uBFT: Microsecond-Scale BFT using Disaggregated Memory

uBFT: Microsecond-Scale BFT using Disaggregated Memory

Marcos K. Aguilera
maguilera@vmware.com
VMware Research
United States

Naama Ben-David
bendavidn@vmware.com
VMware Research
United States

Rachid Guerraoui
rachid.guerraoui@epfl.ch
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Antoine Murat
antoine.murat@epfl.ch
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Athanasios Xygkis
athanasios.xygkis@epfl.ch
École Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Igor Zablotchi
igorz@mit.edu
MIT
United States

ABSTRACT

We propose uBFT, the first State Machine Replication (SMR) system to achieve microsecond-scale latency in data centers, while using only $2f+1$ replicas to tolerate f Byzantine failures. The Byzantine Fault Tolerance (BFT) provided by uBFT is essential as pure crashes appear to be a mere illusion with real-life systems reportedly failing in many unexpected ways. uBFT relies on a small non-tailored trusted computing base—disaggregated memory—and consumes a practically bounded amount of memory. uBFT is based on a novel abstraction called Consistent Tail Broadcast, which we use to prevent equivocation while bounding memory. We implement uBFT using RDMA-based disaggregated memory and obtain an end-to-end latency of as little as 10 µs. This is at least 50× faster than MinBFT, a state-of-the-art $2f+1$ BFT SMR based on Intel's SGX. We use uBFT to replicate two KV-stores (Memcached and Redis), as well as a financial order matching engine (Liquibook). These applications have low latency (up to 20 µs) and become Byzantine tolerant with as little as 10 µs more. The price for uBFT is a small amount of reliable disaggregated memory (less than 1 MiB), which in our prototype consists of a small number of memory servers connected through RDMA and replicated for fault tolerance.

CCS CONCEPTS

• **Computer systems organization → Reliability**; **Availability**.

KEYWORDS

Byzantine fault tolerance, microsecond scale, replication, disaggregated memory, fast path, finite memory, RDMA

ACM Reference Format:
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575732

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575732

1 INTRODUCTION

Data center applications such as social networks, web search, e-commerce, and banking increasingly need both microsecond-scale performance [14, 20, 47, 81] and strong fault tolerance, in order to deliver on their promise of being the backbone of today's online services. Indeed, strong fault tolerance is required by real-life systems which reportedly fail in many unexpected ways. Apart from simple crashes, failures in distributed systems range from software/configuration bugs [57, 75], to hardware failures [66, 67, 72], to hardly detectable hardware bugs [33, 42, 44, 45] and up to malicious activity [10, 83]. Traditionally, protecting against the plethora of different failures required slow and expensive Byzantine Fault Tolerant (BFT) protocols.

The standard way to achieve fault tolerance is state machine replication (SMR). A BFT protocol implementing SMR incurs milliseconds of latency [9, 19], requires a large number of replicas ($3f+1$ to tolerate f failures) [24, 68, 92], consumes unbounded memory [88], and/or relies on a large trusted computing base [16, 51, 88]. These reasons might explain why BFT has had no adoption in data centers.

In this paper, we propose uBFT, the first BFT SMR system that simultaneously offers four key features: (1) microsecond-scale latency, (2) few replicas ($2f+1$), (3) practically bounded memory, and (4) a small non-tailored trusted computing base. In the common case, uBFT leverages unanimity to replicate requests in as little as 10 µs end-to-end without invoking the trusted computing base or expensive cryptographic primitives. In the slow path—when there are failures or slowness in the network—uBFT uses a novel protocol that combines digital signatures with judicious use of a trusted computing base. The trusted computing base in uBFT is non-tailored and small: rather than trusted enclaves with arbitrary logic such as Intel's SGX [28] or trusted hypervisors [94]—which have large attack surfaces due to their complexity [29, 36]—uBFT relies solely on disaggregated memory, a technology increasingly present in data centers due to the availability of RDMA [85] today and CXL [30] in a few years. The key mechanism from disaggregated memory we leverage in uBFT are single-writer regions (regions of memory that can be written by one designated host and can be read by others), implemented in hardware through access permissions.

Providing the above four features is challenging for BFT protocols. To get microsecond-scale latency, BFT protocols need to avoid expensive public-key cryptography and reduce communication rounds in the common path—and doing so has typically required increasing rather than decreasing the number of replicas [1, 53, 64]. Meanwhile, decreasing the number of replicas has usually required unbounded memory, sophisticated, or tailored trusted computing bases such as append-only-memory [25], SGX [16], TrInc [59], or reliable hypervisors [94]. Limiting the amount of memory is a significant challenge in the design of uBFT, as the standard technique to handle Byzantine behavior in systems with $2f+1$ replicas requires storing all messages received, leading to long message histories [6, 88], which consume unbounded memory. Finally, not tailoring the trusted computing base to our needs requires designing around existing technologies—in our case disaggregated memory—rather than custom hardware.

To respond to these challenges, uBFT introduces a new abstraction called Consistent Tail Broadcast (CTBcast) that we use to prevent equivocation [62], while requiring a practically bounded amount of memory.[1] Equivocation—a major source of problems in a system with Byzantine failures [25]—occurs when a faulty process incorrectly sends different information to different processes, which may cause the state of replicas to diverge. CTBcast prevents equivocation for all messages, but only ensures the delivery of the last t broadcast messages, where t is a parameter that trades memory for latency (we explain how to set it in Section 7).

The price for uBFT is a small amount (less than 1 MiB) of reliable disaggregated memory. uBFT is designed modularly to work with a generic such component; our current prototype implements this component using RDMA and a set of memory nodes that themselves are replicated for fault tolerance. These nodes add to the total number of replicas, but these replicas are tiny and simple: they do not store the state of the application, just a few in-flight coordination messages. Moreover, their functionality is application independent, so they can be shared among many applications, amortizing their cost. The memory nodes that provide the disaggregated memory constitute the trusted computing base in our prototype and are assumed to fail only by crashing. This shrinks the vulnerability of the system compared to currently deployed crash-tolerant SMR systems, in which *all* components can fail only by crashing, effectively making the trusted computing base be the entire data center.

We evaluate uBFT against two state-of-the-art systems. First, we compare it against Mu, the fastest SMR system to our knowledge, but that tolerates only crash failures. Compared to Mu, uBFT increases the end-to-end latency by only 2×, while tolerating Byzantine failures. Second, we compare uBFT against MinBFT, a state-of-the-art $2f+1$ BFT SMR system, and showcase that our system has more than 50× and 2× better latency when operating in its fast and slow path, respectively. We also use uBFT to replicate two low-latency KV-stores (Memcached [48] and Redis [80]), and a financial order matching engine (Liquibook [73]). All these applications have request latencies of less than 20 µs when unreplicated and become Byzantine-resilient with as little as 10 µs more.

[1] The required memory is logarithmic in the number of operations. Everywhere we use a bounded number of bits except for sequence numbers.

