uKharon: A Membership Service for Microsecond Applications

Rachid Guerraoui[1], Antoine Murat[1], Javier Picorel[2], Athanasios Xygkis[1], Huabing Yan[2], and Pengfei Zuo[2]

[1]*École Polytechnique Fédérale de Lausanne (EPFL)*
[2]*Huawei Technologies*

Abstract

Modern data center fabrics open the possibility of microsecond distributed applications, such as data stores and message queues. A challenging aspect of their development is to ensure that, besides being fast in the common case, these applications react fast to changes in their membership, e.g., due to reconfiguration and failures. This is especially important as they form the backbone of numerous cloud-powered services, such as analytics and trading systems, trying to meet ever-stringent tail latency requirements. As the microservices-oriented architecture is the de facto standard for building cloud services, a single user request translates to a wide fan-out of microservices interactions sitting on the critical path. The outcome is implacable: the traditionally uncommon events of reconfiguration and failures are exacerbated by the fan-out of communication, making user requests commonly experience such events and quickly impacting the tail latency of the service.

We present uKharon, a microsecond-scale membership service that detects changes in the membership of applications and lets them failover in as little as 50µs. uKharon consists of (1) a multi-level failure detector, (2) a consensus engine that relies on one-sided RDMA CAS, and (3) minimal-overhead membership leases, all exploiting RDMA to operate at the microsecond scale. We showcase the power of uKharon by building uKharon-KV, a replicated Key-Value cache based on HERD [24]. uKharon-KV processes PUT requests as fast as the state-of-the-art and improves upon it by (1) removing the need for replicating GET requests and (2) bringing the end-to-end failover down to 53µs, a 10× improvement.

1 Introduction

State-of-the-art data centers form the backbone of today's online services, including social networks, search engines, video streaming, e-commerce and banking platforms. The ever-increasing popularity of online services and their pervasive role manifest in both huge-scale requirements as well as stringent tail latency to guarantee smooth user interaction. The tail of a cloud service refers to the latency of the slowest requests, and thus provides a limit to the maximum latency experienced by the end user. Despite substantial efforts in both hardware (e.g., InfiniBand/RDMA [40], RoCE [4], FPGA [6], Gen-Z [28], CXL [50]) and hardware-accelerated software [15, 21–23, 38, 52, 53, 55], keeping the tail short at large scale is one of the most important challenges in the cloud computing industry.

Dean *et al.* [9] shed light on the challenge of building tail-tolerant software at data center scale. This challenge mainly stems from the architecture of modern online services, which are composed of a plethora of layers that communicate frequently. Despite the scalability and cost benefits of such architectures, each end-user request results in a wide fan-out of interaction across tiers, each of which lies in the critical path between the service and its reply to the user. The probability of the traditionally rare reconfiguration and failure events is thus multiplied by the fan-out of the communication. As a result, user requests encounter such events more frequently, which quickly impacts the tail latency of the services.

Existing systems are not capable of handling failures within microseconds. Key-Value stores like Hermes [26], state machine replication [44] systems like Mu [2] and Hovercraft [29], and transactional systems like FaRM [12], process requests in a few microseconds in failure-free scenarios, but miss the microsecond envelope when handling failures. Mu and HoverCraft take 0.5ms and 10ms respectively to failover. Aguilera *et al.* [2] reported that Hermes has a failover of 150ms, while FaRM mentioned ZooKeeper [20], a widely used distributed coordination service that offers at-best millisecond failover, for its membership management.

This paper builds on the observation that a crucial step in making tail-tolerant microsecond applications is reacting fast to failures. We thus propose uKharon[1], a membership service tailored to the microsecond scale. Apart from acting as a distributed membership storage for (distributed) applications, uKharon monitors their nodes, detects their failures and

[1]"u" stands for microsecond, and Kharon is the carrier of the souls of the dead in Greek mythology. It is pronounced ma · ka · ron.

changes their membership within 50µs. When uKharon itself experiences a failure, it recovers within 64µs. uKharon particularly benefits applications with efficient state transfer which can swap a faulty replica with a hot one in microseconds, for example via shadow replication. It targets cloud services that require seamless reconfiguration for fault tolerance and scalability, such as indexes, datastores and transactional systems.

The key to the performance of uKharon is the careful design of three fundamental components, all of which leverage RDMA to operate at the microsecond scale. *First*, uKharon achieves microsecond failure detection by employing a multi-level failure detector. It distinguishes the failures related to the application (e.g., segmentation faults), from those related to the kernel (e.g., driver faults), and failures related to the hardware (e.g., RDMA NIC faults), employing for each a different failure detector. *Second*, uKharon decides on memberships using a consensus engine which solely relies on one-sided RDMA verbs. This engine takes advantage of RDMA Compare-and-Swap (CAS) to handle leader changes within 10µs. *Third*, uKharon provides membership leases that add minimal overhead to the end application and last ∼20µs. As a result, our membership service combines typically opposing forces: having applications with low-overhead dynamicity in failure-free scenarios and very fast failover upon failures.

We showcase the benefits of our membership service by building uKharon-KV, a replicated in-memory KV-cache based on HERD [24]. It uses uKharon to track the set of nodes and react to node failures. We compare uKharon-KV against HERD+Mu [2] (i.e., HERD replicated by Mu), a system which—to the best of our knowledge—achieved the lowest replication latency to date. Our evaluation shows that uKharon-KV processes `PUT` requests as fast as HERD+Mu in failure-free periods. Moreover, thanks to its leasing mechanism, uKharon-KV manages to spare the replication of `GET` requests, an optimization that is algorithmically impossible in HERD+Mu. As a result, uKharon-KV `GET`s are 31.8% faster than HERD+Mu's. uKharon-KV, though, shines in the event of failures, achieving an end-to-end failover of 53µs, improving on HERD+Mu's failover of 531µs by up to a factor of 10.

In a nutshell, we present uKharon, the first ever membership service suitable for the needs of tail-tolerant microsecond applications. We make the following contributions:

- A multi-level failure detector for the microsecond scale.
- A consensus engine that relies on one-sided RDMA CAS to change leader within microseconds.
- Microsecond leases that have minimal impact on the performance of the end application.
- uKharon-KV, a replicated KV-cache which outperforms the previous state of the art.
- The source code of uKharon is available at `https://github.com/LPD-EPFL/ukharon`.

The rest of this paper is organized as follows: Section 2 introduces background concepts. Section 3 gives an overview of uKharon's design. Sections 4, 5 and 6 discuss the failure detection, consensus and leasing components, respectively. Section 7 reports on the performance of uKharon. Finally, Section 8 discusses related work and Section 9 concludes.

2 Background

2.1 Membership Service

To achieve resilience, long-lived distributed systems must be dynamic. Many systems [30, 31, 39, 45, 47] achieve dynamicity by relying on a coordination substrate, such as ZooKeeper [20] or etcd [14]. Among the various services (e.g., atomic locks, registers) these substrates offer, dynamicity is fundamentally addressed via their *membership service*.

A membership services offers dynamicity both in graceful executions and upon failures. In the former case, it serves join and leave requests issued by processes that want to become part of a distributed application or exit it. In the latter, it detects process failures and reacts to them. All these events are reflected through new configurations (called views or simply memberships). Essentially, a membership service acts as a storage of configuration information, keeping track of how the set of processes evolves, and exposes this information.

Typically, membership services rely on consensus [16] to establish a totally ordered sequence of views. Such services, including Zookeeper and etcd, offer strong semantics as all processes using the membership service transition through the same sequence of views.

Consensus-based membership services also offer real-time semantics. Apart from knowing the sequence of memberships, it is also important to know which is the (single) *active* membership. To understand why this real-time property is useful, consider the following example that incorrectly builds a cache storage solely relying on the sequence of memberships: The cache serves READ and WRITE requests. Initially, membership $M_1 = \{S_1\}$ designates server S_1 as responsible for the cache (i.e., S_1 stores it and serves requests). Eventually, a second membership $M_2 = \{S_2\}$ replaces S_1 with S_2. S_2, being part of M_2, proceeds with serving clients' requests and updates the content of the cache. At the same time, S_1 is unaware of M_2 and continues serving clients' requests as well. As a result, a client that is also unaware of M_2 and reads from S_1 will get stale data. This example demonstrates a violation of consistency. It shows that total order of memberships does not provide any real-time guarantees by itself.

Membership services provide real-timeness by making outdated memberships nonoperational. A commonly used mechanism to achieve this property is the use of a distributed invalidation protocol. Another solution is to rely on leases. With leases, processes are forced to periodically check the active membership, execute operations in this membership, and abort operations that span over multiple memberships. uKharon provides real-timeness via leases.

2.2 RDMA

Remote Direct Memory Access (*RDMA*) [49] is a networking technology that allows processes to access the memory of a remote machine without involving the CPU of the latter. By implementing several layers of the networking stack in hardware and relying on kernel bypass, RDMA achieves microsecond inter-machine communication. It allows applications within the data center to communicate in as little as 0.9μs [25]. This technology is supported by different fabrics such as Infiniband [49] and commodity Ethernet via RoCE [4].

Applications communicate over RDMA by relying on primitives called *verbs*. There exist *one-sided verbs* that include READ, WRITE and Compare and Swap (CAS) verbs and *two-sided verbs*, such as SEND and RECV verbs. One-sided verbs let a process read, write and apply atomic transformations to a remote machine's memory without involving its CPU. Two-sided verbs are similar to message passing and involve both communicating sides. They let processes send and receive memory buffers. Communication in RDMA can notably occur over established *Reliable Connections* (RCs) or over *Unreliable Datagrams* (UDs). While the former provide FIFO semantics, the latter trade reliability for better performance and support for message multicast [49].

2.3 Communication Model

uKharon is designed for data centers. It is safe under asynchrony and live under partial synchrony [13]. That is, to make progress, uKharon assumes a Global Stabilization Time (GST), unknown to the processes, such that from GST onwards there is a bound Δ on communication and processing delays. This is is a realistic assumption, as data center fabrics are not asynchronous in practice [3, 35, 54]. Additionally, our system relies on bounded clock drift for safety, i.e., time passes at approximately the same across all processes. uKharon also assumes *crash-stop* failures: processes may fail by crashing, after which they stop executing. Finally, we assume that network partitions, which affect uKharon's liveness, are eventually resolved by the data center administrators.

