in the switch.

Odyssey contains a multicast library that will be used under the hood, if the developer specifies that a QP should use the multicast primitive. In Section 6, we investigate the types of protocols that can benefit from the hardware multicast. As far as we know, Odyssey is the first framework to offer access to the RDMA multicast.

5 Infrastructure and workload
We conduct our experiments on a cluster of 5 servers interconnected via a 12-port Infiniband switch (Mellanox MSX6012F-BS). Each machine runs Ubuntu 18.04 and is equipped with two 10-core CPUs (Intel Xeon E5-2630v4) with two hardware threads per core, reaching a total of 40 hardware threads. Furthermore, each machine has 64 GB of system memory and a single-port 56Gb Infiniband NIC (Mellanox MCX455A-FCAT PCIe-gen3 x16). We disable turbo-boost, pin threads to cores and use huge pages (2 MB) for the KVS.

Our experiments use a uniform read/write trace, which is created on each run and is kept in-memory. The KVS consists of one million key-value pairs, which are replicated in all nodes. We use keys and values of 8 and 32 bytes, respectively.

6 Evaluation
In this section, we analyze the performance of the ten protocols that we have implemented over Odyssey. We start the discussion by providing a high-level overview of the key insights of this evaluation. Then we individually analyze the performance of each class of protocols.

6.1 Overview
First, we briefly describe Figure 1 and Table 3 and then analyze our key insights and provide general directives and recommendations.

Figure 1. Figure 1 shows the throughput of all protocols in million requests per second (M.reqs/s), ordering the protocols in ascending throughput order. Specifically, Figure 1a and 1b show the write throughput of the protocols when they are single-threaded and multi-threaded (default scenario), respectively. Finally, Figure 1c shows the throughput (multi-threaded), with 95% reads.

Note the following three remarks for Figure 1. Firstly, both the x-axis and y-axis are different in all three graphs. Crucially, protocols in the x-axis are ordered in ascending throughput order. Secondly, MP and ZAB are the same protocol in the write-only workload, i.e., in Figure 1a and 1b, because they only differ in the execution of reads. Third and final, note that there is a protocol called CHT-mcast: this is the CHT protocol with the hardware multicast enabled. We show its performance separately because it performs significantly better than CHT. Enabling the multicast in the rest of the protocols has a very small impact.

Table 3. The left-hand side of Table 3 shows the throughput in M.reqs/s of all protocols when varying the write ratio. The right-hand side shows the latency (99th / average) of all protocols in microseconds at 100% write ratio, while varying the load of the protocol (i.e., with respect to peak throughput).

Let us now summarize the key insights from this study.
1. Total order is not thread-scalable. Protocols that apply writes in a total order are not thread-scalable: the relative
positions of ZAB, MP (LTO), and Derecho (DTO) in Figure 1a and Figure 1b demonstrate this point. The reason is that explicitly enforcing total order mandates that threads can only apply writes to the KVS in lock-step. In contrast, protocols that enforce per-key order (LPKO and DPKO) can scale well with more threads.

2. The leader jeopardizes load balance. The adverse effect of the leader on load balance is not apparent in LTO protocols because they cannot scale enough to uncover it. However, it is visible in LPKO protocols. Specifically, CHT does not scale well when multi-threaded because the send side of the leader becomes the bottleneck. There are two protocol-level optimizations that restore load balance: propagating writes through a chain (i.e., CRAQ) and using multiple leaders (i.e., CHT-multi-lkr).

3. Hardware multicast is effective for LPKO. The hardware multicast primitive can make a huge difference, but only in LPKO protocols. Specifically, the hardware multicast primitive provides a 3x benefit for CHT, i.e., CHT-mcast. The benefit for the rest of the protocols is very small, typically around 5%. The reason is that the multicast only relieves load on the send side of the node that performs the broadcast: it reduces the number of messages sent, but not the number of messages received. Therefore, multicast is extremely useful for leader-based protocols that are bottlenecked by the send bandwidth of the leader. It is not so useful for already well-balanced protocols (i.e., DTO and DPKO), while LTO protocols do not benefit, as they are already bottlenecked by thread-scalability. We will expand in Section 6.4.