In summary, our main contributions are the following:

- The design of uBFT, a BFT system for state machine replication with microsecond-scale latency in the common case, using only $2f+1$ replicas, practically bounded memory, and a small trusted computing base (disaggregated memory).
- A new abstraction against equivocation, Consistent Tail Broadcast (CTBcast), and a protocol for CTBcast that uses a small amount of disaggregated memory and has a signature-less fast path.
- An open-source implementation of uBFT, CTBcast, and reliable shared disaggregated memory using RDMA, available at https://github.com/LPD-EPFL/ubft.
- A thorough evaluation of the performance of uBFT and its applications.

2 BACKGROUND

2.1 State Machine Replication

State Machine Replication (SMR) is an approach to replicate a service (e.g., a key-value store) across the memory of multiple machines called *replicas* to tolerate failures. With SMR, replicas receive requests in the same order, and thus remain in sync. If durability of SMR is desired, we rely on persistent rather than volatile memory.

The heart of an SMR system is consensus [22], which ensures total order in the sense that replicas agree on the same sequence of client requests. Consensus protocols are often leader-based [12, 24, 43, 46, 63, 70]: a replica designated as the *leader* orders the client requests, and forwards them to the other *follower* replicas to ensure agreement. Every replica feeds the client requests to its local copy of the service, and forwards the service's output to the clients.

Consensus and SMR differ in their memory consumption. Consensus must ensure agreement among participants indefinitely, so decided requests must be retained forever if a participant is unresponsive, thereby requiring unbounded memory. In contrast, SMR need not store all requests, as it cares only about replicating the application state, which can be finite even if there are infinitely many requests. Thus SMR systems need to adapt consensus to use finite memory [82].

2.2 Non-Equivocation

Byzantine processes can equivocate, i.e., they can maliciously say different things to different processes. In SMR, specifically, a Byzantine leader may propose different values to try to cause replicas to diverge, justifying why SMR protocols must ensure non-equivocation.

Under the Byzantine asynchronous model, $3f+1$ replicas are needed to prevent equivocation [54]. However, if equivocation is prevented and transferable authentication is available, Byzantine SMR requires only $2f+1$ replicas [26], the same number as in the crash-stop case. With transferable authentication, a process that verifies a proof about the origin of a message can transfer the proof to other processes and be assured they can also verify it. For example, digital signatures provide transferable authentication, while arrays of Message Authentication Codes (MACs) do not [8].

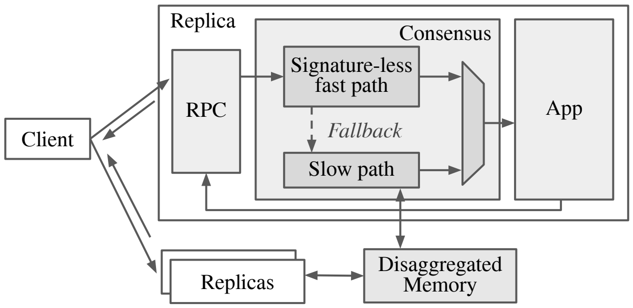

Figure 1: Overview of uBFT's architecture.

Preventing equivocation using up to $2f+1$ replicas requires a compromise [26], i.e., a hybrid model where part of the system—called the *trusted computing base*—fails only by crashing. Ideally, this base is as small as possible, since a small and simple base is less likely to be susceptible to failures (e.g., vulnerabilities, bugs).

2.3 Disaggregated Memory

Disaggregated memory is an emerging data center technology that separates compute from memory, by providing a shared memory pool that compute nodes access over a network. The memory pool has limited compute capabilities, which it uses for management tasks such as connection handling. Disaggregated memory improves memory utilization, separates the scaling of compute and memory, and achieves better availability due to the separation of fault domains [89].

Disaggregated memory can be provided by different technologies. The emerging CXL standard will support disaggregated memory in the future [39], while today disaggregated memory is available via Remote Direct Memory Access (RDMA) [85] on InfiniBand [49] or RoCE [15]. RDMA is a networking technology that allows a process to read or write the memory of another machine without involving the CPU of the latter. Combined with kernel-bypass, RDMA enables sub-microsecond communication and stringent tail latency. Access rights to RDMA-exposed memory can be set individually for each accessor.

2.4 Model

We consider a system with $2f+1$ compute nodes and single-writer multiple-reader disaggregated memory. Up to f compute nodes are Byzantine and may thus fail arbitrarily. We assume network connections are authenticated and tamper-proof (processes know who they get messages from and messages cannot be altered) and eventually available (network partitions are intermittent). We also assume that the disaggregated memory is trusted: it may fail only by crashing. The disaggregated memory is divided into chunks, where each chunk is readable by all compute nodes and writable by a designated compute node. We assume the existence of public-key cryptography: processes can sign messages using their private key and verify unforgeable signatures using the pre-published public keys of all processes. We further assume eventual synchrony: network and processing delays are unbounded until an unknown *Global Stabilization Time* (GST) after which delays are bounded by a known δ. Lastly, our system assumes bounded clock drift for safety, i.e., the clocks of correct processes drift from each other with a bounded rate. These assumptions are common for distributed systems in data centers [7, 24, 34, 41, 58, 90].

In our prototype, we do *not* assume that we are given a reliable disaggregated memory [56, 93], but rather show how to implement a reliable disaggregated memory using RDMA.[2] To do so, we assume $2f_m+1$ memory nodes out of which f_m can fail. Memory nodes are part of the trusted computing base: they are not Byzantine and may fail by crashing only. Memory nodes are simple: they just provide read and write functionality with access control. Their size and functionality do not depend on the application being replicated, and they can be shared among many applications.

3 DESIGN

3.1 Overview

uBFT follows the design of PBFT [24], a seminal paper that describes how to build practical BFT SMR systems. Figure 1 depicts the architecture of uBFT. On the left, a client sends requests to replicas on the right and waits for responses from a majority of them. The replicas go through two stages. First, they totally order client requests using a leader-based BFT consensus protocol. Second, they execute the ordered requests on their local instance of the replicated application before forwarding the outcome of the execution to the client. To achieve microsecond-scale latency, the consensus engine uses a fast/slow path approach. As long as the system is synchronous and all replicas collaborate, the fast path orders requests without signatures. If the fast path does not make progress, uBFT's consensus switches to the slow path, which makes progress with a mere majority of processes using signatures and disaggregated memory.

uBFT significantly differs from PBFT in the way it prevents equivocation. PBFT, being a $3f+1$ fault tolerant protocol, relies on intersecting quorums to ensure that malicious replicas do not make the state of honest replicas diverge. By contrast, uBFT operates with $2f+1$ replicas, and therefore cannot rely on the same mechanism as PBFT; instead, it relies on trusted disaggregated memory, which is encapsulated within a new primitive called *Consistent Tail Broadcast* (CTBcast).

CTBcast is a variant of Consistent Broadcast [6]. Consistent Broadcast prevents equivocation by ordering all messages broadcast by a given process. With it, a Byzantine leader is constrained from sending different request orderings to different followers. Our *tail* variant is a relaxation that requires correct processes to deliver only the last t messages sent by a correct broadcaster, while preserving non-equivocation for all messages. This relaxation is essential to practically bound the memory use. Importantly, the implementation of CTBcast has a signatureless fast-path to achieve the latency requirements of uBFT.