3 Design Overview

3.1 Architecture

Figure 1 gives an overview of uKharon. Our system, as a membership service, runs on application nodes as well as a set of dedicated nodes called *coordinators*.

Central to uKharon is uKharon Core, a single-threaded library that hosts monitoring functionalities of the membership service. This includes detecting failures of member nodes

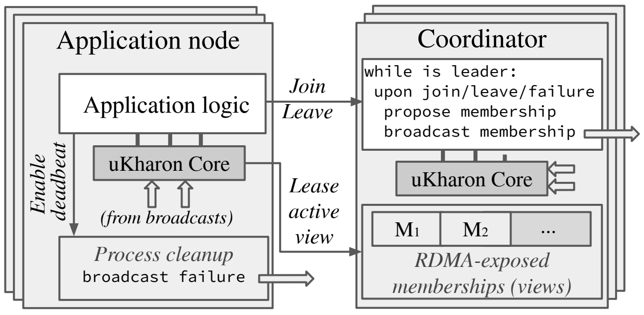

Figure 1: Overview of uKharon

(including coordinators), listening for failures and new memberships, as well as renewing leases. The application receives these events via thread-safe accessors: a stream of failures, a stream of memberships and a method `Active(M)`$\rightarrow$ *bool* which checks whether a given membership `M` is active.

The generation and storage of memberships is delegated to coordinators. Coordinators achieve fault tolerance through consensus. One of them is the leader, which processes join/leave requests from both application nodes and coordinators, proposes new memberships and broadcasts decided memberships which are picked up by the uKharon Core instance running on every node. The rest of coordinators help the leader decide and replicate the sequence of memberships. Finally, coordinators assign each member a unique identifier.

Running uKharon Core on both application nodes and coordinators helps bootstrap the membership service. uKharon Core learns about the new memberships from coordinators, but coordinators require the membership service to learn about each other. Similarly, coordinators rely on uKharon Core to detect failures of application nodes or themselves.

Part of uKharon's failure detection logic resides in the kernel, outside of uKharon Core. It consists of a kernel module hooked to Linux's process cleanup routine. This module can be enabled by the application logic and broadcasts a failure notification (called *deadbeat*) when the application crashes.

New memberships are merely broadcast by coordinators, putting the burden of detecting the active membership to the application nodes. uKharon Core is responsible for bringing real-timeness to applications. It reads the RDMA-exposed memberships at a majority of coordinators to determine whether a membership has been superseded by a new one or whether it is still active. The active membership is leased for a limited amount of time, in our case ~20µs.

3.2 Communication

uKharon relies extensively on the performance of today's RDMA-enabled fabrics to achieve its microsecond latency target. It leverages one-sided RDMA verbs, two-sided ones (i.e., HERD-style RPC [24]), as well as RDMA Multicast. Coordinators run consensus using RDMA Reliable Connections

(RCs). In particular, coordinators establish all-to-all connections among themselves and communicate using RDMA READ, WRITE and CAS. Additionally, coordinators use RDMA Multicast, which is backed by RDMA Unreliable Datagrams (UDs), to notify all nodes about new memberships. uKharon also uses RDMA Multicast to emit failure notifications. uKharon Core relies on RDMA READs over RCs to retrieve the active membership from coordinators and to detect the failure of remote nodes. Finally, processes send *join* and *leave* requests to the coordinator leader using RPC.

3.3 Challenges

Our system is designed for applications that operate and failover at the microsecond scale. To do so, uKharon meets two important design goals. First, it itself operates at the microsecond scale, meaning that it is able of changing the active membership within as few as 50µs. Second, we ensure that uKharon Core has minimal performance overhead on the end application it is bundled with. To meet these goals, uKharon is structured around three major components:

Failure detection. Efficient failure detection is the first step towards fast failover. Conventional wisdom suggests that there is a trade-off between the speed and accuracy of a failure detector. We work around this limitation by building a hierarchy of RDMA-tailored failure detectors suited for the microsecond scale. Our hierarchy detects failures within a few tens of microseconds, as we explain in Section 4.

Consensus engine. The second step of failover is agreeing on the new membership. Existing leader-based consensus engines, although optimized for the microsecond scale, struggle to change their leader at this time scale. In Section 5, we explain how our microsecond consensus engine changes leader in microseconds. This gives our design the unique property that a coordinator failure—especially failure of the coordinator leader—has negligible effect on the failover time.

Leases. As far as the membership service is concerned, the last step towards failover is updating the active membership. However, the new membership cannot become active before leases on previous memberships have expired. Thus, the longer the leases, the higher the failover time. On the other hand, short leases can result in application overhead, as they have to be checked in the application's critical path and renewed in time before expiring. In section 6, we explain how uKharon manages to have $\sim$20µs leases with virtually no cost for the end application and how leases can scale to hundreds of machines for an extra $\sim$20µs.

4 Microsecond Failure Detection

uKharon relies on microsecond failure detection to notify nodes about member failures and to trigger the generation of new memberships. In this section, we describe uKharon's failure detection scheme.

4.1 Multi-Level Failure Detection

A practical failure detector aims at being as complete and as accurate as possible. A complete and accurate failure detector is able to detect all failures and not have false positives, respectively. Completeness without accuracy causes problems in practice, as false positives trigger new memberships which require distributed applications to take further action (e.g., rebalancing data among nodes).

Commonly, failure detectors rely on timeouts for their operation. However, timeouts are hard to set correctly: if they are too low, the failure detector may experience instability (e.g., oscillating behaviors). That explains why most systems set the timeouts to a safe high-enough value. In the microsecond scale this problem is magnified, as small execution delays (e.g., kernel jitter) can take several microseconds.

Our failure detector follows a pragmatic approach: it avoids timeouts when possible. To achieve this, we are inspired by Falcon [35], and identify four levels of failures: (1) *userspace failures* (e.g., segmentation faults, out of memory errors, uncaught exceptions) that cause the application to abort, (2) *kernel failures* (e.g., cores hanging in the kernel, kernel oops caused by driver crashes) that impede the application's execution, (3) *catastrophic failures* (e.g., power failures, RDMA NIC failures) that prevent communication with the application's host, and (4) *byzantine failures* (e.g., stack overflows, mercurial cores [19]) that affect the application state. Each of the first three levels is handled by uKharon via a specialized failure detector. We do not address Byzantine failures.

4.2 uKharon's Failure Detectors

We now explain how uKharon's specialized failure detectors work, depending on the type of failure.

Userspace failures. They are handled by the Linux kernel. The application registers to the kernel to enable a *deadbeat*, which is a failure notification broadcast by the kernel upon the death of the process. This registration happens by means of the `prctl` system call that the application calls early in its execution. The system call includes the node's identifier and modifies the process descriptor (Linux's `task_struct`) with a flag that the kernel checks during the *cleaning routine* of the process. In Linux, when a process crashes, control is transferred to the kernel which starts executing the process cleaning routine. If the flag is set, the kernel broadcasts a failure notification that includes the specified identifier. To achieve this functionality, we extend the `prctl` system call and modify the process cleaning routine that is part of the kernel's `exit` system call. The task of broadcasting the crash notification is delegated to a kernel module. This module uses the kernelspace RDMA driver to broadcast crash notifications which are polled by all instances of uKharon Core. As this failure detector does not use timeouts, it has no false positives.

Kernel failures. To detect application failures caused by the kernel, we rely on the way RDMA is handled in userspace. An application registers memory to an RDMA device by issuing `ioctl` system calls on a file descriptor. By design, the Linux kernel destroys that file descriptor and thus disables remote access to this memory at the end of the process' cleaning routine. If this cleaning routine runs, the failure is caught by the previous failure detector. Otherwise, the memory will remain remotely accessible while the execution of the application is suspended (and the kernel is dying).

For the operation of this failure detector, processes are arranged in a logical ring where every process monitors its successor. Our system uses a local heartbeat counter in a similar fashion to Mu's detector [2]. uKharon Core increments this counter to indicate that the process is alive. This counter is read by the predecessor process. If a process RDMA-reads the same value twice, it reports its successor as having failed.

A process would be wrongly detected if it were unable to increment its counter between two consecutive reads. Thus, we take special care to ensure that processes always increment their counters faster than the time delay between two consecutive reads. Importantly, we deploy (the single-threaded) uKharon Core in its own dedicated physical core. We resort to a custom kernel compiled with the `NO_HZ_FULL` option, which disables regular timer interrupts [37] on the dedicated core and and thus reduces the kernel jitter towards uKharon Core. Additionally, we boot this kernel with the `isolcpus` parameter, which prevents other userspace processes from sharing the dedicated core with uKharon Core. In experiments, the interval we observed between two counter increments under heavy load was 5μs most of the time and never more than 15μs. To account for unexpected jitter (e.g., thermal throttling), we make processes wait 30μs after the completion of an RDMA READ before issuing the next one. As RDMA READs are issued sequentially, network delays do not negatively impact the accuracy of this failure detector.

Catastrophic failures. uKharon relies on a timeout-based scheme to detect failures that prevent machines from communicating. We set the timeout to 1ms, which is $2-3$ orders of magnitude higher than the common case latency of modern data center fabrics. As reported by Li *et al.* [36], 1ms is safe even in case of network congestion.

The detector works by having processes periodically broadcast a heartbeat and poll for heartbeats from others. Processes keeps track of the set of processes they recently received a heartbeat from. They compare this set with the current membership and report which processes they consider failed to the coordinator leader. Then, the leader constructs a connectivity graph based on the reported link states and changes the membership to approximately match the maximum clique in which it is included. Thus, our membership service enforces all-to-all connectivity among the members and does not expose any information regarding network partitions. A systematic treatment of network partitions is out of our scope.

The first two detectors broadcast failure notifications over RDMA-multicast, which offers better scalability than broadcasting using Reliable Connections. Nevertheless, RDMA-multicast is backed by Unreliable Datagrams, thus failure notifications can be lost under high network load. Dropping these notifications is safe, as uKharon-Core rebroadcasts a failure notification until a new membership excludes the failed node.

