Microsecond Replication for Microsecond Applications

Markus Becker
TU Braunschweig
Braunschweig, Germany
markus.becker@tu-braunschweig.de

Abstract

In the space of cloud and datacenter, computing applications are expected to have extremely high availability and low latency. High availability is achieved by running applications on many machines with the ability to tolerate failures gracefully. Latency can be lowered by load-balancing requests among multiple machines as well. To achieve this, multiple approaches have been developed. On the one hand, there is a demand to enable already existing established centralized applications to run with a higher degree of safety. In these scenarios, minimizing added overhead is critical. On the other hand, newly developed applications can take advantage of the unique runtime environment present in datacenters. Potentially lowering overhead when specially designed for this environment. This runtime environment can both be a blessing and a curse as parallel execution allows for higher throughput and lower latency but requires consensus protocols to ensure consistency across different machines. With new promising networking technology like Remote Direct Memory Access new application types become realizable. In this essay, we discuss two state-of-the-art systems. Mu is a State Machine Replication system allowing existing applications to run in such an environment while adding as little latency to each request. Hermes is a protocol, which can be used to build applications taking advantage of the highly parallel environment without adding overheads commonly present in this space.

ACM Reference Format:
Markus Becker. 2020. Microsecond Replication for Microsecond Applications. In *Distributed Systems: Uncovering distributed computing principles.* ACM, New York, NY, USA, 7 pages.

1 Introduction

Microsecond applications are programs or services that require low-latency responses and actions to be taken. Especially in financial high-frequency trading, embedded computing and cloud micro-services the duration between a user requesting an action to be taken and the action being undertaken by the back-end is critical [1]. Once the request has reached the target network the required latency often lies in the range of a few microseconds. Almost equally important to these kinds of services is availability, both operationally and from a users perspective. To achieve high availability a common practice is to replicate servers responsible for handling users' requests to allow for fail-over in case of a software crash or hardware malfunction [1]. Doing so, however, requires the service provider to perform additional work to synchronize and manage the different replicated services to guarantee they perform actions as if only a single one of them were running to avoid data loss on failure or other non-consistent behavior. Specifically, we are looking for strong-consistency in replicated microsecond applications to enable fast fail-over [1, 2]. The scope of replication protocols change depending on their goals. Depending on the replication protocol and the application, integrating replication can require only little change to the application[1]. To goal of a State Machine Replication System is being almost plug-and-play. General replication protocols, instead, describe an entire communication protocol, leaving developers to implement the communication in the applications directly.

A common general application type that profits from this setup are distributed datastores. Often taking the form of key-value stores that aim to offer fast reads and writes with strong-consistency [3]. This aim comes from the fact that a single networked machine is often incapable to handle all simultaneous requests in the world of micro-services and growing number of distributed users [2]. Analysis of those systems is, however, often limited to throughput metrics, considering latency as a secondary goal or simply ignoring that metric [4].

Strong-consistency in a distributed environment can be achieved by applying consensus protocols. These protocols are historically considered to be slow, because of a lot of added communication overhead, therefore, need to be chosen and applied wisely to be able to use them in microsecond applications.

2 Background

Consensus Protocols Many parts of current computer systems are not reliable or can be attacked. Processors can crash and malfunction, networks can fail and packages can be lost. An Agreement or Consensus Protocol describes a mechanism by which a group of participants can share a state while allowing for failures to occur. The properties a consensus protocol ensures are:

1. *Safety*: Participants receive the identical data without errors and agree on the length of values. Therefore, there is no ambiguity on the completeness of the data as it is being received.
2. *Liveness*: Participants are guaranteed to receive any data eventually.

Strong-consistency is a stronger guarantee requiring that participants further agree on the order of values and have the exact same state any other does. This follows from the values being *linearizable* [5], which can be performed by some form of leader to decide the order of values, but alternate options are also explored. A part of any consensus protocol is also the consideration of possible fault modes. For this essay, unless stated differently, we assume that messages are error-free, such that if a message arrives at its destination it arrives without bit flips. Furthermore, while we allow for so-called Crash Faults, where a host becomes unable to operate at all in case of failure, we do not consider Byzantine Faults which allows a host to behave arbitrarily after a fault.