Figure 2 depicts uBFT's consensus component with its fast/slow path design. After receiving a request from RPC, the leader proposes its ordering via a round of CTBcast. The rest of consensus tries to turn this ordering into a globally accepted one (i.e., stable across

[2] uBFT's clean encapsulation of disaggregated memory allows future replacement of its RDMA implementation with CXL-powered memory.

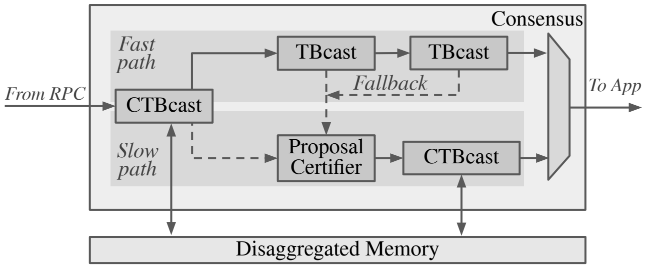

Figure 2: Overview of uBFT's consensus engine.

leaders). Depending on the synchrony of the system and the number of faulty replicas, this round of CTBcast might execute the fast or slow path. In the former case, consensus continues with its fast path and executes two rounds of Tail Broadcast (TBcast), a form of best-effort broadcast designed for finite memory (§4.1). Importantly, none of these three broadcasts involve signatures nor disaggregated memory. If liveness is lost during the fast path of consensus, uBFT activates the slow path, shown by the dashed arrows, which executes a certification round and another round of CTBcast. The slow path is also executed if the fast path of the initial CTBcast fails. Differently from the three rounds of the fast path, the three rounds of the slow path all require signatures, and CTBcast invocations additionally require disaggregated memory.

3.2 Challenges

uBFT addresses the following challenges:

2f+1 Replicas and Finite Memory. Previous theoretical work [2] proposed to prevent equivocation with fewer than $3f+1$ replicas by building Consistent Broadcast on top of shared registers. However, this abstraction requires replicas to use infinite memory in order to store and deliver *all* broadcast messages, which is not implementable in practice. We work around this memory issue by designing CTBcast, a weaker form of Consistent Broadcast where replicas are allowed to skip the delivery of old messages in order to favor the delivery of newer ones (§4).

SMR with CTBcast. The reliance of uBFT on CTBcast brings additional complexity to its consensus algorithm, notably on preventing equivocation *across* messages. Typically, protocols rely on the entire history of messages to prevent equivocation. Yet, CTBcast only guarantees the delivery of the tail, which may lead correct replicas to have gaps in their delivery history. uBFT works around this limitation via *CTBcast summaries*, which allow a replica to make progress in spite of gaps (§5).

Microsecond-Scale Operation. Systems that operate at the microsecond scale should avoid signatures on their critical path. Yet, Aguilera *et al.* [6] show that Consistent Broadcast cannot completely remove signatures. Moreover, recycling memory requires the generation of proofs which also involve signatures. uBFT addresses this challenge by avoiding expensive cryptography in the fast path of CTBcast and relegating the few bookkeeping signatures to a background task (§5.4).

Resilient Disaggregated Memory. uBFT relies on RDMA to implement disaggregated memory. However, raw memory exposed over RDMA is not enough to implement our SMR protocol. Indeed, RDMA-exposed memory does not tolerate failures, and data accesses can be inconsistent, since RDMA provides only 8-byte atomicity. uBFT addresses these limitations of RDMA using efficient, yet Byzantine fault tolerant, algorithms (§6).

4 CONSISTENT TAIL BROADCAST

Consistent Tail Broadcast (CTBcast) is a novel variant of Consistent Broadcast (CBcast) that uBFT uses to prevent equivocation. Briefly, CTBcast resembles CBcast, except that it allows processes not to deliver outdated messages. In this way, CTBcast avoids maintaining the full history of messages, to bound memory use.

4.1 Definition

CTBcast is defined in terms of two primitives, *broadcast(k,m)* and *deliver(k,m,p)*, where k is a numeric identifier, m is a message, and p is a process. When p invokes *broadcast(k,m)*, we say that p broadcasts (k, m), i.e., it broadcasts message m with identifier k. A correct broadcaster increments k sequentially at every broadcast, starting with $k = 1$. Similarly, when a process q invokes *deliver(k,m,p)*, we say that q delivers (k, m) from p.

In simple terms, CTBcast is a multi-shot abstraction that prevents correct processes from delivering different messages from a given broadcaster p for the same identifier k. CTBcast is parameterized by a tail t, which specifies which messages are guaranteed to be delivered. Informally, in CTBcast, a correct process q is only required to deliver the last t messages broadcast by a correct process p, while the delivery of previous messages is best-effort.

CTBcast has the following properties:

Tail-validity If a correct process p broadcasts (k, m) and never broadcasts a message (k', m') with $k' \geq k+t$, then all correct processes eventually deliver (k, m).

Agreement If p and q are correct processes, p delivers (k, m) from r, and q delivers (k, m') from r, then $m = m'$.

Integrity If a correct process delivers (k, m) from p and p is correct, p must have broadcast (k, m).

No duplication No correct process delivers $(k, *)$ from p twice.

The difference between CTBcast and CBcast lies in their validity property. Tail-validity implies that a correct process is only obliged to deliver a message m from p if m is among the last t messages broadcast by p. When t=∞, tail-validity reduces to CBcast's validity.

The infinite tail of CBcast is what prevents it from recycling memory. Indeed, given that the broadcaster cannot distinguish between network asynchrony and receiver failures [32], it is required to keep re-transmitting all messages until they are explicitly acknowledged. Thus, in CBcast, the broadcaster can garbage collect messages only after they have been acknowledged by all receivers. As a result, once a single process fails, memory cannot be recycled and any correct implementation of CBcast must block after running out of memory. By not enforcing the delivery of *old* messages, CTBcast's tail-validity lets processes recycle the memory dedicated to these messages. This is why CTBcast requires only finite memory while CBcast does not. In Section 5, we show that despite its weaker semantics, CTBcast is sufficient for solving consensus.

4.2 Algorithm

Algorithm 1 implements CTBcast using finite memory with a fast/slow path approach that avoids signatures and disaggregated memory in the common case. For pedagogical reasons, we assume that a designated process is the broadcaster while the others are receivers. Each receiver owns an array of t Single-Writer Multiple-Reader (SWMR) regular registers. Each register is only writable by its owner, but is readable by all processes. The regularity of the registers forces READs that execute concurrently to a WRITE to return either the value being written or the previous one. Moreover, each process uses a Tail Broadcast (TBcast) primitive which ensures the delivery by correct processes of the last $2t$ messages broadcast through it, but does not prevent equivocation. Formally, TBcast has all properties of CTBcast except agreement.

Implementing TBcast using finite memory is simple. The broadcaster buffers its last $2t$ messages and retransmits them until it receives acknowledgements from all receivers. To broadcast a new message when the buffer is full, the broadcaster makes room for it by evicting the oldest buffered message.