5 Microsecond Consensus

In this section, we present a state-of-the-art consensus engine that is tailored for the needs of uKharon and powers its coordinators. Our engine is efficient regardless of failures: in the absence of failures, it decides in one RDMA delay (by issuing an operation to a majority of processes in parallel), while it decides in one additional RMDA delay in the event of a failure. It uses a slightly modified version of Paxos based on the observation that the original algorithm contains RPCs that can be emulated with RDMA CAS operations. In the rest of the section, we intuitively describe our consensus algorithm and discuss implementation details. Appendix A provides its pseudocode and a proof of its correctness.

5.1 Consensus and Paxos

Consensus is a fundamental problem in distributed computing. Informally, each process proposes a value and eventually all processes irrevocably agree on one of the proposed values. Processes agree on a sequence of values and totally order them by running multiple instances of consensus.

Several algorithms solve consensus in the partially synchronous model. Many are variants of Paxos [32]. In Paxos, processes are divided in two groups: *proposers* and *acceptors*. Proposers *propose* a value for decision and acceptors *accept* some proposed values. Once a value has been accepted by a majority of acceptors, it is decided by its proposer.

Intuitively, Paxos is split in two phases: the *Prepare* phase and the *Accept* phase. During these phases, messages from the proposer are identified by a unique *proposal number*. The Prepare phase serves two purposes. First, the proposer gets a promise from a majority of acceptors that another proposer with a lower proposal number will fail to decide. Second, the proposer updates its proposed value using the accepted values stored in the acceptors. This way, if a value has been decided, the proposer will adopt it. The prepare phase can also *abort* if any acceptor in the majority previously made a promise to a higher proposal number. If the proposer manages to complete the Prepare phase without aborting, it proceeds to the Accept phase. In this phase, the proposer tries to store its value in a

```
# Paxos's RPCs pattern
def rpc(x):
 if compare(x, state):
  state = f(state, x)
 return proj(state)
```

```
1  def cas-rpc(x):
2   expected = fetch_state()
3   if not compare(x, expected):
4    return proj(expected)
5   move_to = f(expected, x)
6   old = state.cas(expected,
        ↪ move_to)
7   if old == expected:
8    return proj(move_to)
9   abort
```

Algorithm 1: Paxos's RPCs turned into CAS-based RPCs.

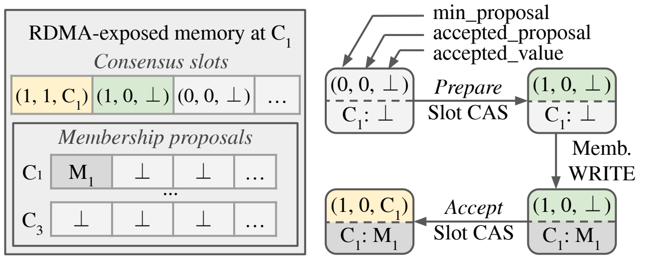

Figure 2: uKharon's Consensus Engine with its RDMA-exposed memory for multiple instances of consensus (left) and a state machine for a single instance of consensus (right).

majority of acceptors. If it succeeds (i.e., a majority accepted the value), it decides on that value.

5.2 One-Sided Paxos

Paxos uses RPC in a very specific form. The acceptors' state consists of only three variables: `min_proposal`, `accepted_proposal` and `accepted_value`. In both phases, acceptors atomically update these values based on the proposer's input and return some of them.

Algorithm 1 proposes an obstruction-free transformation to turn Paxos's RPCs into purely one-sided conditional writes using RDMA CAS. Paxos's RPCs follow the pattern seen in `rpc`. The acceptor executing the RPC compares the received value `x` to its state (stored in `state`). If the comparison is successful, the acceptor updates its state (shown with function `f`) using the provided value `x`. Finally, the acceptor unconditionally returns part of its state (shown with function `proj`).

The pattern presented in `cas-rpc` allows RDMA to emulate `rpc` while solely relying on one-sided verbs. Opposite to `rpc`, which is executed on the acceptor's side, `cas-rpc` is executed on the proposer's side. To execute the one-sided RPC, the proposer first needs to know the `state` that is stored in the memory of the acceptor. This value can either be guessed (e.g., using a previous value of `state`) or fetched (e.g., using RDMA READ, as shown in line 2). Then, the proposer executes the comparison locally (line 3) and decides whether to continue or terminate. If the comparison succeeds, the proposer proceeds with updating the state of the acceptor. It is this update that utilizes CAS [2]. In line 7, if the CAS succeeds, the acceptor's `state` has been updated successfully with the value of `move_to`. Otherwise, `state` remains unchanged.

When the RDMA CAS succeeds, i.e., in the absence of contention, both `rpc` and `cas-rpc` are equivalent (see Appendix A.2). However, if the RDMA CAS fails, `cas-rpc` will abort while `rpc` would not. In this case, `rpc` and `cas-rpc` are not equivalent, but this does not violate the correctness of Paxos. The reason is that Paxos tolerates an arbitrary number of proposer failures and that aborting the RPC and starting over is indistinguishable from such a failure.

[2]As a reminder, `variable.cas(expected, new)` atomically checks if `variable` equals `expected` and sets `variable` to `new` if this is the case. The operation always returns the initial value of `variable`.

5.3 uKharon's Consensus Engine

We now explain how to make the variant of Paxos described in Section 5.2 practical and compare it with Mu [2], a state-of-the-art consensus engine.

5.3.1 Practical Considerations

Leader election. To avoid the contention rising from multiple concurrent proposers, our consensus engine adopts the same leader election scheme as Mu. The process with the lowest identifier among the coordinators considered alive is elected as the leader. In the event of a partial network partition, this scheme can elect multiple leaders. For example, if coordinator C_2 is the only one unable to reach C_1, it will think of itself as the leader, while other coordinators will consider C_1 as their leader. Having multiple leaders cannot lead to multiple values being decided, i.e. safety is always preserved. Leader contention can, however, prevent the engine from being live. Thus, a leaders that fails to decide uses a randomized backoff before proposing until the partition is resolved.

Pre-preparation. Coordinators decide on a sequence of values by running consensus on a sequence of *slots*, as shown in Figure 2. It requires two RDMA delays for each slot: one for the Prepare and another for the Accept phase (shown with horizontal arrows in the figure). A stable leader can prepare slots in advance and only run the Accept phase to decide. In this case, the leader decides in a single RDMA delay. The leader uses the time spent waiting for the Accept phase to complete on a slot to run the Prepare phase for the next one. Thus, it always maintains one pre-prepared slot (depicted in the second consensus slot of Figure 2), with no latency overhead. Switching to the new leader requires re-preparing the next slot. As an optimization, the new leader predicts that the last slot had been prepared by the previous leader and uses this prediction as the expected value of the RDMA CAS. With this approach, the new leader manages to re-prepare the next slot in a single RDMA delay instead of two.

CAS size limitation. Algorithm 1 assumes that the consensus state fits within a single CAS. Current RDMA NICs only support CAS up to 8 bytes. We set both `min_proposal` and `accepted_proposal` to be 2 bytes each[3]. The remaining 4 bytes are dedicated to the `accepted_value`.

Our consensus engine uses indirection to overcome the limited size of the `accepted_value` and store uKharon's memberships. Instead of deciding on the membership itself, coordinators decide on its location in memory. First, the proposer RDMA-writes the membership to a part of acceptors' memory dedicated to membership proposals (see Figure 2) to which it has exclusive write access. Then, the proposer runs the Accept phase where it proposes its own identifier (C_1 in the figure). If the Accept phase succeeds at a majority of acceptors, then the proposer decides. Thanks to the FIFO semantics of RDMA RCs, if the last RDMA operation (i.e., the Accept phase CAS) succeeds, the previous RDMA operation (i.e., storing the membership with an RDMA WRITE) also succeeded. The two RDMA operations combined do not execute atomically, yet a coordinator cannot have accepted an identifier without knowing its associated membership.

5.3.2 Comparison with the State-of-the-Art

Many systems, such as Mu [2], DARE [41] and APUS [51] study consensus over RDMA. They primarily focus on improving the throughput and latency of common case executions, thus achieving consensus in a few microseconds. However, these systems have failovers ranging from 0.5ms (in Mu) to 10s or 100s of ms (in DARE and APUS, respectively).

Mu has the best performance in failure-free executions among competition as it solves consensus in $\sim$1.4µs. It relies extensively on RDMA permissions. During its Prepare phase, a proposer asks acceptors for the exclusive write permission to their memory and waits for a majority of replies. This step guarantees that only one proposer can write to an acceptor at a time. In the Accept phase, the proposer decides by merely writing to a majority of acceptors. As acceptors give write permissions to a single proposer at a time, no two concurrent proposers can successfully write to a majority of acceptors and decide on different values. Since WRITE is the most efficient RDMA verb and the Prepare phase runs only once per leader change, Mu is optimal in failure-free executions.

The Accept phase of our algorithm relies on a WRITE followed by a CAS. Importantly, these one-sided operations have lower tail latency compared with the two-sided verbs present in DARE and APUS. The CAS increases the decision time from 1.4µs to 2.9µs compared with Mu. When it comes to a leader change, Mu's permission change mechanism requires approximately 250µs, since it constitutes a control path operation that involves a system call and a reconfiguration of the NIC. In our consensus engine, the additional CAS lets coordinators change leader in under 10µs. Thus, our algorithm is designed for short tail latency and makes the failure of the coordinators' leader no more important (latency-wise) than the failure of any other node.

[3]Appendix A.6 discusses how to prevent overflows after 2^{16} failed attempts to decide on a slot by switching from CAS-based to two-sided RPCs.

6 Microsecond Real-Timeness

In addition to reacting to failures and deciding on views, uKharon lets applications track the active membership via the `Active` method. While this information is essential for consistency, it must not burden the end application. In this section, we describe the challenge of making `Active`'s overhead negligible while preserving microsecond view changes.

6.1 The Active Method

uKharon exposes real-timeness to end applications via the `Active(Membership)`$\rightarrow$`bool` method. If `Active(M)` returns `true`, we say that M is *active* at some point between the call and return of the method. `Active` satisfies three important properties. First, there are no two overlapping active memberships. Second, after a membership M is active, no memberships older than M become active. Third, the active membership converges to the latest decided membership.