4. DPKO excels when multi-threaded. In the absence of a leader or a total order, DPKO protocols must find creative ways to serialize writes in a decentralized manner. On the one hand, this invites a level of complexity that has an adverse effect on the work-per-request ratio. This is portrayed by the single-threaded performance of CP and All-Aboard, which is the lowest among all protocols. On the other hand, the decentralized nature of these protocols makes them naturally thread-scalable and load balanced. This is why multi-threading yields a ~9-10x throughput improvement. Notably, by downgrading the availability guarantees, as in Hermes, or downgrading the API, as in ABD, it is possible reduce the work-per-request ratio.

5. Thread-scalability > load balance > work-per-request. From Figure 1b, we observe that the non-thread-scalable protocols, ZAB, MP and Derecho are the worst performers, rendering thread-scalability the most critical property to honour in the modern era. Furthermore, All-Aboard, a protocol with a very high work-per-request ratio, significantly outperforms CHT, which sacrifices load balance, even though CHT offers lower availability guarantees (discussed in §3.6). From that we concur that it is preferable to optimize for load balance rather than work-per-request ratio. At the limits of the work-per-request ratio (i.e., in CP), the two metrics appear equally important, as CHT and CP are roughly matched.

6. Local reads are great but with caveats. Recall that MP performs reads by sending them to the leader. CP, All-aboard and ABD perform ABD-reads (typically 1 broadcast round). The rest perform reads locally. From Figure 1c, we see that there is a big gap between protocols with local reads and the rest, which perform them remotely. However there are a couple of caveats. Firstly, local reads always come at a cost as they downgrade either the consistency or the availability guarantees, as we saw in Section 3.6. Furthermore, note that ZAB, even though it performs its reads locally, is on par with the protocols that perform reads remotely. This is because it is bottlenecked by its write throughput. We elaborate in Section 6.2.

7. For better latency, choose throughput. In the Introduction, we hypothesized that a request’s latency should not exceed a few tens of microseconds in a lightly loaded system. Furthermore, we argued that to ensure a low latency, we should favour high-throughput protocols. The latency measurements for 25% load in Table 3 verify that at a light load, all protocols incur a latency of a few tens of microseconds. Furthermore, we observe that for all protocols, as load increases so does latency, with a big spike at 100% load. Therefore, to maintain a latency of a few tens of microseconds, one should
we first provide some general directives on protocol design and then offer recommendations on choosing a protocol.

**General Directives.**
- Prioritize thread-scalability, then load-balancing and then the work-per-request ratio.
- Total order should be avoided in read/write systems.
- Leader-based protocols can achieve high-performance, but care must be taken to ensure load balance.
- It is worth investing in the hardware multicast primitive only in the case of LPKO protocols.
- Local reads can deliver great performance, but it’s not guaranteed.
- In order to minimize latency, choose protocols with high throughput.

**Recommendations.**
- All-board is the most attractive design point for a scenario where: 1) availability is the most important concern and 2) conditional writes are required.
- If simple writes will do, then we recommend ABD.
- If a small window of unavailability on a failure is tolerable, then Hermes is the best candidate, while CHT-multi-ldr and CRAQ are good alternatives.

### 6.2 LTO: ZAB and Multi-Paxos

In this section, we first briefly describe the operation of our two implemented LTO protocols: ZAB and Multi-Paxos (MP). Then we focus on their results, first discussing thread-scalability for write throughput, and then the throughput when varying the write ratio.

**ZAB & MP operation.** All writes must be propagated to the leader which executes them in two broadcast rounds: a prepare round and a commit round. The difference between ZAB and MP is in reads. ZAB executes reads locally downgrading consistency guarantees to SC. MP offers lin, and so, all reads are sent to the leader.

#### Thread-scalability

The thread-scalability problem occurs when the different workers, either in the leader or the followers, try to apply the writes to the KVS. For example, the write with write-id = 200 (i.e., write-200), can only be applied after write-199 has been applied. If worker-0 is responsible for applying write-200, but not write-199, then worker-0 must wait until the worker responsible for write-199 applies it. Therefore the thread-scalability problem arises from the fact that workers can only apply their writes to the KVS in lock-step. Figure 2a shows the write-only throughput of ZAB and MP when varying the number of threads (i.e., workers). Scaling saturates at four workers. When deployed with more than 10 workers, the performance drops because the additional workers are pinned to the second socket of the server, hindering inter-thread communication.