Algorithm 1: Consistent Tail Broadcast

```
1  # at the broadcaster
2  def broadcast(k, m):
3    TBcast-broadcast <LOCK, k, m>
4    TBcast-broadcast <SIGNED, k, m, sign((k, m))>

6  # at receivers:
7  SWMR[me] = [(−1, ⊥, ⊥), ... ] # array of t slots
8  delivered = [−1, ... ] # array of t slots
9  locks = [(−1, ⊥), ... ] # array of t slots
10 locked = [[(−1, ⊥), ... ], ... ] # array of t∗n slots

12 upon TBcast-deliver <LOCK, k, m> from p:
13   k', _ = locks[k%t]
14   if k > k':
15     locks[k%t] = (k, m)
16     TBcast-broadcast <LOCKED, k, m>

18 upon TBcast-deliver <LOCKED, k, m> from q:
19   k', _ = locked[q][k%t]
20   if k > k':
21     locked[q][k%t] = (k, m)
22     if locked[r_0][k%t] == ... == locked[r_{n−1}][k%t]:
23       deliver_once(k, m)

25 upon TBcast-deliver <SIGNED, k, m, sig> from p:
26   if valid(sig, (k, m), p):
27     k', m' = locks[k%t]
28     if k > k' or k == k' and m == m':
29       locks[k%t] = (k, m)
30       SWMR[me][k%t].write((k, sig, m))
31       for each (k', s', m') in SWMR[∗][k%t]:
32         if valid(s', (k', m'), p):
33           if k' == k and m' != m:
34             return # Byzantine broadcaster
35           if k' > k and k' ≡ k (mod t):
36             return # out of tail
37       deliver_once(k, m)

39 def deliver_once(k, m):
40   if k > delivered[k%t]:
41     delivered[k%t] = k
42     trigger deliver(k, m)
```

As mentioned, Algorithm 1 has a low-latency fast path that avoids signatures and disaggregated memory. It also incorporates a fall-back slow path for liveness. For presentation simplicity, it triggers the slow path in parallel to the fast path (lines 3 and 4), but in reality, uBFT triggers it when replicas fail to decide on new client requests after some configurable duration. In addition to the shared SWMR registers, receivers use three finite-size local arrays for bookkeeping (lines 8-10).

In the fast path, the broadcaster first TBcast-broadcasts its message alongside its identifier within a `LOCK` message (line 3). When receivers deliver this message (line 12), they commit not to deliver any other message for the given identifier, and tell other receivers about their commitment by broadcasting a `LOCKED` message (line 16). Receivers use `locks` (lines 13-15) to avoid committing to different messages for the same identifier. Importantly, receivers store only up to t commitments in this array, by evicting earlier commitments that alias to the same index $k\%t$ (line 15). When receivers learn that everyone committed to the same message (lines 18-22), they know that no correct replica will deliver a different message and thus deliver it (line 23).

In the slow path, the broadcaster additionally TBcast-broadcasts a signed version of its message (line 4). After TBcast-delivering a signed message (line 25), receivers verify its signature (line 26). Then, they check that they have not committed to a different message for the same k (line 28), and ensure that they will not do so in the future (line 29). Subsequently, they copy the signed message to their SWMR register associated with the message identifier k (line 30), before reading the associated SWMRs owned by other receivers (line 31). Receivers ignore messages with invalid signatures (line 32) and abort delivery if they detect a different message for the same identifier (line 33). In case receivers detect another message with a higher identifier that is associated with the same SWMR registers (line 35), they drop their own message as it no longer belongs to the tail. Otherwise, they deliver it (line 37).

The correctness of the slow path of Algorithm 1 hinges on that all correct processes will find the message copied by the *fastest* correct replica when reading the registers. Thus, they can deliver no message other than the first copied, hence preserving agreement. The fast and slow paths are linked together via the `locks` array (lines 15 and 29), which ensures that whichever path executes first forces the value of the message for the other path. Note that when the broadcaster is Byzantine, correct processes are allowed not to deliver. A more detailed proof of correctness is given in the extended version of this paper [5].

5 STATE MACHINE REPLICATION

Like all leader-based BFT consensus protocols, uBFT's protocol has the same high-level layout as PBFT [24]: it shares naming conventions and splits the protocol in similar phases. However, uBFT has different goals ($2f+1$ fault tolerance, finite memory, microsecond latency), different assumptions (disaggregated memory), and a different non-equivocation mechanism (Consistent Tail Broadcast (CTBcast)). As a result, uBFT's protocol is structurally different from PBFT, as we describe in this section.

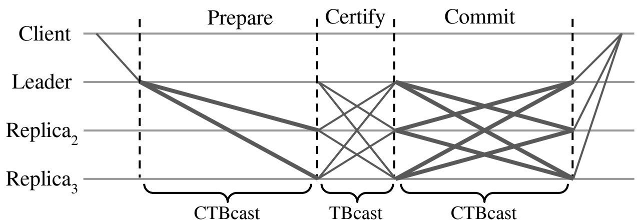

Figure 3: Communication pattern of uBFT's slow path. Bold lines represent messages sent over CTBcast. Thinner lines represent direct messages.

We start by giving an overview of our consensus protocol. Then, we analyse its slow path and explain how it deals with finite memory and view changes. Finally, we describe how its fast path leverages unanimous and timely collaboration of replicas to achieve microsecond-scale latency. For presentation clarity, this section gives only an informal description of our protocol. Its pseudocode is given in Section 11 and detailed correctness arguments are given in the extended version of this paper [5].

5.1 Basic Protocol

From a high-level point of view, the slow path of our consensus protocol—shown in Figure 3—has three phases: *Prepare*, *Certify* and *Commit*. After the leader receives a signed request from a client, it *proposes* it by broadcasting a `Prepare` message via CTBcast. When replicas (including the broadcaster) deliver this message, they proceed to its certification. Each replica signs the proposal and TBcast-broadcasts its signature in a `CERTIFY` message. Then, it waits to aggregate $f+1$ signatures on the proposal, which constitute an unforgeable proof that the proposal was emitted by the leader. Moving on to the Commit phase, the replicas broadcast via CTBcast a `COMMIT` message containing the aforementioned proof. Finally, after delivering $f+1$ `COMMIT` messages, replicas apply the client's request to their local state machine and reply to the client. Note that we use TBcast instead of the more expensive CTBcast when broadcasting `CERTIFY` messages: equivocation of `CERTIFY` messages does not harm correctness as all certificates involve at least one correct replica, and all correct replicas certify the same proposal due to the CTBcast in the Prepare phase.

So far, the described protocol replicates a single client request. Similarly to PBFT, uBFT uses a sliding window to run its consensus protocol on a series of slots. As the leader receives multiple requests, it proposes each one in a different slot, handling many slots in parallel. uBFT also uses PBFT's application checkpoints to throttle the number of concurrent request proposals. This mechanism limits the impact of a Byzantine leader, and bounds the number of relevant messages at any point in time. An application checkpoint is signed by $f+1$ replicas and includes (1) the state of the application after applying the first i replicated requests to it, and (2) an implicit authorization to work on slots $[i+1, i+\mathit{window}]$.