Intuitively, processes use the `Active` method to determine the membership they should be executing operations in. When coordinators decide on a new membership M', a process p may stay in an older membership M due to a delay in receiving M'. Calling `Active(M)` will eventually return `false` at p, thus letting it realize that it misses the latest membership M'. To ensure consistency, an application typically calls `Active` once before starting an operation and a second time before committing it, only committing if both calls return `true`.

6.2 Leases

uKharon uses leases for efficiency. We proceed incrementally, first describing an implementation of `Active` without leases, before moving to a more efficient lease-powered scheme.

The basic implementation of `Active` requires communication in every invocation. Let M be the k-th membership decided by the coordinators and assume a process p invokes `Active(M)`. In essence, `Active` declares that M is active if it can conclude that no newer membership M' has been decided. To this end, the process RDMA-reads the $k+1$-th consensus slots at coordinators and waits for a majority of replies. If all replies are empty, then the $k+1$-th membership has not been decided, meaning that M is (still) active at some point between the invocation and return of the method. If, on the other hand, at least one of the replies is non-empty it is inconclusive whether M has been superseded by M'. In case M' has been decided before p issues the READs, then at least one of the replies must be non-empty, but the opposite is not always true. For safety, `Active` returns `false` if at least one of the READs on the next consensus slot is non-empty.

```
 1 leased_membership = ⊥; $t_{start}$ = 0; $t_{end}$ = 0

 3 def Active(M) → bool: # M is always a decided membership
 4   t = hw_timestamp()
 5   if leased_membership != M:  # First-time lease on M
 6     if majority_active(M):
 7       leased_membership = M; $t_{start}$ = t + δ; $t_{end}$ = $t_{start}$
 8   else: # Check/extend lease on M
 9     if t in [$t_{start}$, $t_{end}$): return True
10     if majority_active(M):
11       $t_{end}$ = t + δ
12       return t > $t_{start}$
13   return False
```

Algorithm 2: Leased active membership.

A lease refers to a membership and has a start and an expiration date. A lease guarantees its holder that its associated membership will remain active until it expires. In our system, leases are created by uKharon Core and last $\delta \approx 20\mu s$.

Algorithm 2 provides an efficient alternative implementation of `Active` that relies on leases to reduce communication. It starts by taking a hardware timestamp t (line 4) and then checks if a lease on M already exists (line 5). If no lease exists (lines 6-7), the method checks for a newly decided membership by contacting a majority of coordinators. If no membership newer than M could have been decided (i.e., all replies are empty), it creates a lease on M (line 7) that starts at $t + \delta$ and has no duration. This prevents overlapping active memberships since any lease that processes could hold on a previous membership $M' < M$ will have expired before M becomes active. In case a lease on M already exists, the method tries to use it in order to avoid reaching the coordinators (line 9). If it cannot use it, it tries to extend the lease (line 11) by checking the coordinators. It returns `True` only if leases on previous memberships have expired (line 12), which takes—in the worst case—δ to happen. As a result, leases affect the speed at which memberships can change, justifying the desire for a small lease duration. Section 7 demonstrates that leases of $\delta \approx 20\mu s$ are feasible in practice.

This efficient implementation of `Active` renews its lease on demand. As long as its lease is valid, the method merely takes a hardware timestamp—which takes a few tens of nanoseconds—and returns immediately without reaching the coordinators. The latency overhead of `Active` to the application that invokes it is thus very low. Communication with the coordinators is only necessary when leases expire and have to be renewed, which results in a spike in `Active`'s latency. In practice, uKharon Core renews leases in the background to ensure that—when the membership remains unchanged—`Active` is not delayed by the calls to `majority_active`.

uKharon does not rely on operational leases for either liveness or safety. Timely renewal of leases is only a way to reduce the latency of `Active` as Algorithm 2 would work even with zero-duration leases. uKharon relies on bounded clock drifts for safety, as opposed to clock synchronization. This ensures that durations are approximately the same across all processes, thus preventing overlapping memberships. Appendix C includes a microbenchmark evaluating the clock drift of actual hardware and gives an overestimated drift that is no more than 0.001% of the lease duration. Thus, clock drift is accounted for by making leases last a few nanoseconds less than their nominal value. As drift is reset on each lease renewal, it does not accumulate over time. Therefore, no matter how long a system is up for, its operation remains unaffected by the clock drift. A proof of correctness of uKharon's leases is given in Appendix B.

6.3 Extensions

Adaptive leases. So far, we have assumed a fixed lease duration δ. Network delays greater than δ render leases useless as, every time the lease is extended (line 11), t_{end} is always in the past. In this case, `Active` always contacts the coordinators. In order to work under partial synchrony and avoid this scenario, we extend the leasing mechanism as follows: Coordinators store the lease duration for a given membership along with the membership itself. An application node that wants to increase the lease duration contacts the coordinator leader. This results in a new *compatible* membership that is identical to the previous one apart from the lease duration. Compatible memberships receive special handling by uKharon Core in order to ensure that—when going from one compatible membership to another—`Active` does not wait for leases on the previous membership to expire. Also, if the latest membership M is not compatible with the previous one, invocations to `Active(`M`)` return false until all possibly ongoing leases on previous memberships have expired.

Lease caches. `Active` reaches a majority of coordinators to renew its lease, which scales badly as the number of application nodes increases. uKharon solves this issue with an intermediate lease renewal layer, the *lease caches*. These caches use the `Active` method to lease memberships for Δ (by reading from a majority of coordinators). In turn, application nodes use leases that last for δ and a modified version of `Active`. This version differs from the one presented in Algorithm 2 in the `majority_check` calls, which are replaced with RPCs to a *single* lease cache. As a result, application nodes reduce the communication cost required to renew their lease by a factor of—at least—3 (the typical number of coordinators). However, lease caches increase the failover time of applications by at least Δ. The reason is that when the coordinators change the membership, the `Active` method of caches waits Δ before making the new membership active. At the same time, the `Active` method of application nodes that is directed to some lease cache, waits δ before making the new membership active. Thus, the overall time from the moment a new membership is decided until application nodes start using it jumps from (at least) δ to (at least) $\Delta + \delta$.

7 Evaluation

We evaluate the various performance traits of uKharon and verify its suitability as a membership service for microsecond applications. We aim to answer the following:

- How much does uKharon increase the latency of end applications and what is its impact on their throughput?
- How fast does uKharon respond to failures?
- How can uKharon be leveraged to build replication protocols and what performance can they achieve?

CPU	2x Intel Xeon Gold 6244 CPU @ 3.60GHz (8 cores/16 threads per socket)
NIC	Mellanox ConnectX-6 MT28908
Switch	Mellanox MSB7700 EDR 100 Gbps
OS/Kernel	Ubuntu 20.04.2 / `5.4.0-74-custom`
RDMA Driver	Mellanox OFED `5.3-1.0.0.1`

Table 1: Hardware details of machines.

We evaluate uKharon in a 8-node cluster, the details of which are given in Table 1. The custom kernel sets the `NO_HZ_FULL` option and uses the `isocpus` boot parameter, as explained in Section 4.2. Our dual-socket machines are NUMA and their RDMA NIC lies on the first socket. For this reason, we ensure that all threads during our experiments execute on cores of the first socket. We also make all threads exclusively use the memory bank closest to this socket.

Our implementation measures time durations using the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. The function uses the TSC clocksource of the Linux kernel, which offers efficient and accurate timestamping [43]. Appendix C discusses details regarding the drift and synchrony of TSC in symmetric multiprocessing (SMP) systems.

Finally, in all experiments we deploy 3 coordinators.

Applications. We integrate uKharon with HERD [24]. HERD is a non-replicated microsecond-scale RDMA-based KV-cache. Clients send requests to a HERD server by RDMA-writing to a dedicated buffer that the server has allocated for them. Requests contain an 8-byte key and are either `PUT`s or `GET`s. `PUT`s additionally contain the value to be stored for the specified key. The server discovers new client requests by polling its local memory, executes the requests locally and then replies to the clients using RDMA UDs. We also leverage uKharon to build uKharon-KV, an extended version of HERD which supports replication. We compare our solution with HERD replicated by Mu (HERD+Mu) [2] which—as far as we know—offers the lowest replication latency to date.

Implementation effort. We implemented uKharon on top of our own RDMA framework. uKharon Core and the consensus engine span 4448 and 1324 lines of C++, respectively. The kernel module of the deadbeat failure detector is 404 lines of C. uKharon-KV extends HERD by 1498 lines of C++. The only unimplemented features are clique-based memberships (Section 4.2) and adaptive leases (Section 6.3).

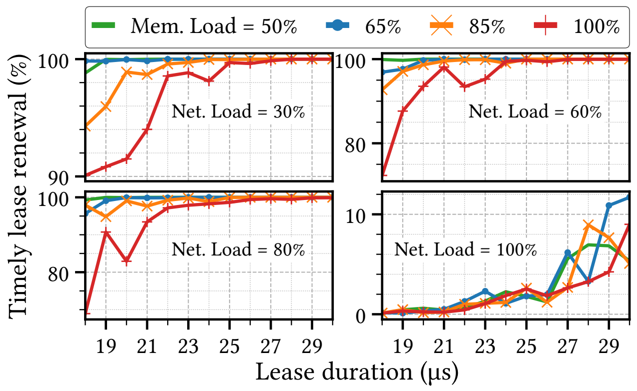

Figure 3: Percentage of timely lease renewal depending on the lease duration, network load and memory load.

7.1 Overhead Induced by uKharon

Latency overhead. Applications bundled with uKharon Core rely heavily on its `Active` method. As long as (the background running) uKharon Core renews the lease on the active membership in time, the `Active` method adds negligible latency overhead to the application. We experimentally determine that the 99th percentile latency for invoking `Active` is 38ns when the lease is renewed in time, which is the time it takes to fetch the hardware timestamp and compare it with the expiration date of the lease. Fluctuations in the network's latency or execution delays when uKharon Core renews the lease (e.g., due to cache misses) induces additional latency to the application, as explained in Section 6.2.