**Throughput when varying the write ratio.** Figure 2b compares the throughput of ZAB and MP with Derecho, when varying the write ratio. ZAB’s consistency relaxation that allows for local reads pays off, as ZAB significantly outperforms MP in low write ratios.

However, note that ZAB’s write throughput does not scale well in low write ratios. For instance, at 5% write ratio, ZAB achieves 102 M_reqs/s, which means that its write throughput is roughly 5 million per sec. Ideally, since local reads are fairly cheap, one might expect that ZAB should have been able to maintain its peak write throughput (14M at 100% write ratio) at lower write ratios. Note that Derecho maintains its 16.6M write throughput at both 75% write ratio and 50% write ratio. Derecho is able to sustain its write throughput better due to its decentralized nature and thus outperforms ZAB in lower write ratios. In contrast, in ZAB (and MP), followers must send their writes to the leader which coordinates their execution. When decreasing the write ratio, the ability to batch multiple writes together into network packets and steer them into the leader is disrupted by the execution of reads, and so the write throughput cannot be maintained.

**Passive followers.** In order to examine whether it would be beneficial to spawn requests only at the leader node, Figure 2c shows the throughput of ZAB-passive-flr, a ZAB variant where followers are passive: i.e., followers are not connected with clients and thus do not initiate the execution of requests. Rather, only the leader initiates requests, while followers are only used to help coordinate writes. In this case, MP and ZAB are identical, because in both protocols reads at the leader can execute locally. ZAB-passive-flr can achieve the same write throughput as ZAB at 100% write ratio because all writes must execute at the leader anyway. However, its performance degrades as reads increase. The reason is that the single node (i.e., the leader) cannot compete with a 5-node deployment when it comes to executing local reads.

<table>
<thead>
<tr>
<th>Throughput vs. Write ratio</th>
<th>Latency vs. Load</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZAB</strong></td>
<td><strong>MP</strong></td>
</tr>
<tr>
<td>876</td>
<td>170</td>
</tr>
<tr>
<td>276</td>
<td>100</td>
</tr>
<tr>
<td>156</td>
<td>51</td>
</tr>
<tr>
<td>34</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>50%</strong></td>
<td><strong>50%</strong></td>
</tr>
<tr>
<td>119</td>
<td>95</td>
</tr>
<tr>
<td>103</td>
<td>16</td>
</tr>
<tr>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 3: Left-hand side: Throughput in M_reqs/s varying the write ratio. Right-hand side: 99th percentile and average latency (99th/avg) in μs; seconds varying the load in a write-only workload.

EuroSys ’21, April 26–28, 2021, Online, United Kingdom
Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan
Specifically, followers’ cpu and memory resources must be utilized to scale at low write ratios. Therefore active followers that are responsible for client sessions are beneficial. This result holds for LPKO protocols, too.

6.3 DTO: Derecho

We have already established the effects of the total order in write throughput and contrasted Derecho with ZAB and MP. Here we will briefly describe Derecho’s operation and comment on its performance in lower write ratios, contrasting it with two DPKO protocols.

**Derecho operation.** In Derecho, writes are totally ordered and applied in that order. The different write-ids are statically pre-allocated to different nodes. Node-0 will propose writes 0 to N – 1, node-1 will propose writes N to 2N – 1, and so on. Furthermore, Derecho performs reads locally, relaxing the consistency guarantees from lin to SC (similarly to ZAB).

**Performance.** Without considering thread-scalability, DTO is a powerful idea as the different nodes need not coordinate in order to serialize the writes. They merely need to compute the order of their own writes through their node-id and broadcast them. This is why Derecho is one of the better performing protocols in single-threaded performance (Figure 1a). However, as we saw with ZAB and MP, applying writes in a total order does not scale across many threads.

As discussed in the previous section, Derecho scales better than ZAB at lower write ratios (Figure 2b); however its low write throughput still limits its total throughput at low write ratios. For instance, when compared with Hermes (lin local reads) and CP (ABD reads) in Figure 3a, Derecho is significantly outperformed by Hermes even in low write ratios, because Hermes has a higher write throughput (due to its thread-scalability), which allows it to scale well at low write ratios. However, Derecho’s local reads allow it to outperform CP, on low write ratios, despite the fact that CP has a higher write throughput.