State Machine Replication State Machine Replication (SMR) describes a process by which a service can be replicated across multiple physical machines. The goal is to achieve higher availability by allowing the continued operation even if some machines fail. Furthermore, SMR is able to provide strong-consistency for any application that is a deterministic state machine. The central step in most SMRs consists of providing each machine, called replica, an identical copy of an application and some configuration data [1, 5]. Each machine then has to keep a log of all requests and performs the corresponding actions in order once consensus on the log is achieved. To ensure strong-consistency the logs of all replicas need to be identical, including the order. A consensus protocol can be used to perform the agreement on the log across all replicas, and, therefore, the performance of it contributes greatly to the overall performance, as it describes how the data is shared and how much communication has to occur.

Another desirable feature of SMR is that replicas can further replicate themselves to more physical machines by copying the initial state machine, their state and the necessary configuration to new machines, provided the log can be maintained during this operation [1].

Remote Direct Memory Access Remote Direct Memory Access (RDMA) is a network card hardware feature that allows networked machines to read and write into each other's memory directly without the involvement of the CPU of either machine. RDMA also commonly is performed over very fast and already low latency connections [6]. Data transferred by means of RDMA are received by the network card configured to immediately write to memory. Reads, similarly, instruct the remote network card to respond with data found in requested memory locations, which the local network card writes into local memory directly. There are several modes of operation, allowing for the optional generation of notifications. Before any data can be transferred using RDMA the network cards and Memory Region (MR) access permissions need to be configured. These permissions can be revoked and changed throughout operations [1]. The endpoints on the machines are called Queue Pairs (QPs) [1, 6]. Applications can then post Work Requests (WRs) on the Queue Pair which adds them to the Completion Queue (CQ). The RDMA hardware then asynchronously performs the Work Requests and adds a corresponding Work Completion event to the Completion Queue. One-sided RDMA operations do not generate Work Completion events and therefore do not require any further communication after the original Work Request was performed. For that reason, one-sided operations are often measurably faster but require more consideration as failures and data races are harder to detect [7].

3 Motivation

Ideally, any already low-latency deterministic application could be wrapped as a state machine and replicated across multiple hosts to achieve very high availability. The problem is that to allow fail-over we need to ensure that all replicas are running identically, which means that a consensus has to be reached as to which requests to process, and in which order. Furthermore, the required steps to introduce the necessary additional communication to the application, as well as the overhead added by this consensus protocol and the latency of a fail-over in case of a crash have to be investigated.

This is where different consensus protocols and State Machine Replications systems offer varying features. The purpose of an SMR system is that any application, that can be turned into a state machine, which often is the case for existing centralized deterministic client-server applications, can be replicated and profit from the added fault-tolerance and availability.

When building an application from the ground up taking a consensus protocol with features and optimizations critical for the specific application in mind can offer additional performance.

Most state-of-the-art RDMA-based replication protocols are measured in reliable datastore settings [3]. Figure 1 shows how a low latency deterministic application, that would ordinarily run on a single machine with the risk of crashing and being unavailable, can instead be spread over an arbitrary number of replicas. Users communicate with a leader replica, all other replicas become follower replicas. While the exact discussion of how this can be achieved follows later, and differs from one protocol to another, it follows from the figure that there is a need to find a consensus between the followers. To find the most general approach one would need

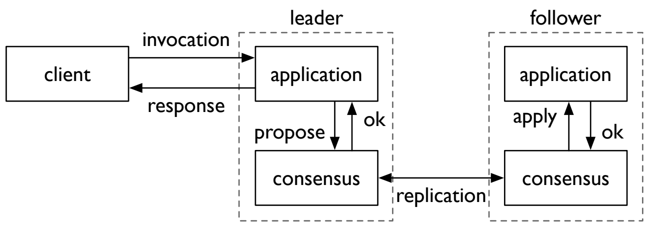

Figure 1. Application architecture in Mu redirecting communication to a leader, which in turn replicates to all followers to allow for failures [1].

to abstract away all communication between a client and an application. If done correctly the clients can be unaware whether they are talking to a single machine that never fails or the leader of an SMR System.