Figure 3 shows how the duration of leases affects their timely renewal. We run 1-minute experiments under a steady membership with 32 lease renewers contacting coordinators directly and lease durations ranging from 18 to 30µs. Each machine has a maximum memory bandwidth of 480Gbps and a maximum network bandwidth of 100Gbps. We apply variable network and memory load by running `stress-ng` [27] and `perftest` [42] on the first socket of our machines.

When the network load is maximum (bottom right figure), less than 12% of the calls to `Active` return immediately, irrespective of the memory load. For network loads of $30-80\%$ (other figures), the memory load progressively affects lease renewal. Maximum memory load causes expired leases when lease duration is shorter than 27µs. For most other configurations, a duration greater than 23µs suffices. For example, with 80% network and 50% memory load, lease renewal fails 0.0011% of the time, which corresponds to `Active` inducing latency every 300 out of 1.5 billion invocations. In other

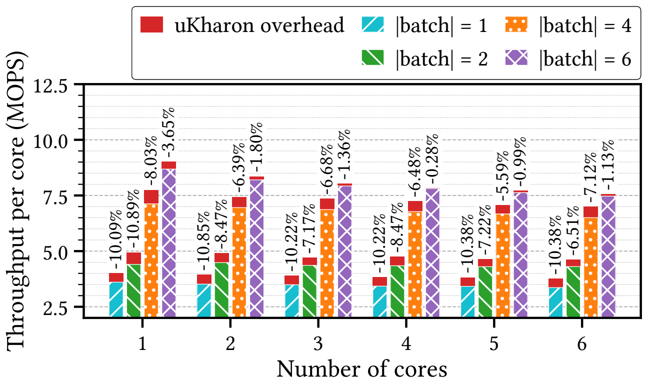

Figure 4: Impact of uKharon on HERD's throughput for different batch sizes and numbers of cores. Full bar shows the throughput w/o uKharon; labels show uKharon's overhead.

words, the 99.999th percentile of `Active`'s latency is 2µs.

We get similar (omitted) results when an application renews its leases through lease caches. In fact, RPC-based renewal requires at most 2µs longer leases (compared with reading from coordinators) to achieve the same percentages of timely lease renewal. We attribute this difference to RPC, which involves the CPU of both the application and the lease cache.

From this experiment we select the lease duration that we use for the rest of our evaluation. We pick the lease duration when renewing from coordinators (δ) to be $23\mu s$, and the lease duration when renewing from lease caches (Δ) to be $25\mu s$.

Throughput reduction. We use uKharon to make HERD dynamic. The original HERD assumes a static set of servers, each of which serves a shard of the key space. Clients are aware of this sharding and use the key of a request to determine the appropriate server. The lack of dynamicity affects HERD's flexibility in two ways. First, if a server fails, its shard becomes unavailable forever. Second, the system is unable to re-balance the load among the servers. Importantly, the use of a static set of servers ensures consistency of clients' requests: `GET`s return the value of the most recent `PUT`.

In our implementation, each server dedicates up to 6 cores to the KV-cache and each core is responsible for a part of the key space. Every core processes clients' requests and invokes the `Active` method before replying to avoid inconsistencies. If `Active` returns `true`, the core executes the request (if the key belongs to its shard) and replies to the client. Otherwise, the core rejects the request. Given that every core invokes `Active` in the critical path of serving requests, the latency of requests increases (by ∼38ns) and the throughput decreases.

Figure 4 shows the per-core throughput of a static deployment of HERD, along with the drop in performance caused by the integration of the `Active` method. The workload is 80% `GET`s and 20% `PUT`s with 32 byte-long values. We vary the number of cores from 1 up to 6 as well as the batch size (i.e., the number of clients' requests processed at once). Typically, static HERD issues a reply every 350ns. Without batching, having `Active` in the critical path raises the reply time to 388ns, an increase of 11%. Batching has a positive impact on `Active`'s overhead as a single call to the method is used to serve all the requests in a batch. Thus, for batches of 6 replies, `Active` effectively takes $38/6 = 6.3$ns per reply, an increase of just 1.8%. Finally, the overhead of `Active` does not increase with the number of cores, even though they invoke the method concurrently. This indicates good multicore scalability, which implies that a single uKharon Core instance per server is sufficient to serve all applications running on it.

Bandwidth overhead. uKharon Core reduces the bandwidth available to applications. Lease renewal requires 240 bytes when contacting 3 coordinators and 132 bytes when contacting a lease cache, which translates to (assuming renewal every 10µs) 192Mbps and 105Mbps, respectively. This bandwidth requirement accounts for $0.1 - 0.2\%$ of a 100Gbps link, thus the bandwidth of application nodes is marginally impacted. Failure detection has similar bandwidth requirement.

7.2 Failover Time

We study uKharon's failover time considering userspace and kernel failures. We do not further evaluate catastrophic failures, as 95% of the failover is for their 1ms-long detection, making microsecond-scale agreement and leases insignificant.

Table 2 summarizes the median failover (over 100 measurements) for various failure scenarios. We consider the failure of a single application node optionally combined with the failure of the coordinator leader or/and a lease cache. We emulate simultaneous failures by relying on RDMA Multicast. An auxiliary program executes alongside the program which we emulate the failure of. When the auxiliary program receives the multicast message, it uses `SIGKILL` to kill the targeted program. We assume the worst scenario, i.e., the failure of the application node results in global unavailability that is resolved only by a new (active) membership that excludes it.

In every entry of Table 2, we present the failover time when detecting the failure using the deadbeat mechanism (left) and the RDMA-based heartbeat mechanism (right). We now discuss the failover time when using the deadbeat, first considering the case when the lease caches are absent. For a single application failure, uKharon is able to failover in 50µs using the deadbeat. If the coordinator leader crashes at the same time as the application, the failover time increases by around 15µs. We attribute this increase to (1) the leader switch mechanism of the consensus engine (∼10µs) and (2) the imperfect synchronization of `SIGKILL` among the failed nodes (∼5µs). When lease caches are part of uKharon, the failover times for the same failure scenarios increase (as expected) by $20 - 25$µs, which is about the lease duration of the cache. Failure of a cache has no impact on the failover time (bottom entries of the first and third columns). This is because (1) the application node receives the broadcast failure

L exists?	**A**	**A + C**	**A + L**	**A + L + C**
No	50\96	64\114	-	-
Yes	74\108	96\138	75\113	101\139

Table 2: Failover time (in µs) for failures in **A**pp, **C**oordinator leader and **L**ease caches; using the deadbeat\heartbeat.

notification and switches lease cache before the membership changes and (2) the new membership is compatible with the previous one. The simultaneous failure of all three types of nodes has a downtime of 101 µs, instead of 96µs. Again, the failure of the cache does not affect the failover time, but with three nodes the imperfect synchronization of failures adds up. Finally, the same failures when using the RDMA-based heartbeat mechanism range from 96 to 139µs. This mechanism adds $\sim$ 45µs of failover compared to the deadbeat. The reason is that reading the same value twice upon failure takes 1.5 delays on expectation and READs are issued every 30µs.

7.3 uKharon-KV

Both uKharon-KV and HERD+Mu follow a primary-backup replication scheme. All requests are served by the primary, which replicates them to backups. Backups are only used for fault tolerance. All replicas (primary and backups) execute requests in the same order, but only the primary replies to clients. In the event of a failure of the primary, one of the backups becomes the new primary and continues serving clients' requests. All replicas execute all requests in the same total order, thus replicas are an exact copy of the failed primary. This means that when a replica becomes the new primary, it can respond to clients without breaking consistency.

One problem these systems have to deal with is multiple nodes trying to replicate clients' requests simultaneously. This happens when the primary fails and multiple nodes, believing they are the new primary, try to handle clients' requests. Mu avoids this problem by relying on RDMA permissions (see §5.3.2). On the other hand, uKharon-KV relies exclusively on the membership service to address it. Each membership determines a single primary. When the primary fails, a new membership is emitted that determines the new primary. Since only one membership is active at a time, no two replicas can believe to be the primary simultaneously.

The replication protocol of uKharon-KV works as follows: The primary P replicates all clients' requests to a single backup B by RMDA-writing them to a dedicated buffer on the latter. In parallel, P *speculatively* executes the requests. Upon completion of the RDMA WRITE, the primary checks that the membership in which P is the primary is still active. If that is the case, P replies to the client. Otherwise, P drops the request. Upon membership change, B waits for the new membership—in which it is the primary—to become active. Then, B scans the local buffer that was dedicated to P and applies all unprocessed requests in it. Only then B starts pro-

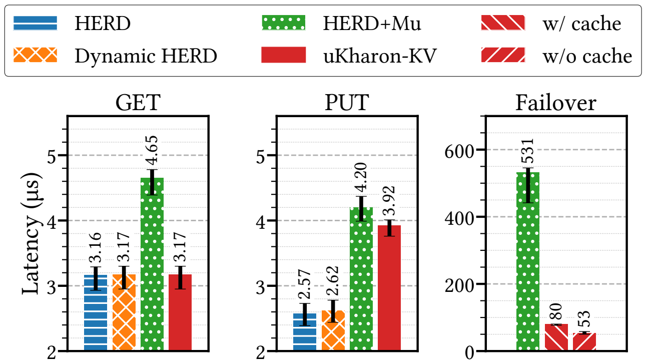

Figure 5: Latency comparison (left) of vanilla HERD, Dynamic HERD, HERD+Mu, uKharon-KV. Failover time comparison (right) of HERD+Mu and uKharon-KV. HERD+Mu uses 3-way replication; uKharon-KV uses its deadbeat. Bar height shows 95th %-ile latency; numerical label shows the 95th %-ile; error bars show the median and 99th %-ile.

cessing clients' requests. The client's failover time is the time interval between the client's last successful request to P and its first successful request to B (as the new primary).

If P's speculative execution turns out to be incorrect, its state may diverge from the one of the new primary B. uKharon-KV, however, does not follow the common practice of rolling back unsuccessful speculations, because our prototype adopts a simple design: when a node is removed from the membership, it is not allowed to re-enter the system. Thus, the state of the old primary P is no longer used when B takes over, hence skipping the rollback.