6.4 LPKO: CHT, CHT-multi-ldr, and CRAQ

We start the discussion of the LPKO protocols with CHT and then extend it to CRAQ.

**CHT operation.** All writes in CHT are propagated to the leader. The leader completes the writes in two broadcast rounds, similarly to ZAB and MP, with two differences: 1) it does not create a total order of all writes and 2) it waits until a write has reached all followers before committing it. The latter allows for local reads at the follower nodes. Notably, reads need to block if there is an ongoing write to the same key, until that write commits.

In CHT-multi-ldr each node is the leader for 1/N of all keys, with N being the number of nodes. Upon receiving a write request for key K, the worker finds out the leader for that key through a simple modulo operation on the key. Then, similarly to CHT, the write is propagated to its leader, which executes it to completion.

**CRAQ operation.** CRAQ organizes the nodes in a chain. All writes are steered to the head of the node, which then propagates them down the chain. When a write reaches the tail (i.e., the last node of the chain), it is said to be committed and an ack propagates back, all the way to the head. On receiving the ack, nodes commit the write. Reads are executed locally. As an optimization, reads do not block when there is an ongoing write to the same key, but instead are propagated to the tail. The tail is guaranteed to always know the latest committed write, because of its position in the chain.

**Performance.** Firstly, recall that from Figure 1b, we observed that CHT cannot balance the load and is bottlenecked by the send side of the leader, which saturates its NIC. There are three possible optimizations: using multiple leaders (CHT-multi-ldr), using a chain (CRAQ), and finally using the hardware multicast primitive (CHT-mcast).

Notably, CRAQ has the lowest impact among the three techniques, because it does not completely balance the load, as the tail does not contribute in the propagation of a write. In our 5-node deployment, the load is split between 4 nodes...
which explains why CRAQ reaches only 4/5 of the through-
put of a well-balanced protocol such as CHT-mcast.

CHT-multi-ldr also falls short of CHT-mcast. The reason
is a bit subtler. There is less opportunity to amortize cpu
and network costs in CHT-multi-ldr, because writes need
to be steered to different leaders. For example, assume that
in our 5-node deployment a worker in one of the nodes
receives 5 write requests from a client. Also assume that
each request must be steered to a different leader. The worker
cannot batch all messages to the same packet. Instead, it must
create a packet for each of the writes, sending them to the
different leaders. Furthermore the worker itself may be the
leader for one of the writes, which means it must broadcast
it, again losing the opportunity to batch it with other writes.
Conversely, in vanilla CHT, the worker would simply batch
all writes to the leader.

CHT-mcast enhances CHT with the multicast primitive.
In CHT, the send side of the leader is overloaded, because
the leader broadcasts all writes, and every broadcast requires
N unicasts (for N followers). However, the followers receive
only one message from each broadcast, and thus when the
leader utilizes 100% of its send bandwidth, the followers only
utilize 100/N% of their receive bandwidth.

CHT-mcast improves upon CHT exactly because in CHT
the followers underutilize their receive side. When the mul-
ticast primitive is used, the leader sends one message per
broadcast instead of N. The preexisting underutilization in
the followers’ side allows us to leverage the leeway created
by the multicast at the leader’s send side, to send more writes
to the followers. Had there been no room in the receive side
of the followers, the multicast would simply reduce the band-
width used at the leader send side, without improving per-
formance. In fact this is exactly what happens for most of
the broadcasting protocols (ABD, Hermes, CHT-multi-ldr,
Derecho). Notably, ZAB and MP, even though leader-based,
are not scalable enough to tap into the multicasts benefits.

Figure 3b shows the throughput of CHT-multi-ldr, CHT
and CRAQ when varying the write ratio. Firstly note that
CHT outperforms the other two for low write ratios. This
is because 1) CHT has a smaller work-per-request ratio and
2) CHT is not bottleneched by the leader’s send side at low
write ratios. CHT’s work-per-request ratio is smaller than
CRAQs, because broadcasting writes is more efficient than
propagating them through a chain, as it allows for a better
amortization of compute and network costs. CHT-multi-ldr
has an even higher work-per-request ratio than CRAQ, be-
cause as the write ratio decreases, the opportunity to amor-
tize costs by batching writes reduces, exacerbating its pre-
existing problem. This is why it is outperformed by both
CRAQ and CHT. CHT-mcast scales CHT’s throughput at
high write ratios as it avoids the bottleneck in the leader’s
send side bandwidth. As a result, its throughput is at the
highest level for all write ratios, matching that of Hermes
(Figure 3c).