4 Related Works

There are several different approaches when trying to provide State Machine Replication using different consensus algorithms. Each provides a different set of guarantees while having the common goal to ensure safety and some form of liveness. The exact set of features and exact description of what a failure may entail and what liveness is to be expected are protocol-specific [8].

The goal of all the following families of protocols is to provide consensus on a set of values, where consensus in general means:

1. A valid value is chosen.
2. A single value is chosen.
3. Only a chosen value is ever committed.

We still assume to have a reliable connection, meaning messages may be slowed down, may be duplicated, connections might drop entirely, but data that is received is never corrupted [8] and was sent from, at least at the time, a correctly operating machine.

Paxos Paxos is the golden standard in providing strong-consistency, being one of the simplest crash fault-tolerant distributed system. It is criticized due to its slow throughput and high communication overhead [1]. Paxos is based on phases which each require all machines to communicate in multiple rounds for each value [8]. Due to this, it is rarely used in practice without further optimizations.

The main downsides of Paxos are that reads require communication between all replicas and writes have to make multiple round-trips throughout the network. Another downside of Paxos is that it requires the same high overhead even if no failure occurred [3].

ZAB ZAB is a state-of-the-art strong-consistent datastore finding application as the internal consensus protocol used in the Apache ZooKeeper service [9]. Its main performance gain over Paxos is that reads can be performed by any replica without having to communicate with any other replica. Writes, however, have to be serialized by a leader. The leader orders write requests, then proposes them to the followers which respond with an Ack and the leader sends another message committing to the write, once a majority is reached.

CRAQ CRAQ is also a state-of-the-art strong-consistent datastore. It tries to improve on the shortcomings of ZAB by reducing the bottleneck the single leader represents when it comes to writes. The essential change is that the replicas are linked in a chain and writes propagate the system from one end to the other. This architecture requires each replica only to write to one other replica, thus spreading the work-load over all the replicas more evenly.

DARE DARE is the first SMR system to use RDMA [10]. It was specifically developed for RDMA [11] to provide a low-latency and high-throughput advantage over similar SMR systems. This specialization enables the use of features present in RDMA and further advantages over just emulating traditional communication methods over a faster network connection [11].

HovercRaft HovercRaft is a scalable low-latency general-purpose scalable State Machine Replication system based on the Raft protocol [12] aimed at cloud deployments. Key features are the use of specialized hardware to bypass the CPU and achieving a speed-up over the non-replicated deployments by applying a load-balancer. Like Raft, HovercRaft uses the leader with which the clients communicate but tries to solve the scalability problem by accelerating the internal network by using an ASIC [12]. This *custom-hardware*-approach combined with a well load-balanced system achieves lower latency at higher node count, while avoiding the usual leader bottleneck.

APUS APUS is a scalable RDMA-based Paxos protocol. It uses State Machine Replication to enforce the same inputs for a program which runs replicated on multiple hosts [13]. APUS was the first RDMA-based Paxos protocol, which showed that RDMA was able to increase the performance of SMRs if they could be adjusted to run using the RDMA verbs.

5 Design

State Machine Replication systems aim to be deployable for arbitrary applications, provided they can be partitioned into a client and deterministic back-end. This is often the case for centralized server applications that perform actions based on incoming requests. Mu enables running such an application in a replicated fashion, providing higher availability and fault-tolerance, while trying to reduce the added overhead, especially replication latency. Furthermore, the fail-over time is also incredibly critical. One of the main causes of latency