Replication latency. We compare the latency of HERD, HERD+Mu and uKharon-KV. For HERD, we deploy a single node. For HERD+Mu, we deploy three nodes, a primary and two backups, all of which execute an instance of HERD and Mu. For uKharon-KV, we deploy a primary and a backup, both running uKharon-KV, as well as three coordinators. For these experiments, a HERD client connects to the primary and issues `PUT` and `GET` requests. We measure the time it takes for a client to complete a request and compute the median, the 95th and the 99th percentiles over 10 million requests.

Figure 5 shows the end-to-end latency of vanilla HERD and of both replication approaches. In vanilla HERD, `PUT`s are more efficient than `GET`s by 23%, due to the way HERD handles the two types of requests. Briefly, `PUT`s rely mostly on RDMA WRITEs, which is the most efficient RDMA verb [25], while `GET`s rely mostly on RDMA SENDs. For reference, we also show the latency of Dynamic HERD, which uses uKharon's `Active` method in the critical path of executing clients' requests, as explained in section 7.1. We verify, once again, the efficiency of the `Active` method. At the 95th percentile, Dynamic HERD's requests are delayed by 10ns (for `GET`s) and 50ns (for `PUT`s), compared with vanilla HERD.

The two replicated solutions exhibit different costs.

HERD+Mu replicates all requests, regardless of whether they are PUTs or GETs, while uKharon-KV replicates only PUTs. HERD+Mu does not distinguish between PUTs and GETs, because in Mu the primary uses the result of replication (whether it is successful or not) to determine if it is still the primary or not. If Mu were to skip the replication of GETs, inconsistency would occur (see §2.1). On the other hand, uKharon-KV executes GETs locally, without replicating them, since the primary relies on the `Active` method to determine if its data is stale or not. Also, observe that uKharon-KV replicates PUTs approximately 300ns faster than Mu. This improvement is merely attributed to the speculative approach adopted by uKharon-KV. In HERD+Mu, the primary executes the request *after* it has been replicated to a majority. On the other hand, the primary in uKharon-KV executes the request in *parallel* to the replication to the backup. Thus, our solution hides the cost of executing the request, which is approximately 300ns, as shown by the difference of the two rightmost bars in the middle plot of Fig. 5. Regardless, uKharon-KV provides the same fault tolerance as Mu, even with one less replica: if a single replica crashes in either HERD+Mu or uKharon-KV, the system remains operational but cannot tolerate another failure. Fundamentally, both HERD+Mu and uKharon-KV assume a majority of correct nodes, the former among the replicas and the latter among the coordinators.

Failover. We compare the failover latency of uKharon-KV with HERD+Mu in the event of userspace failures. We run uKharon-KV in two configurations. In the first one, clients directly RDMA-read from coordinators to renew their lease. In the second one, clients go to lease caches. The third graph of Figure 5 shows that HERD+Mu has a 95th-percentile failover time of 531µs. This number is almost half of what Mu's authors report since we fine-tuned their failure detector for our own setup. At the same time, uKharon-KV without cache (resp. with) achieves a $10\times$ improvement (resp. $6.5\times$) at 53µs (resp. 80µs) of end-to-end failover time.

8 Related Work

Membership services in general. They are widely used in the data center. Distributed data processing apps (e.g., Kafka [30], MapReduce [10]), storage systems (e.g., Cassandra [31], HDFS [46]) and orchestration tools (e.g., Mesos [18]) rely on Zookeeper [20] for leader election, membership management, locks, watches, etc. uKharon focuses on membership management, yet it can be extended to support Zookeeper's features. Indeed, uKharon-KV (excluding the lack of durability) offers similar guarantees to the strongly consistent KV-store of Zookeeper, which comprises its basic building block. ZooKeeper's strongly consistent KV-store that forms its basis. For instance, locks can be implemented on top of uKharon-KV by extending its interface with CompareAndSwap. Watches, being an unreplicated pub/sub system, only require modifying uKharon-KV's primary. The important difference is that Zookeeper is not suitable for the microsecond scale and does not exploit RDMA.

Failure detection in the data center. A common approach to detect failures is to use end-to-end timeouts, which are hard to set. Falcon [35] proposes to use inside information in order to build faster and more accurate failure detectors by relying on hierarchies of specialized detectors. It maximizes accuracy by killing suspected processes. Albatross [34] is slightly more forgiving and isolates suspected processes so that they cannot affect the state of the system. Pigeon [33] provides fine-grained reports that end applications use to act accordingly. We embrace Falcon's philosophy and use RDMA-tailored failure detectors to operate at the microsecond scale.

Time-bound leases. Time-bound leases are widely used to implement consistent distributed applications at the price of some synchrony assumptions. They are often provided by a distributed coordination framework such as ZooKeeper [20] or etcd [14]. Leases are used for leader election [48], as well as for guarding memberships (e.g., in FaRM [12] and Hermes [26]). uKharon guards memberships with purely client-side leases. As a result, uKharon brings leases down to a few tens of microseconds and only assumes bounded clock drift instead of loosely synchronized clocks as in Hermes.

9 Conclusion

Continuous breakthroughs in data center fabrics have paved the way for microsecond applications. A key challenge for building tail-tolerant software at scale is for applications to react fast to events such as reconfigurations and failures. Yet, existing microsecond applications lack an equally fast membership service to provide microsecond dynamicity. This lack is counter-intuitive, as the vast ecosystem built around ZooKeeper showcases the usefulness of membership services. uKharon fills this gap by being the first membership service tailored to microsecond scale applications. To achieve this demanding target, uKharon relies on (1) a multi-level failure detector, (2) a consensus engine that takes advantage of RDMA CAS, as well as (3) leases, all of which have been carefully designed to operate in the microsecond envelope. We used uKharon to implement uKharon-KV, a replicated KV-cache which outperforms the state of the art in latency while improving its failover time by up to $10\times$.

Acknowledgments

We thank our NSDI '22 and ATC '22 anonymous reviewers as well as our shepherd, Abhinav Duggal, for their valuable comments. We would also like to thank our anonymous artifact evaluators for reviewing our implementation. This work was partly funded by Huawei Technologies.

References

[1] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, July 2018.

[2] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 599–616, 2020.

[3] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 3, USA, 2009. USENIX Association.

[4] Motti Beck and Michael Kagan. Performance evaluation of the RDMA over ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proceedings of the 3rd Workshop on Data Center-Converged and Virtual Ethernet Switching*, pages 9–15, 2011.

[5] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[6] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[7] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.

[8] Intel Corporation. Volume 3B: System Programming Guide, Part 2. In *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2016.

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

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Travis Downs. A benchmark for low-level CPU micro-architectural features. https://github.com/travisdowns/uarch-bench. Accessed 2022-05-25.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, April 2014.

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[14] Etcd. https://etcd.io. Accessed 2022-05-25.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[16] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[17] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, April 1985.

[18] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[19] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16, 2021.

[20] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference (ATC)*, June 2010.

[21] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, April 2018.

[23] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *USENIX*

Symposium on Networked Systems Design and Implementation (NSDI), pages 1–16, February 2019.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM Conference on SIGCOMM*, pages 295–306, August 2014.

[25] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, June 2016.

[26] Antonios Katsarakis, Vasilis Avrielatos, M R Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojević, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 201–217, March 2020.

[27] Colin King. stress-ng: A tool to load and stress a computer system. https://github.com/ColinIanKing/stress-ng. Accessed 2022-05-25.

[28] Patrick Knebel, Dan Berkram, Al Davis, Darel Emmot, Paolo Faraboschi, and Gary Gostin. Gen-z chipsetfor exascale fabrics. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2019.

[29] Marios Kogias and Edouard Bugnion. HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *European Conference on Computer Systems (EuroSys)*, April 2020.

[30] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[31] Avinash Lakshman and Prashant Malik. Cassandra—a decentralized structured storage system. In *International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, October 2009.

[32] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[33] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2013.

[34] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *European Conference on Computer Systems (EuroSys)*, April 2015.

[35] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Linux Kernel Developers. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt. Accessed 2022-05-25.

[38] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[39] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.

[40] Gregory F Pfister. An introduction to the InfiniBand architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[41] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on RDMA networks. In *Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118. ACM, June 2015.

[42] Linux RDMA. perftest: Infiniband verbs performance tests. https://github.com/linux-rdma/perftest. Accessed 2022-05-25.

[43] Red Hat, Inc. RHEL for Real Time Timestamping. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping. Accessed 2022-05-25.

[44] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

[47] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012.

[48] Anish Sukumaran and Vincent Gerard Nicotra. Lease based leader election system, May 29 2018. US Patent 9984140.

[49] Mellanox Technologies. RDMA aware networks programming user manual. rev 1.7. https://docs.nvidia.com/networking/spaces/viewspace.action?key=RDMAAwareProgrammingv17. Accessed 2022-05-25.

[50] Stephen Van Doren. HOTI 2019: Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE, 2019.

[51] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable paxos on RDMA. In *Symposium on Cloud Computing (SoCC)*, pages 94–107, September 2017.

[52] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation OSDI 18)*, pages 233–251, 2018.

[53] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, October 2015.

[54] Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. The synchronous data center. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 142–148, 2019.

[55] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

A One-Sided Paxos

A.1 Assumptions

In the next subsections, we consider the M&M model [1]. It allows processes to both pass messages and share memory. We assume that communication channels are lossless and have FIFO semantics, which is ensured by InfiniBand's Reliable Connections. The system has n processes $\Pi = \{p_1, \dots, p_n\}$ that can attain the roles of *proposer* or *acceptor*. There are p proposers and n acceptors. Up to $p-1$ proposers and $\lfloor \frac{n-1}{2} \rfloor$ acceptors may fail by crashing. As long as a process is alive, its memory is remotely accessible. When a process crashes, subsequent operations to its memory hang forever. We assume partial synchrony for consensus's liveness [17].

A.2 One-Sided RPC

In this section, we prove that the one-sided RPCs of Algorithm 1 are equivalent to two-sided RPCs when not obstructed. Moreover, we prove that when equivalence is violated (due to obstruction), one-sided RPCs have no side effects. We assume that both `compare` and `f` are deterministic.