5.2 Non-Equivocation at the Consensus Level

An important aspect of uBFT's consensus protocol is how it prevents the leader from sending conflicting proposals. CTBcast only partially solves this issue: it prevents a Byzantine leader from sending conflicting proposals in a single message, but it does not prevent it from sending conflicting proposals across multiple messages. Conceptually, a process can ensure that another process has not equivocated if the former knows the entire history of messages broadcast by the latter. For this reason, processes in our protocol interpret messages of other processes in FIFO order. However, CTBcast does not guarantee the delivery of all broadcast messages due to its tail-validity property, which might prevent processes from delivering *all* the messages from a correct one in FIFO order. We solve this issue by pairing CTBcast with *CTBcast summaries*.

A CTBcast summary is an unforgeable synopsis of what has been broadcast by a process p via CTBcast up to a given CTBcast identifier. Summaries constitute certificates that are signed by $f+1$ replicas, which have witnessed the messages broadcast by p and assert that p has not equivocated at the consensus level. When a process receives a summary about p up to a certain CTBcast identifier i, it is able to continue handling p's new messages above i in FIFO order. Essentially, CTBcast summaries restore FIFO delivery that may have been broken by the tail-validity of CTBcast.

CTBcast summaries are generated interactively. Every t consecutive CTBcast messages (t is CTBcast's tail parameter) that a replica r delivers from p, r participates in creating a summary about the state of p. Replica p knows how many messages it has broadcast and blocks waiting for a summary of its state every t messages. Using its previous summary, p can bring correct processes that missed some of its messages up to speed, and help them deliver the last t messages it broadcast so far. Even if f Byzantine replicas fail to help building p's next summary, all correct replicas would eventually collaborate to generate it, thus ensuring the liveness of the summaries and allowing p to keep broadcasting.

It is important to ensure that CTBcast summaries have finite size. To this end, a process keeps only a limited number of messages that it CTBcast-delivers from others, by maintaining a window of consensus slots for every other broadcasting process. If a process p CTBcast-delivers a message from another process q out of the window it maintains for q, p considers q as being Byzantine. uBFT requires processes to CTBcast-broadcast application checkpoints in order to slide their window on the receivers' side. When a receiver slides a broadcaster's window forward, it drops messages referring to slots that fall out of it, since the consensus slots they refer to have been checkpointed. This way, the entire relevant state of a broadcaster is a combination of a consensus window range and the collection of CTBcast messages that fall into it.

5.3 View Change

To tolerate faulty leaders, uBFT follows PBFT's *view change* approach. Briefly, every view has a dedicated leader determined in a round-robin manner. If a replica does not make progress or believes that the leader is censoring client requests, it moves to the next view by CTBcast-broadcasting a `SEAL_VIEW` message. The leader of the new view first transfers any potentially applied requests to the new view by CTBcast-broadcasting a `NEW_VIEW` message before proposing new requests, as detailed below.

The `NEW_VIEW` message contains the latest application checkpoint, as well as all `COMMIT` messages about the open slots following

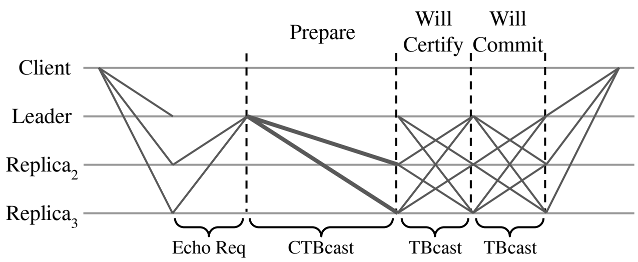

Figure 4: Communication pattern of uBFT's fast path. Bold lines represent messages sent over CTBcast. Thinner lines represent direct messages.

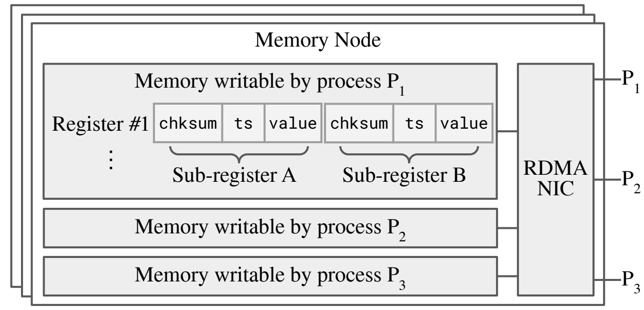

Figure 5: Reliable SWMR Regular registers using RDMA.

this checkpoint from f+1 replicas. More precisely, when a replica p receives a SEAL_VIEW message from another replica q, p proceeds with generating a certificate share about q's state. In fact, this state includes q's latest application checkpoint, as well as the latest COMMIT messages sent by q for each one of q's open slots. The content of the NEW_VIEW message consists of f+1 matching certificate shares about f+1 different replicas. The broadcast NEW_VIEW message constrains the values that the new leader can propose for open slots. That is, for every open consensus slot, the new leader is required to propose the value of the COMMIT message with the highest view number (if any).

The view change mechanism prevents a leader from omitting requests that were decided in previous views. Intuitively, if a correct replica decided on a request in some view, it must have received f+1 matching COMMIT messages. As the leader collects certificates about f+1 replicas, it must necessarily include at least one certificate with a COMMIT message for all decided requests.

5.4 Fast Path

To operate at the microsecond scale, uBFT incorporates a fast path—shown in Figure 4—that moves signatures out of the critical path in times of synchrony and unanimous collaboration. Similar to the slow path, the fast path has three phases: Prepare, which is common with the slow path but executes the fast path of CTBcast, followed by *WillCertify* and *WillCommit*. The last two phases replace the Certify and Commit phases of the slow path with inexpensive rounds of Tail Broadcast.

The operation of the fast path is simple. For a given consensus slot, after the end of the Prepare phase, replicas broadcast a WILL_CERTIFY message and wait to receive the same message from all others. Once received, they proceed with broadcasting a WILL_COMMIT message and again wait for unanimity before deciding on the proposed value. Both messages contain solely the view number and the consensus slot. These messages are essentially promises that their broadcaster will run the slow path before CTBcast-broadcasting its next SEAL_VIEW message. By broadcasting WILL_CERTIFY, a replica promises to participate in certifying the PREPARE message. With WILL_COMMIT, it promises to CTBcast-broadcast the resulting certificate within a COMMIT message.

The safety of this scheme is intuitive. If a replica receives $2f$+1 WILL_CERTIFY messages, it knows that at least f+1 correct replicas will certify the PREPARE message in the Certify phase. Similarly, receiving $2f$+1 WILL_COMMIT messages means that at least f+1 correct replicas will send a COMMIT message in the Commit phase and thus no other value will be decided for this slot.

The fast path also takes care of finite memory as it does not keep promises forever: it drops the promises that refer to consensus slots included in an application checkpoint. These checkpoints along with the CTBcast summaries make up the required background signatures of uBFT's fast path.