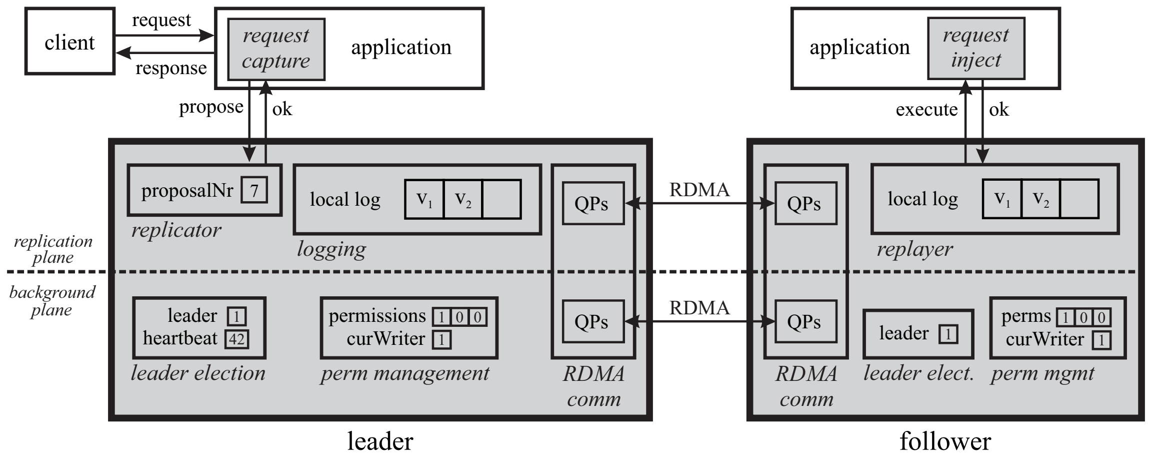

Figure 2. RDMA connected leader and follower communication as well as application architecture. This is the more detailed version of the general replicated application structure shown in figure 1

spikes come from the timeout that replicas need to have to detect the leader's failure, followed by the time it takes to determine and switch to the new leader.

Hermes, on the other hand, aims to provide a membership-based replication protocol with incredibly fast reads and writes without requiring a leader which may bottleneck a system with many replicas. The goal is to reduce overheads and enable greater parallelism, at the cost of a more involved integration into applications.

5.1 Mu

Mu aims to make use of the low latency of RDMA one-sided operations while having a very light-weight failure-detection mechanism to speed up SMR as much as possible. Mu's architecture uses SMR to create replicas. One is chosen as the leader, having additional responsibilities. The remaining replicas default to being followers. Client applications are then changed to contact the leader with all of their requests. The requests are abstracted as buffers. The leader makes decisions about the ordering of the requests and then writes the request buffers directly into the memory of all followers. This also means that every request has to enter the network of replicas through the leader. Given that the write-access to the followers' logs is exclusive to the leader, the leader can assume consensus after having successfully written the request into a majority of the followers. In the meantime, the followers are using one-sided RDMA operations to read a counter value on the leader to ensure the connection is operational and the leader hasn't crashed.

The architecture and RDMA connection pairs are shown in figure 2. Each follower has a separate RDMA Queue-Pair to read only the heartbeat of the leader. This heartbeat counter is read very frequently and indicates a crash if not incremented regularly. This is a very simple but sufficient way to perform this check. The network delay is notably irrelevant as long as the delay between reads doesn't change significantly, as the heartbeat value will still have incremented. This allows Mu to use aggressively low timeout values, reducing the time it takes to detect a failed leader [1]. The other Queue-Pair is controlled by the follower to only allow the active leader to write into the correct memory region. Mu makes use of RDMA's permission management system to ensure that only the leader is able to write requests into their log. Once a follower is notified of a completed RDMA write operation into their own memory by the leader they hand off the memory that makes up the original client's request to the replicated state machine application. Critically this is done in the order in which the requests are added to the log, making sure each followers state is identical, thus, strongly-consistent. This requires that the next message is only written out to followers if the preceding message was already accepted by the majority of followers, otherwise, a crash of the leader would leave the followers in an inconsistent state.

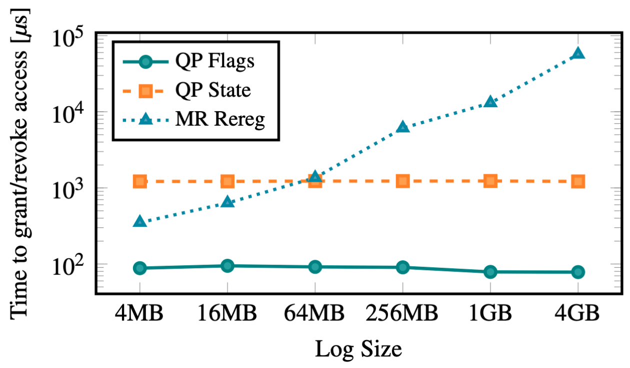

Figure 3. Permission change latency methods

RDMA offers multiple methods to manage access permissions. Aguilera et al. [1] survey the available methods with respect to the latency at which a permission change can be performed, shown in figure 3. The three compared permission change mechanics are:

1. QP Flags: Changing the access permission flags on the Queue Pair itself.
2. QP State: This operation cycles the Queue Pair through its initialization states again, resetting access permissions in the process.
3. MR Rereg: On Memory Region registration the access permissions can be specified. Once a Memory Region is unregistered that memory is still accessible to the process but the network card cannot read or write to that Memory Region anymore. But the host can re-register that Memory Region again, with different access permissions on the already existing Queue Pair. This is the fastest mechanism as the Queue Pair itself doesn't have to be changed and the memory allocation itself doesn't have to re-allocated.

They note that hardware manufacturers could reduce the permission change latency even further. Once a follower isn't able to verify the leader's correct operation, either because of an RDMA connection crash or the stagnation of the heartbeat counter of the leader they withdraw the leader's permission to write into their memory.

The leader election process is very simple in Mu: each replica is initialized with a unique numeric id, and assumes the replica with the lowest id that hasn't failed is the leader. If the client then tries to continue to communicate with the old leader it will either not be able to establish the connections (due to a complete crash) or at least not receive the "ok" message because the leader will fail in at least one write to the followers due to the changed permissions. The client is then expected to find the new leader and send future requests to that replica. How the client performs this is dependent on the application, but usually includes keeping a list of replicas and use them in order as they timeout.

Mu shares a lot of similarities with DARE with some additional optimizations in the use of RDMA, using fewer operations, and lighter leader election process [1]. In DARE the leader election process is started by a follower suspecting a leader-failure and starts a process to elect itself as the new leader [10, 11]. This leads to more communication rounds, and, therefore, a higher overhead than what Mu sets out to achieve.

5.2 Hermes

Hermes is a broadcast-based, invalidating replication protocol [3]. It provides strong-consistency, as this allows an intuitive and safer usage of datastores [3], but tries to avoid the low-performance pitfalls of other implementations, like Paxos or ZAB. The assumption used for optimization is that

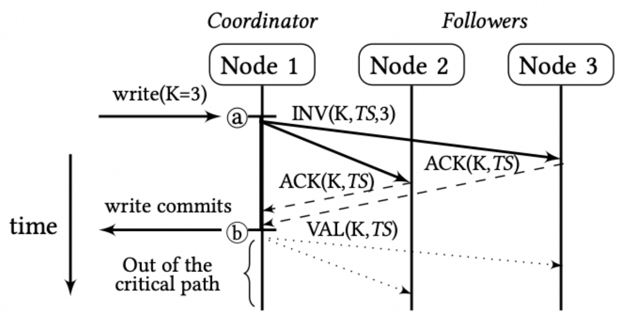

Figure 4. A successful write in Hermes. [3]

faults are not the common-case performance. Furthermore, Hermes aims to reduce inter-replica communication for reads and writes during non-failure operation. Similarly to ZAB, reads on any object can be locally handled at any replica without any further communication in the default case. The important additional feature of Hermes is that writes can also be performed by any replica.

Unlike a leader in ZAB or Mu, any replica can become the Coordinator for a write. Upon receiving a write request the replica broadcasts a message invalidating a stored object on other replicas and becomes the Coordinator for that write. The coordinator of a write attaches a logical timestamp to the Invalidation to allow ordering of Invalidations and ensure that a newer write isn't overwritten with an older one. In case two writes happen on separate replicas at the same logical timestamp the one with the lower node id *wins* and the write overwrites the one with the higher node id. A replica receiving an Invalidation with a timestamp higher than its current timestamp will respond with an Ack and update its timestamp as well as the object's value. The Coordinator commits a write once it has received a majority of Acks. Once a replica receives an Invalidation for a stored object it will not serve that object until it has received a validation from the Coordinator that a majority of followers have responded with an Ack. This process is shown in figure 4. This fault-free operation makes Hermes:

1. Decentralized: Any replica can accept writes for any object at any time. This reduces required network hops drastically and allows to load-balance writes [3] with no single bottlenecked machine.
2. Fully concurrent: Writes are processed in parallel. Without requiring a leader any number of objects could be written to in parallel.
3. Fast: Writes never abort and commit in one network round-trip. This also minimizes the write latency as after a majority of Acks are received the write can be reported as successfully performed.