Lemma A.1. *If `cas-rpc` does not abort, `rpc` and `cas-rpc` are equivalent.*

Proof. An execution of `rpc` solely depends on the value of `state` and the input value `x`. We denote such execution of `rpc` as $\langle state, x \rangle_{rpc}$. If an execution of `cas-rpc` does not abort, it solely depends on the value of `expected` fetched at line 2 and the input value `x`. We denote such execution of `cas-rpc` as $\langle expected, x \rangle_{cas-rpc}$.

We show that any execution $\langle s, x \rangle_{rpc}$ is equivalent to the execution $\langle s, x \rangle_{cas-rpc}$ in the sense that both `rpc` and `cas-rpc` will have the same `state` value and return the same projection at the end of their execution.

If an execution $\langle s_1, x \rangle_{rpc}$ makes the comparison at line 3 fail, then `state` is not modified and `proj(`s_1`)` is returned. In the execution $\langle s_1, x \rangle_{cas-rpc}$, the comparison at line 3 will also fail and `proj(`s_1`)` is also returned without modifying the remote state. In this case, both executions are equivalent.

If an execution $\langle s_2, x \rangle_{rpc}$ makes the comparison at line 3 succeed, then `state` is modified to `f(`s_2`, x)` and `proj(f(`s_2`, x))` is returned. In the execution $\langle s_2, x \rangle_{cas-rpc}$, the comparison at line 3 will also succeed. As the execution is assumed not to abort, the CAS will succeed. Thus the remote state will atomically be updated from s_2 to `f(`s_2`, x)` and `proj(f(`s_2`, x))` is also returned. In this case, both executions are also equivalent. □

Lemma A.2. *If `cas-rpc` aborts, it has no side effects.*

Proof. If `cas-rpc` aborts, the comparison at line 7 has failed. This implies that the CAS failed and thus that `state` is unaffected by the execution. □

From lemmas A.1 and A.2, `cas-rpc` exhibits all-or-nothing atomicity. We now prove that such a transformation is obstruction-free.

Lemma A.3. *If `cas-rpc` runs alone, it does not abort.*

Proof. Let's assume by contradiction that `cas-rpc` runs alone and aborts. For `cas-rpc` to abort, the comparison at line 7 must have failed. This implies that the CAS at line 6 failed due to `state` not matching `expected`. `state` must thus have been updated between lines 2 and 6. This implies a concurrent execution, hence a contradiction. □

A.3 Consensus and Abortable Consensus

In the consensus problem, processes *propose* individual values and eventually irrevocably *decide* on one of them. Formally, consensus has the following properties:

Termination Every correct process eventually decides once.

Uniform agreement If v and v' are decided on, then $v = v'$.

Validity If v is decided on, v is the input of some process.

We implement consensus by composing two abstractions:

- *Abortable consensus* [5], an abstraction weaker than consensus that is solvable in the asynchronous model,
- *Eventually perfect leader election* [7], the weakest failure detector required to solve consensus.

Abortable consensus is identical to consensus except for:

Termination Every correct process eventually decides once or aborts.

Decision If a single process proposes infinitely many times, it eventually decides.

A.4 One-Sided Abortable Consensus

Algorithm 3 appears in [5] and implements abortable consensus Algorithm 4 transforms algorithm 3 by replacing its RPCs with CAS-based RPCs. This transformation causes it to abort strictly more than the original algorithm. To see why, consider the following execution: Let proposers P_1 and P_2 concurrently initiate the Prepare phase with respective proposals 1 and 2. Both fetch the remote state and get $\langle 0,0,\perp\rangle$. Then, P_1 succeeds in writing its proposal to acceptor A_1. Later on, the CAS of P_2 fails at A_1 as the value is now $\langle 1,0,\perp\rangle$ instead of the expected $\langle 0,0,\perp\rangle$. Thus, P_2 aborts even if it had a larger proposal number than P_1. The more relaxed comparison in the original algorithm would not have caused P_2 to abort.

Lemma A.4. *Algorithm 4 preserves Decision.*

Proof. If a single process proposes infinitely many times, it will eventually run the one-sided RPCs obstruction-free. By Lemma A.3, this guarantees that the one-sided RPCs will eventually terminate without aborting. In such case, Lemma A.1 guarantees the execution to be equivalent to one of the original algorithm. Thus, the transformation preserves the decision property of Algorithm 3. □

Algorithm 3: Paxos's Abortable Core

```
Proposers execute:
decided = False
proposal = id
proposed_value = ⊥

def propose(value):
   proposed_value = value
   prepare()
   accept()

def prepare():
   proposal = proposal + |Π|
   broadcast ⟨Prepare | proposal⟩
   wait for a majority of ⟨Prepared | ack, ap, av⟩
   adopt av with highest ap as proposed_value
   if any not ack: abort

def accept():
   broadcast ⟨Accept | proposal, proposed_value⟩
   wait for a majority of ⟨Accepted | mp⟩
   if any mp > proposal: abort
   trigger once ⟨Decide | proposed_value⟩

Acceptors execute:
min_proposal = 0
accepted_proposal = 0
accepted_value = ⊥

upon ⟨Prepare | proposal⟩:
   if proposal > min_proposal: min_proposal = n
   reply ⟨Prepared | min_proposal == n, accepted_proposal,
       ↪ accepted_value⟩

upon ⟨Accept | proposal, value⟩:
   if proposal ≥ min_proposal:
      accepted_proposal = min_proposal = n
      accepted_value = value
   reply ⟨Accepted | min_proposal⟩
```

Lemma A.5. *Algorithm 4 preserves Termination.*

Proof. Assuming a majority of correct acceptors, CASes will eventually complete at a majority. Due to the absence of loops or blocking operations inside `prepare`, `accept`, `cas_prepare` and `cas_accept` in algorithm 4 (apart from waiting for the completion of CASes at a majority), a proposer that invokes `propose` will either abort or decide. □

Algorithms 3 and 4 differ only in some executions where the transformed algorithm aborts whereas the original does not. Nevertheless, aborting does not violate safety, as we show next.

Lemma A.6. *Algorithm 4 preserves the safety properties.*

Proof. Assume, by contradiction, that adding superfluous abortions in Algorithm 3 violates safety. Consider an execution E_1, where processes $\{P_1, ..., P_n\}$ deviate from the algorithm and abort at times $\{t_1, ..., t_n\}$ after which the global state is $\{S_1, ..., S_n\}$ and safety is violated. Also, consider another execution E_2, where processes $\{P_1, ..., P_n\}$ crash at times $\{t_1, ..., t_n\}$ after which the global state is $\{S_1, ..., S_n\}$. In execution E_1, safety is violated. On the other hand, execution E_2 preserves safety, since Algorithm 3 tolerates arbitrarily many proposer crashing. The two executions, however, are

Algorithm 4: One-Sided Abortable Consensus

```
Acceptors execute:
state = { min_proposal: 0, accepted_proposal: 0,
    ↪ accepted_value: ⊥}

Proposers execute:
proposal = id
proposed_value = ⊥

def propose(value):
   proposed_value = value
   prepare()
   accept()

def prepare():
   proposal = proposal + |Π|
   async cas_prepare(p) for p in Acceptors
   wait for a majority to return ⟨ack, ap, av⟩
   if any not ack: abort
   adopt av with highest ap as proposed_value

def accept():
   async cas_accept(p) for p in Acceptors
   wait for a majority to return mp
   if any mp > proposal: abort
   trigger once ⟨Decide | proposed_value⟩

def cas_prepare(p):
   expected = fetch_state(p)
   if not proposal > expected.min_proposal:
      return ⟨False, expected.accepted_proposal, expected.
         ↪ accepted_value⟩
   move_to = expected
   move_to.min_proposal = proposal
   read = state_p.cas(expected, move_to)
   if read == expected:
      return ⟨True, expected.accepted_proposal, expected.
         ↪ accepted_value⟩
   abort

def cas_accept(p):
   expected = fetch_state(p)
   if not proposal ≥ expected.min_proposal:
      return expected.min_proposal
   move_to = expected
   move_to.min_proposal = proposal
   move_to.accepted_proposal = proposal
   move_to.accepted_value = proposed_value
   read = state_p.cas(expected, move_to)
   if read == expected:
      return expected.min_proposal
   abort
```

indistinguishable, hence a contradiction. Thus, Algorithm 4 preserves safety regardless of how often it aborts. □

Theorem A.7. *Algorithm 4 implements abortable consensus.*

Proof. The result follows directly by composing lemmas A.4, A.5 and A.6. □

A.5 Streamlined One-Sided Algorithm

In this section, we make Algorithm 4 efficient in order to increase its practicality.

First, it is not required to fetch the remote state at the start of each RPC. As it is safe to have stale `expected` states, it is safe to use states deduced from previous CASes. Predicted states can thus be initialized to $\langle 0, 0, \bot \rangle$ and updated each time a CAS completes (either succeeding or not). Moreover, wrongly predicting states can only result in superfluous aborts which have been proven to be safe by Lemma A.6. Thus, it is safe to optimistically assume that onflight CASes will succeed. Second, in the Prepare phase, the `proposal` variable can be increased upfront to value higher than any predicted remote `min_proposal` to reduce predictable abortions.