Lastly, the fast path eschews signatures between clients and replicas by having the clients send unsigned requests to all replicas, instead of a signed request to the leader. A replica endorses a PREPARE message on a client request only if it also received it from the client. Thus, the fast path contains an additional communication round (denoted *Echo Req*) in which the leader waits for the followers to echo the client request before proposing it. This way, a Byzantine client cannot cause a leader to get stuck (and thus incur a view change) by not sending its requests to all correct replicas.

6 IMPLEMENTATION

The design of uBFT relies on disaggregated memory to build the shared registers used by CTBcast, and on a fast networking fabric to achieve its microsecond-level latency. This section explains how we use RDMA [85] to achieve these goals.

6.1 Reliable SWMR Regular Registers

The CTBcast component of uBFT requires reliable SWMR regular registers. *Reliable* means that the registers do not fail, i.e., READs and WRITEs always complete. *SWMR* means Single-Writer Multiple-Reader: each register has an owner, which is a replica that is allowed to write to it, while all other replicas may only read the register. *Regular* means that, when a READ executes concurrently to a WRITE, the former should return either the value that is being written or the previous one. We explain below how to implement each property using RDMA.

SWMR Register. We implement SWMR registers using RDMA permissions. RDMA splits memory into regions, each with different access permissions based on a token. We create a read-write and a read-only region for the same memory range, give the read-write token to the writer of the register, and the read-only token to the other replicas.

Regular Register. RDMA-exposed memory is atomic (hence regular), but only at an 8-byte granularity. While RDMA WRITEs have left-to-right ordering, RDMA READs do not. Thus, an RDMA READ concurrent with a WRITE may return partially written data, mixing old and new values. To detect this problem, we use checksums, as in Pilaf [69]. A simple approach, where a reader retries until the checksum is valid, violates liveness as a Byzantine writer can write bogus checksums. To avoid such scenario, we follow an evolved double-buffering strategy, which ensures that a reader is always able to find a complete WRITE or detect the owner of the register as being Byzantine. As depicted in Figure 5, each register is made of two sub-registers. Each WRITE to a given register is directed to one of the sub-registers in a round-robin manner. To write a value, the writer prefixes it with a logical timestamp (denoted ts) and a checksum. Importantly, the writer waits for δ (the known communication bound after GST) between two WRITES to the same register. To perform a READ, the reader reads both sub-registers at once and, out of the values with a valid checksum, returns the one with the highest timestamp. [3] If both checksums are invalid and the READ took less than δ, the writer is Byzantine and a default value is returned. Otherwise, the READ is retried. The writer is also deemed Byzantine if both sub-registers have the same timestamp. This scheme works in an eventually synchronous system with bounded-drift clocks [34, 41]: when the writer and a reader are correct, a READ does not overlap with two WRITEs, so one of the sub-registers will have a valid checksum, and ordering based on timestamps ensures regular register semantics.

Reliable Register. We replicate each register to $2f_m$+1 memory nodes, where each memory node exposes its memory over RDMA (Figure 5). Here, f_m is the maximum number of memory nodes that may crash. While memory nodes add to the total number of replicas in the system, these nodes do not replicate the application, and they can be shared among many replicated applications, as each application takes a small amount of space in the memory nodes (§7.6). Our register replication scheme is straightforward. WRITEs are issued to all memory nodes in parallel and return after having completed at f_m+1 of them. READs are also issued to all nodes in parallel, wait for f_m+1 of them to complete, and return the value of the regular register with the highest timestamp. This scheme tolerates Byzantines replicas, as READs and WRITEs always complete at a majority, and preserves regular semantics. Indeed, when a READ is not concurrent to any WRITE, intersecting quorums ensure that the last written value is returned. In case of concurrency, the READ intersects with the concurrent WRITE in some register and with the last completed WRITE in some other register. Thus, it will return a value no older than the value of the last completed WRITE, ensuring regularity.

6.2 A Fast Message-Passing Primitive

To achieve microsecond-scale communication, uBFT implements a fast one-way messaging primitive between a sender and a receiver,

[3]The hardware is allowed to reorder RDMA READs following RDMA WRITEs when issued to different Queue Pairs (QPs) [11]. To ensure regularity, i.e., that subsequent register READs see the RDMA-written value, a register WRITE only returns after the PCIe WRITE transaction reaches the last-level cache (L3). We do so by issuing an RDMA READ after the RDMA WRITE to the same QP—which acts as a PCIe fence [40]—and only considering that the register WRITE completes when the RDMA READ completes.

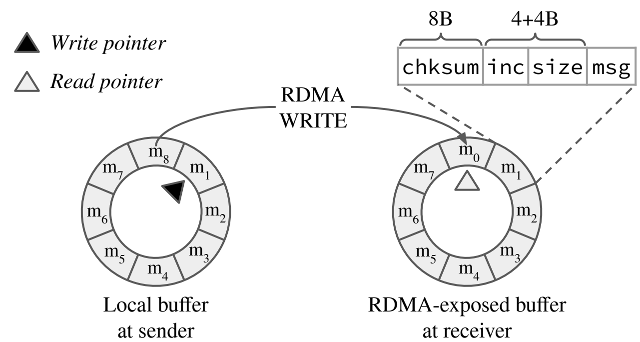

Figure 6: uBFT's RDMA-powered message-passing primitive.

where the receiver is required to deliver only the last t messages of the sender, similarly to CTBcast. This primitive allows an implementation without receiver acknowledgements, which we found to be important for microsecond-scale performance.

Figure 6 depicts the implementation of this primitive over RDMA. The receiver has a circular buffer exposed over RDMA; the buffer is divided into t slots of equal size large enough for the largest message. Briefly, the sender RDMA-writes messages to the receiver's buffer, while the latter scans its local buffer for new messages. There are no acknowledgments: the sender overwrites old messages with newer ones, even if they were not yet received.

We now explain this implementation in detail. The sender allocates a mirror image of the receiver's buffer in its local memory, and maintains locally a *write pointer* to the slot for its next message. Each slot has a header composed of a checksum, an incarnation number (the number of times it was written), and a message size. To send a new message, the sender writes it to the slot pointed by the writer pointer and fills its header. Then, it issues an RDMA WRITE to the corresponding slot in the receiver's memory, and marks the slot as unavailable until it is notified of the completion of the WRITE by the RDMA NIC. Finally, the sender advances its write pointer. If the pointed slot is unavailable, the new message is queued in a second (not depicted) circular buffer. This buffer acts as a staging area: it forwards its messages for transmission when slots become available, and evicts the oldest queued message to accommodate a newer one.

The receiver maintains a *read pointer* to the slot where it will read the next message. The receiver polls this slot for a particular incarnation number, which identifies the next message it expects to find. Once this incarnation number is seen, the receiver copies the entire message to a private buffer in order to avoid interfering WRITEs on the same slot. Then, the receiver checks the incarnation number again in the copied message. If the incarnation number has not changed, the sender verifies the checksum before delivering the message (rescheduling the polling if the checksum is invalid). If the receiver finds a higher incarnation number than expected, it concludes that some older messages may exist in other slots of the buffer that will have to be delivered first. So, the receiver aborts the delivery and advances its pointer to the oldest undelivered message. With this strategy, the receiver guarantees FIFO delivery of the last t messages.