In case of a failure on the Coordinator during a write a majority of replicas will be left in an Invalidated state for the written object. This Invalid state can be resolved by a write replay which is started once a read is performed on a follower with an invalid state. This follower becomes the Coordinator for a write with the stale object's data and failed writes timestamp. If it manages to receive a majority of Acks it can send Validations for that object (in the old state) and the Invalidated state of the object has been resolved. The failed write was *undone* by a write replay. The local data can then be served to respond to the read which initiated the write replay.

They present a high-performance RDMA-based reliable Key-Value store using Hermes and their own RDMA RPC library "Wings", implementing the usual Key-Value store API.

6 Evaluation

Mu was published by Aguilera et al. [1] in October 2020, and was, therefore, able to be benchmarked against Hermes directly which was published in March 2020 [3]. As Mu's main aim was to reduce latency it is not surprising to see that in their main evaluation they achieve an incredibly low latency as shown in figure 5. The application evaluated using the Hermes protocol is HermesKV which is an in-memory RDMA-based Key-Value Store by Katsarakis et al. [3]. It is the same application used by Katsarakis et al. [3] to benchmark Hermes themselves in figure 6, showing their throughput beating CRAQ, another replication protocol, and ZAB, in similar environments.

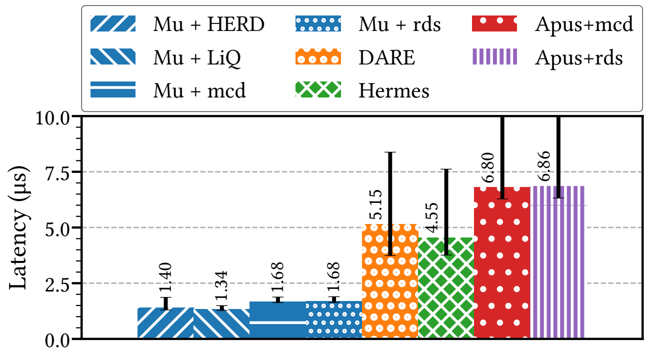

Figure 5. Replication latency for several microsecond applications replicated with Mu compared to DARE KVS, Hermes (specifically. HermesKV) as well as two KVS being replicated with APUS, with cut off error-bars.

The other benchmarked applications replicated by Mu are HERD [7], an RDMA-based Key-Value Store optimized for throughput, Memcached, a distributed in-memory Key-Value Store, Redis, in-memory Key-Value Store, and Liquibook an order matching engine for financial exchange [1]. All the applications were benchmarked on four nodes connected by a 100Gbps Switch with Mellanox Connect-X 4 links able to saturate the 100Gbps connection.

Most impressive is the small amount of changed lines of code to make these application inter-operate with Mu, having changed or added 228 lines of code at most [1]. This is comparable to the integration necessary to use APUS, however, Mu achieves better results than APUS, as APUS also applies the heavier full Paxos protocol. APUS, on the other hand, has more optional features that may be required for other deployments that were enabled in this evaluation and, therefore, APUS is not entirely replaceable by Mu.

The evaluation results clearly show that Mu performs the replication with incredibly small overhead whilst only requiring a small number of changes. HermesKV's performance in terms of latency is still competitive, however as it is purpose-built for Hermes the results are less impressive than the plug-and-play nature enabled by Mu's SMR system.

One point of criticism on part of Mu, however, is that the figure 5 error bars are cut off by the legend, and it, alongside other plots contained in the paper are hard to read, if not confusing.