6.5 DPKO: CP, All-aboard, ABD, and Hermes
Firstly we briefly explain the operation of the protocols and
then discuss their performance.

Operation. In DPKO protocols, each node coordinates its
own writes. An ABD write requires two broadcast rounds.
The first round finds out the version of the key stored in a
majority of nodes and the second sends out the new value.
An ABD read requires one broadcast round with an optional
second. The first round finds out the latest value from a
majority of nodes. If the reader cannot infer from the replies
to its first round that a majority of nodes store this value,
then it performs a second round to broadcast it. Notably,
the second round is not necessary in more than 99% of the reads.

CP requires three broadcast rounds to complete a write:
propose, accept and commit. All-aboard is an optimization
over CP, allowing a write to commit after two rounds when
there are no conflicts or slow nodes, using CP as a fallback.
Both CP and All-aboard execute reads using ABD reads.
Finally, Hermes requires two broadcast rounds to complete
a write. Its rounds are substantially more light-weight than
CP and All-aboard (and even ABD) but all messages must
always reach all nodes. For that reason, Hermes reads are
delocal.

Performance. Firstly, from Figure 1a, we observe that CP
has the lowest single-threaded performance. This is because
of the extremely high work-per-request ratio required in CP,
as explained in Section 3.5. However, CP is thread-scalable and well load balanced, enjoying a 10x improvement when multi-threaded (Figure 1b) outperforming ZAB, MP and Derecho and matching CHT.

The All-aboard optimization reduces CP’s high work-per-request but not completely. This is why All-aboard is the second worse protocol when single-threaded. Note that All-aboard has a significantly higher work-per-request ratio than Hermes and ABD, which also require two broadcast rounds. This highlights the fact that simply using the number of broadcast rounds as a metric to gauge performance is not sufficient. We need to factor in the size of the messages and the responses along with the complexity to create them.

Similarly to CP, All-aboard scales very well (10x) when multi-threaded, outperforming CP, CHT and the total order protocols. Recall from Section 3.6 that CP and All-aboard are the only two protocols (out of the ten) that can perform conditional writes while remaining available in the event of a failure. Therefore, for those keen on offering high availability, All-aboard comprises a great candidate, as it can also provide reasonably high performance.

ABD also offers the same levels of availability, but it is the only protocol out of the ten that cannot perform conditional writes. This simplification affords ABD a significantly lower work-per-request ratio than CP and All-aboard, which is why ABD outperforms CP and All-aboard both single-threaded and multi-threaded. Figure 4 compares ABD, CP and All-aboard, varying the write ratio. Notably the read throughput is equal for all three, as they all implement ABD-reads. However, as the write ratio increases, ABD outperforms the other two due to its lower work-per-request ratio for writes. Therefore, ABD comprises a great candidate, in cases where high availability is required and simple writes will suffice (as opposed to conditional writes).

Figure 3c compares ABD with Hermes (and CHT-mcast). Even though ABD is within a close distance in the write throughput, there is a big gap in the read throughput, demonstrating the cost of high availability. Specifically, Hermes mandates that every write reaches every node. In doing so, it concedes that all nodes must block on a failure (discussed in Section 3.6). However, it takes advantage of this concession in both reads and writes. In reads, by enabling them to execute locally, leveraging that all nodes have received the latest committed write. And in writes, by accelerating their operation, leveraging that a node that performs a write, has received all concurrent, conflicting writes.

This renders Hermes the better performing protocol out of all ten, making it an ideal candidate, for those who can afford an unavailability period in case of a failure.

### 7 Related Work

**Related Frameworks.** Similarly to Odyssey, Paxi [1] offers a rich interface that enables the fast development of replication protocols. However, Paxi is neither multi-threaded nor RDMA-enabled. eRPC [29] is a general-purpose networking framework offering RDMA-based RPCs, similarly to Odyssey. However, Odyssey also provides functionality tailored for replication protocols, such as the smart messages (§4.4.3).