Table 1: Hardware details of machines.

CPU	2x Intel Xeon Gold 6244 CPU @ 3.60 GHz (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-generic`
RDMA Driver	Mellanox OFED `5.3-1.0.0.1`

This scheme has two benefits: it uses practically bounded memory and avoids acknowledgements. The latter—even when batched—increase the application's tail latency as scheduling an acknowledgement alone takes ≈300 ns [50], which is time lost handling incoming events. Instead, by the End-to-End Principle [79], acknowledgements are piggybacked in SMR-level messages.

7 EVALUATION

We evaluate the various performance traits of uBFT and verify its suitability as a BFT SMR system for microsecond applications. We aim to answer the following:

- How much latency does uBFT induce on the replicated applications (§7.1)?
- How does the replication latency of uBFT compare to other SMR systems (§7.2)?
- How do the internal components of uBFT contribute to its end-to-end latency (§7.3)?
- How does our implementation of CTBcast perform in comparison to SGX-based non-equivocation mechanisms (§7.4)?
- How does the tail parameter of CTBcast impact uBFT's tail-latency (§7.5)?
- What is the memory consumption of uBFT (§7.6)?

We evaluate uBFT in a 4-node cluster, the details of which are given in Table 1. 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 exclusively use the memory bank closest to this socket.

Our implementation [4] measures time using the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. The function uses the TSC clock source of Linux for efficient and accurate timestamping [76].

In all experiments we deploy 1 client and 3 replicas, and take at least 10,000 measurements. Additionally, we set the consensus window to 256 requests and—unless stated otherwise—the tail parameter of CTBcast to 128 messages.

Applications. We integrate uBFT with MemCached [50], Redis [80] and Liquibook [73]. MemCached and Redis are non-replicated high-performance KV-stores. Liquibook is a data structure that implements a financial order matching engine. We also integrate all the aforementioned applications with Mu [3], the SMR system that has the lowest replication latency (to our knowledge) but tolerates only crash faults. In all applications, the client sends messages using uBFT's RPC mechanism. Additionally, using a no-op application, we compare uBFT against MinBFT [88], a state-of-the-art $2f+1$ BFT SMR system with a publicly available implementation which uses

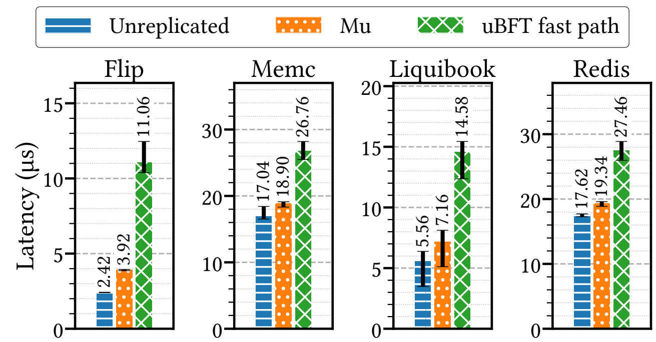

Figure 7: End-to-end latency of different applications when either not replicated or replicated via Mu and uBFT's fast path. The printed values are the 90th %-iles. The whiskers show the 50th and 95th %-iles.

Intel's SGX [28]. SGX provides a secure CPU enclave for executing arbitrary code, thus offering a general-purpose trusted computing base.

Implementation Effort. We implemented uBFT on top of a framework [41] that facilitates the use of RDMA. Our prototype spans 11,750 lines of C++, out of which 2,966 are dedicated to CTBcast. The prototype includes all features on the critical path of a complete implementation: the only major unimplemented features are application and replica state transfers. We use Dalek's implementation of EdDSA over Curve25519 [61] for public-key cryptography, BLAKE3 [74] for HMACs and xxHash [27] for checksums.

7.1 End-to-End Application Latency

Figure 7 explores the replication overhead that uBFT induces to end applications. We compare the latency of its fast path against the unreplicated latency, and the latency when replicating with Mu. We study four applications: Flip, a toy application that reverses its input, as well as Memcached, Redis, and Liquibook.

The KV-stores use 16 B keys and 32 B values. Our workload is 30% `GET`s, out of which 80% succeed and return a non-empty value. Liquibook's requests are 32 B. Its responses range from 32 B to 288 B, depending on how many orders match. 50% of the orders are `BUY` and 50% `SELL`. Finally, Flip's requests and responses are 32 B long.

From the figure, observe that uBFT is consistently slower than Mu by approximately 7.5 µs at the 90th percentile. uBFT's additional induced end-to-end latency is most significant for ultra fast applications, such as Flip, for which our system is 3 times slower than Mu. As the application's unreplicated latency increases, uBFT's latency overhead diminishes. For Liquibook, it is 2× slower than Mu and for the KV-stores, it is only 1.5× slower. At the same time, uBFT's replication increases the latency variance (i.e., the difference between the 50th and 95th percentiles) compared to Mu, which is a consequence of the complexity of our system. More precisely, uBFT's RPC requires one more round of communication than Mu's to ensure that all correct replicas have received the client request. Additionally, uBFT's replication scheme requires 4 rounds of broadcast before replying to the client instead of a single majority `WRITE` in Mu. Overall, uBFT's fast path has 4 additional rounds of communication

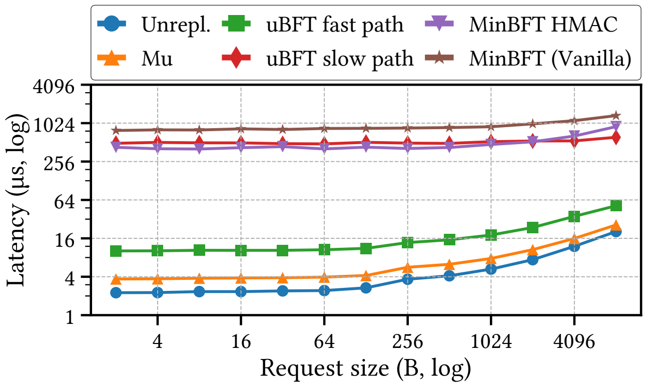

Figure 8: Median end-to-end latency for different request sizes of an unreplicated no-op application, as well as its latency when replicated with Mu, uBFT and MinBFT.

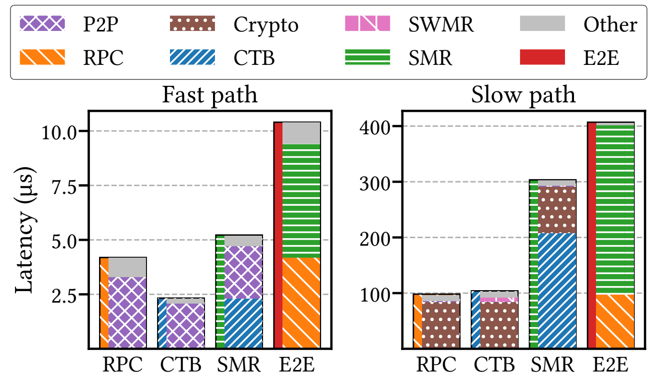

Figure 9: Recursive decomposition of the end-to-end latency of uBFT's fast and slow path when replicating Flip with requests of 8 B.

compared to Mu, hence the higher tail latency. Nevertheless, uBFT does not worsen the variance by more than 3 µs.