Algorithm 5: Streamlined One-Sided Abortable Consensus

```
 1  Acceptors execute:
 2  state = { min_proposal: 0, accepted_proposal: 0,
        ↪ accepted_value: ⊥}

 4  Proposers execute:
 5  predicted[] = { 0, 0, ⊥}
 6  proposal = id
 7  proposed_value = ⊥

 9  def propose(value):
10     proposed_value = value
11     prepare()
12     accept()

14  def prepare():
15     while any predicted[.].min_proposal ≥ proposal:
16        proposal = proposal + |Π|
17     for p in Acceptors:
18        move_to[p] = {min_proposal: proposal, ..predicted[p]}
19        reads[p] = async state_p.cas(predicted[p], move_to[p])
20     wait until majority of states are read
21     for p in Acceptors:
22        if reads[p] ∈ {predicted[p], ⊥}:
23           predicted[p] = move_to[p]
24        else:
25           predicted[p] = reads[p]
26     if any CAS failed: abort
27     adopt proposed_value from predicted accepted_values with
         ↪  highest accepted_proposal if any

29  def accept():
30     reads = ⊥^|Acceptors|
31     move_to = (proposal, proposal, proposed_value)
32     for p in Acceptors:
33        reads[p] = async state_p.cas(predicted[p], move_to)
34     wait until majority of states are read
35     if any CAS failed:
36        for p in Acceptors:
37           if reads[p] ∈ {predicted[p], ⊥}:
38              predicted[p] = move_to
39           else:
40              predicted[p] = reads[p]
41        abort
42     trigger once ⟨Decide | proposed_value⟩
```

With the aforementioned optimisations, Algorithm 4 is transformed into Algorithm 5. Notably, the liveness of the resulting algorithm is preserved: Let's assume that a *single* proposer runs infinitely many times. Eventually, it will run obstruction-free. In the worst case, each time it will abort at line 26 or 41 because of a single wrong guess and update its prediction. The optimistic update of expected states at lines 23 and 38 and the FIFO semantics of communication links provide that, once a remote state is correctly guessed, any later CAS will succeed. Thus, after at most n runs, all CASes will succeed and the proposer will decide.

A.6 Overcoming Limited CAS Size

As explained in Section 5.3.1, the RDMA hardware limits the size of CASes. Thus, proposal fields will overflow after 2^{16} attempts. In such an unlikely scenario, our consensus engine falls back to traditional RPC: Once the RDMA-exposed `min_proposal` of an acceptor reaches $2^{16} - |\Pi|$, proposers switch to RPC to communicate with this specific acceptor. Acceptors check `state` and, if it is above the threshold, initiate the standard RPC version of Paxos with the `min_proposal`, `accepted_proposal` and `accepted_value` variables initialized to match `state`.

B Active Method Correctness

In this section, we provide a formal definition and a proof of correctness of the `Active` method described in Section 6.

B.1 Formal Definition

`Active(`*Membership*`)`$\rightarrow$*bool* has the following properties:

Monotonicity If `Active(M')` returns `true` at any process, future calls `Active(M)` with $M < M'$ will return `false`.

Convergence If M is the last membership to be decided (if any), invoking `Active(M)` will eventually return `true` at all correct processes.

Definition 1. *If* `Active(M)` *returns* `true`*, then M is considered active at the linearization point of the call.*

Definition 2. *If M is active at times t and t', then it is considered active in the interval $[t, t']$.*

From these simple properties and definitions, it follows that no two active memberships can overlap.

Theorem B.1. *Only one membership can be active at a time.*

Proof. Assume by contradiction that M and M' ($M < M'$) are simultaneously active. By definition, `Active(M)` must have returned `true` after `Active(M')` returned `true`. This breaks Monotonicity, hence a contradiction. □

B.2 Non-Leased Active Membership

We prove the correctness of uKharon's implementation of `Active`. We assume no gaps in the sequence of decided memberships. This is enforced by coordinators by not proposing the $(k+1)$-th membership until the k-th is decided.

Lemma B.2. *Algorithm 6 ensures Monotonicity.*

Proof. `Active` can only be called on decided memberships. Let M and M' be two decided memberships with $M < M'$. If `Active(M')` returned `true`, by the no-gap assumption, all

Algorithm 6: Active built on top of the consensus engine

```
1 def Active(M) → bool:
2    reads = ⊥^{|Acceptors|}
3    for p in Acceptors:
4       reads[p] = async paxos[M.id + 1].slot_p.read()
5    wait until majority of slots are read
6    if all slots are not accepted:
7       return true
8    propose_membership(M.id + 1, first accepted value)
9    return false
```

memberships between M and M' have been decided. Because M's successor has been decided, a majority of acceptors' slots `M.id` + 1 have been written. Thus, `Active(M)` will read at least one non-empty slot and return `false`. □

Lemma B.3. *Algorithm 6 ensures Convergence.*

Proof. Assume by contradiction that M is the last decided membership and `Active(M)` never returns `true` at some correct process. Thus, this process executes line 8, which means that it proposes a new membership. Given that the process is correct, some membership with id `M.id` + 1 will eventually be decided. Therefore, M is not the last membership, hence a contradiction. □

Theorem B.4. *Algorithm 6 implements* `Active`.

Proof. Follows directly from Lemmas B.2 and B.3. □

B.3 Leased Active Membership

Algorithm 2 reduces communication by leasing the output of Algorithm 6. We prove that it preserves `Active`'s properties.

Lemma B.5. *Algorithm 2 preserves Monotonicity.*

Proof. Let e be an execution of `Active(M)` that returned `true`. e either returned at line 9 or at line 12 with `t` > `t`$_{start}$. We denote the former case *leased*(M) and the latter *checked*(M). Assume by contradiction that `Active(M')` returned `true` in an execution e_1 and then `Active(M)` returned `true` in an execution e_2 with $M < M'$. Either:

- *leased*(M): In e_2, `majority_active(M)` returned `true` at most δ before `Active(M)` returned `true`. In e_1, lines 5–7 ensure that M' was decided at least δ before `Active(M')` returned `true`. Thus, `majority_active(M)` returned `true` after M' was decided. However, because M' has been decided, a majority of acceptors' slots `M'.id` = `M.id` + 1 must have been written. Thus, `majority_active(M)` should have read at least one non-empty slot and returned `false`. Hence, a contradiction.

- *checked*(M): `majority_active(M)` returned `true` after `majority_active(M')` returned `true`. This breaks `majority_active`'s Monotonicity, hence a contradiction.

□

Lemma B.6. *Algorithm 2 preserves Convergence.*

Proof. Assume that M is the last membership to be decided. Thus, `majority_active(M)` will eventually always return `true`. At most δ after `Active(M)` returns for the first time, $\mathtt{t}_{start}$ will be in the past and `leased_membership` set to M. Thus, eventually, the `else` branch at line 8 will always be visited and either return *true* via line 9 or 12. □

Theorem B.7. *Algorithm 2 implements* `Active`*.*

Proof. Follows directly from Lemmas B.5 and B.6. □

C Clocks

uKharon relies on hardware timestamps to check if a membership is `Active`. When using modern Intel processors, Linux has three available clocksources: `tsc`, `hpet` and `acpi_pm`. The `tsc` clocksource is the most efficient and requires 20-25ns to take a timestamp [11].

Architectural considerations. The `tsc` clocksource uses Intel's TSC hardware to measure time accurately. TSC stores the number of cycles executed by the CPU after the latest reset. Traditionally, TSC is considered an unreliable way to take timestamps. The reason is that Intel processors have variable clock speed, thus the number of cycles does not correspond to wallclock time. However, modern Intel processors have three features [8]: *Constant TSC*, *Nonstop TSC* and *Invariant TSC* which solve this problem. The combination of these features results in a TSC that is incremented at a constant rate regardless of the power state of the processor. As a result, it is safe to use this counter for efficient timestamping.

TSC synchrony. In Intel processors, every core has its own TSC. All processors in the same socket start the TSC hardware using the same RESET signal, thus the absolute values of the TSC across cores of the same socket match. This means that one can compare safely the values of TSC across different cores, assuming that all TSCs run at the same frequency. Because this assumption does not always hold, Linux determines the base frequency of every core during boot and uses this frequency to convert clock cycles to wallclock time. To accomplish it, Linux uses the more accurate (and more expensive) `hpet`.

uKharon takes further care to deal with TSC synchrony. More precisely, it checks for the synchronization of TSC between cores using a ping-pong test. In this test, core A takes a timestamp t_1 and signals core B to do the same. Core A signals core B by writing to a lock-free Single-Producer Single-Consumer (SPSC) queue that is polled by B. When B receives the signal it also takes a timestamp t_2 and sends it back to A (using another SPSC queue). Upon reception of the timestamp from B, core A takes the last timestamp t_3. In our test we confirm that always $t_1 < t_2 < t_3$. Additionally, in our hardware, the minimum difference between t_1 and t_2 is $\varepsilon = \min(t_2 - t_1)$ is 64ns. uKharon takes ε into consideration by incorporating into the leases as follows: Suppose a lease is valid for a duration of δ starting at time t. uKharon considers that the lease starts at time $t + \varepsilon$ and has a duration of $t + d - 2\varepsilon$.

Inter-machine clock drift. In order to ensure that active memberships do not overlap, uKharon assumes that clock drift is bounded, i.e., that time passes approximately at the same speed on different machines. This assumption is necessary to enable client-side leases. It guarantees that after a lease duration period, leases across all clients will have to be renewed. Our system is built to tolerate clock drift, as long as this drift is bounded. We experimentally determine an upper bound for the clock drift with a simple test. In this test, machine A takes a timestamp t_1 and pings machine B to wait for 1 minute before replying back to it. Upon reception of B's response, A takes another timestamp t_2. It then computes $t_2 - t_1$ and compares it to the expected 1 minute measured by B (after removing the communication delay). We repeat this test several times and determine that the clock drift between machines differs by at most 0.001%. uKharon incorporates inter-machine clock drift by waiting $1.01 \times \delta$ upon membership discovery, ensuring that when leases become active on a new membership, everyone's leases on the previous membership will have expired.

D Artifact

Abstract

The evaluated artifact is provided as a git repository and contains the source code of uKharon, build instructions and deployment scripts to run the experiments presented in this paper.

Scope

The artifact contains code and steps to reproduce results obtained in Figure 3, Figure 4, Figure 5 and Table 2.

Contents

The artifact contains the source code of uKharon, including the custom kernel modules. It also contains the patches to create the custom Linux kernel, as well as the patches required for HERD [24] and Mu [2]. The artifact describes how to build everything presented in the paper, including the custom Linux kernel and the solutions we compare against. It also describes how to deploy the built binaries.

Hosting

The artifact source code for uKharon is available at https://github.com/LPD-EPFL/ukharon. All the necessary instructions are provided in the README.md file.

Requirements

Building uKharon requires an x86-64 system set-up with Ubuntu 20.04 LTS. Executing uKharon requires 8 machines equipped with Ubuntu 20.04 LTS, RDMA over InfiniBand, ability to install a custom kernel and custom kernel modules, as well as ability to configure and use InfiniBand multicast groups.