The reason we did not use eRPC as the networking layer of Odyssey, is twofold. First, in eRPC, a broadcast requires a separate memcpy for each of the messages. In our setup that would result in multiple GBytes/s worth of unnecessary memcpy, for almost all protocols. Secondly, eRPC would not allow us to use the multicast primitive.

Finally, G-DUR [4] is a generic middleware that enables the developers to implement and evaluate a large family of distributed transactional protocols. G-DUR focuses on providing a substrate for transactional protocols that are based on the Deferred Update Replication (DUR) approach. In contrast, Odyssey focuses on exploring the impact of modern hardware in strongly-consistent replication protocols.

**Analysis of replication protocols.** Ailijiang et al. [1] dissect the performance of strongly-consistent replication protocols. Their analysis is complimentary to ours, as they focused on latency and availability on wide-area-networks and geo-replication, while we focus on performance within the datacenter and over modern hardware.

**Modern Hardware.** Odyssey investigates the interplay between protocol-level design decisions and three advances that are described as modern hardware: many-core servers, user-level high-bandwidth networking and high-capacity main memory. Notably, Szekeres et al. [60] also observe the importance of thread-scalability in the era of user-level networking, and propose the Zero-Coordination Principle a guideline to building thread-scalable replicated transactional storage systems. Furthermore, recent work [16, 27, 28, 41–43, 66] has investigated the impact of programmable hardware (FPGAs, smart NICs and switches) in deploying storage systems in the datacenter. Such programmable hardware can be used to accelerate the replication protocol. We believe that by uncovering the impact of protocol-level actions on performance our comparison of protocols can serve as a
starting point for this endeavor, guiding both the selection of protocols to accelerate and the acceleration process itself.

**Skewed workloads.** Our evaluation does not investigate the sensitivity of replication protocols under a skewed workload (e.g., zipfian distribution [52]). This is not an oversight.

It is possible to apply an optimization where reads and writes to the most popular keys (i.e., the “hot keys”) can be combined within each server by leveraging the fact that: 1) a server can efficiently keep track of the hot keys [12, 45, 50] and 2) at any given moment, a server is expected to be working on multiple requests for each of the hot keys. This optimization turns skew from problem to opportunity. This is not a surprise: researches have repeatedly observed that skew is a form of locality, and as such it can be leveraged to increase performance [15, 18, 43, 45].

Notably, the optimization is equally applicable to all ten protocols. Consequently, evaluating the protocols without the optimization would paint a false picture, suggesting that protocols suffer under skew, when in reality they can thrive under it. However, the optimization will take a different shape for each protocol. Therefore, incorporating the optimization to all ten protocols will require substantial research and we leave it for future work.

8 Conclusion and Lessons Learned

The goal of the paper is to uncover the impact of modern hardware on the performance of strongly-consistent replication protocols. To this end, we presented Odyssey, a framework that enables the fast development and deployment of replication protocols over modern hardware. Over Odyssey, we built and evaluated ten protocols. Extrapolating their results to the entire design space through an informal taxonomy, we provided a characterization of strongly-consistent replication protocols.

On the system side, we experienced first-hand the necessity for a reliable, high-performance framework to design, build and deploy replication protocols. Without it, system-level bugs (networking, KVS etc.) become a black hole for developer time. In hindsight, this is no surprise: clean interfaces that abstract orthogonal components have been the cornerstone of computer science. Nevertheless, we were pleasantly surprised to see that we can build and deploy a new protocol in two days (§4.1).

When it comes to protocol design, the overarching lesson is that the true limits of a protocol will be uncovered only when all artificially imposed bottlenecks have been removed. Plainly, this calls for highly-optimized, multi-threaded and RDMA-enabled implementations. It is very telling that ZAB outperforms Classic Paxos (CP) by more than 2x when both are single-threaded, but the result is inverted when they are multi-threaded. The pseudo bottleneck of single-thread implementations conceal ZAB’s inefficiencies while holding back CP’s capabilities. Multi-threading removes the bottleneck, laying bare the true nature of the protocols.

Acknowledgments

We would like to thank Boris Grot, the anonymous reviewers and our shepherd Vivien Quema for their valuable comments and feedback. This work was supported in part by EPSRC grant EP/L01503X/1 to The University of Edinburgh, ARM and Microsoft Research through their PhD Scholarship Programs.

References


USA, 137–152. https://doi.org/10.1145/3132747.3132756