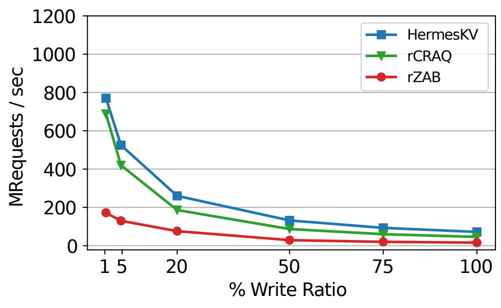

Figure 6. Hermes write rate performance study. [3]

The main advantage of Hermes over Mu is that Mu still is potentially bottlenecked by the leader's ability to serialize and replicate the requests. This leads to the concern because Mu will get slower with additional replicas more than Hermes, as more communication is necessary which has to be serialized by the leader. Thus, the network connection and speed of the leader in Mu can be the bottleneck for throughput, whereas Hermes' reads are mostly unaffected by increased node number. Writes in Hermes also only require a single network round-trip, but can be started from any replica in parallel. In its own benchmarks in figure 6, HermesKV does only slightly outperform similar Key-Value stores. Part of the explanation is that HermesKV's implementation is less optimized than the other Key-Value stores, as the focus was on Hermes the Replication Protocol itself. At low write ratios, the advantage of non-blocking load-balanced writes vanishes as compared to rCRAQ, and at high write ratios, the throughput of all Key-Value stores decline. Due to Hermes' decentralized operation, it is likely to scale better with more replicas when compared to other implementations at the same write-rate. It is also interesting to note that both rCRAQ (RDMA enabled CRAQ) and HermesKV perform so similarly, as both successfully spread the work of the bottlenecked write on the leader across all replicas using different approaches.

7 Conclusion

While there is an abundance of replication protocols, some of which were discussed in this essay, it is notable that each of them has a very different set of guarantees and optimizes for very specific application types and requirements. Furthermore, I personally am surprised by the varying scope of replication protocols: where Mu acts more like a plug-and-play drop-in to enable the advantages of replication with minimal downsides and overhead in any deterministic application. Hermes, on the other hand, required to have HermesKV built from the ground up, whereas Mu was able to be dropped in the middle of existing applications' client-server architectures and still achieve incredible performance.

Hermes offers the trade-off of allowing to increase the number of replicas if one is able to implement it, which in the space of the datacenters is important as commodity hardware is cheap but likely to fail and fault-tolerance is incredibly important. One could argue that the few microseconds advantage Mu has over Hermes are inconsequential when considering the network delays between the end-user and the datacenter.

This is all provided that one has access to the required technologies, i.e. state-of-the-art network switches, modern RDMA enabled network interface cards as well as the required fast system memory to have larger in-memory datastores. The arrival of those technologies in the end-user space is unlikely to occur very soon. Therefore, the protocols discussed are likely to stay in the datacenters for years to come.

References

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygkis, and I. Zablotchi, "Microsecond consensus for microsecond applications," 2020.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[3] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A fast, fault-tolerant and linearizable replication protocol," *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2020. [Online]. Available: http://dx.doi.org/10.1145/3373376.3378496

[4] D. Skeen, "Nonblocking commit protocols," in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '81. New York, NY, USA: Association for Computing Machinery, 1981, p. 133âĂŞ142. [Online]. Available: https://doi.org/10.1145/582318.582339

[5] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463âĂŞ492, Jul. 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[6] Mellanox, "Benefits of Remote Direct Memory Access Over Routed Fabrics," Tech. Rep., 2018. [Online]. Available: https://www.mellanox.com/related-docs/solutions/benefits-of-RDMA-over-routed-fabrics.pdf

[7] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 295âĂŞ306, Aug. 2014. [Online]. Available: https://doi.org/10.1145/2740070.2626299

[8] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[9] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systemsamp;Networks*, ser. DSN '11. USA: IEEE Computer Society, 2011, p. 245âĂŞ256. [Online]. Available: https://doi.org/10.1109/DSN.2011.5958223

[10] M. Poke and T. Hoefler, "Dare," *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC âĂŹ15*, 2015. [Online]. Available: http://dx.doi.org/10.1145/2749246.2749267

[11] M. Poke, "Algorithms for high-performance state-machine replication," Ph.D. dissertation, Helmut-Schmidt-UniversitÃďt, Holstenhofweg 85, 22043 Hamburg, 2019.

[12] M. Kogias and E. Bugnion, "Hovercraft," *Proceedings of the Fifteenth European Conference on Computer Systems*, Apr 2020. [Online]. Available: http://dx.doi.org/10.1145/3342195.3387545

[13] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: fast and scalable paxos on rdma," 09 2017, pp. 94–107.