In a nutshell, with uBFT, assuming a timely network and the absence of failures, an unreplicated application that operates at the microsecond-scale envelope requires at most an extra 10 µs to become Byzantine fault tolerant.

7.2 End-to-End Replication Latency

We now explore the impact of the size of requests on the end-to-end latency of uBFT. Figure 8 shows the median client-side latency for various request sizes to a no-op application with different replication schemes. The service replies with responses matching the size of requests.

As expected, the lowest latency is achieved without any replication (denoted *Unrepl.*). Here, the end-to-end latency ranges from 2.2 µs to 20 µs, which is attributed to communicating with the server side using our RPC. The other lines show the service replicated with: Mu, our BFT SMR solution, and MinBFT, a $2f+1$ BFT alternative.

Mu's replication increases the end-to-end latency by up to 64% for small requests and by at most 26% for 8 KiB ones. In the absence of failures, Mu's leader replicates the requests it receives by just RDMA-writing them to its followers. uBFT's fast path exhibits higher latency than Mu due to its 4 additional rounds of communication, yet it only increases the overhead compared to the latter by at most 175%, even though it offers Byzantine fault tolerance.

One could expect that uBFT's fast path would come at the cost of high latency in the slow path, yet this is not the case. We compare our slow path against MinBFT, with the latter operating in two configurations. In its vanilla configuration, MinBFT uses HMACs only between the replicas, however the clients sign the requests sent to the service using public-key cryptography, leading to a minimum end-to-end latency of 566 µs. We modify MinBFT to also use SGX in the clients, thus replacing their expensive cryptography with HMACs. Since our setup does not offer SGX, we stub it using the latency results of Section 7.4. Note that MinBFT is not an RDMA-tailored application and it uses standard TCP in its implementation.

To increase the fairness of our comparison with MinBFT, we use Mellanox's VMA library [86] to replace MinBFT's TCP stack with a kernel-bypass alternative that uses RDMA NICs for increased performance. The end result is that uBFT's slow path is faster than vanilla MinBFT (up to 109%) and at most 24% slower than its purely HMAC-based variant, even though our slow path uses public-key cryptography.

7.3 Latency Breakdown

To get a better understanding of uBFT's end-to-end latency, we analyze the latency of its internal components.

Figure 9 shows a recursive decomposition of the latency of an 8 B Flip request into its constituents: remote-procedure call, Consistent Tail Broadcast and replication (denoted *RPC*, *CTB* and *SMR* respectively). The rightmost columns of the figure show the client-perceived latency (denoted *E2E*). Each bar has two regions: the narrow left one shows the overall latency of the component, while the wider right one shows its decomposition.

In our decomposition, we identify four primitive sources of latency. First is the time for communication over our message-passing primitive (denoted *P2P*). Next is the time for producing and verifying signatures (denoted `Crypto`), and the time for reading and writing to the disaggregated memory registers (denoted *SMWR*). Both of these are only relevant in uBFT'slow path. The `Crypto` primitive goes beyond pure cryptographic computation, as it includes the synchronization cost of issuing the operation to a thread pool and getting back the result. Lastly, there is the time spent in bridging these basic primitives and connecting the components (denoted `Other`). This last category includes copying buffers, delays between the arrival and processing of asynchronous events, etc.

We can see that, in the fast path, most of the time is spent in communication. Given the small size of the messages, the potential avenues for improving the overall latency is to either reduce the number of communication steps or reduce the latency of the underlying network fabric. In the slow path, public-key cryptography dominates. Given that signatures are unavoidable in our setting [6], achieving better latency requires faster cryptographic primitives. Also, notice that the cost of accessing disaggregated memory—which is part of `CTB`—is negligible, as it accounts for 14 µs (3.5%) of the end-to-end latency.

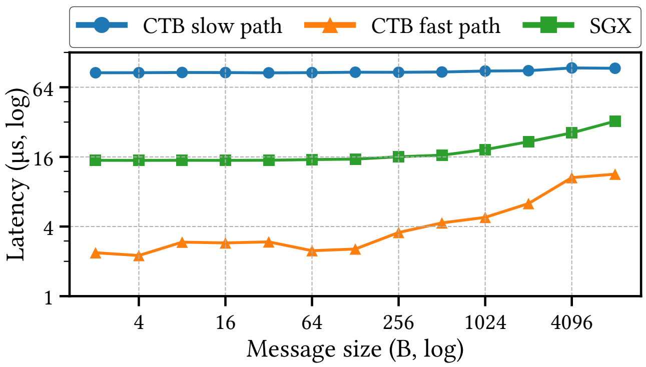

Figure 10: Median latency of multiple non-equivocation mechanisms for different message sizes.

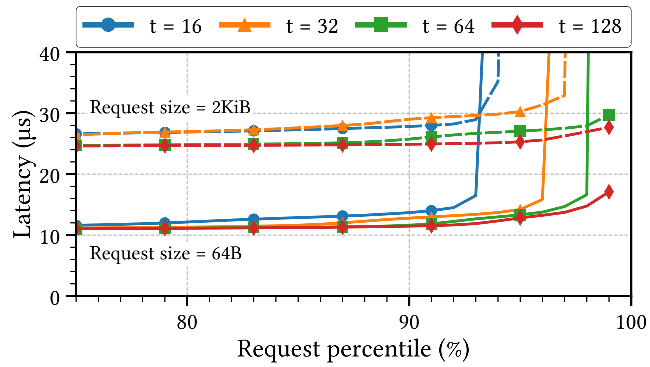

Figure 11: uBFT's tail latency for different CTBcast tails t for 2 KiB requests (top) and 64 B requests (bottom).

7.4 Latency of Non-Equivocation

Figure 9 demonstrates that CTBcast accounts for a substantial portion of the end-to-end latency of our system's fast and slow paths. By using disaggregated memory, our implementation of CTBcast has an advantage over solutions that use CPU enclaves, since disaggregated memory is a much smaller and simpler trusted computing base. Moreover, the fast/slow path approach of our implementation allows it to operate fast in the common case by entirely bypassing signatures and the trusted computing base. This section quantifies the performance of our implementation of CTBcast against the de facto way of preventing equivocation in modern systems [16, 60, 88], namely a trusted counter implemented on Intel's SGX.

Non-equivocation mechanisms based on trusted counters work by securely binding a monotonically increasing sequence number to each message broadcast by a process. All enclaves store a local *counter* and share a common *secret*. Before broadcasting a message, a process feeds it to its local enclave and gets back a proof of non-equivocation of the form $HMAC_{secret}(msg||counter||process\ id)$. The recipients of a message use their own local enclave to verify the authenticity of the HMAC, and thus prevent equivocation. Note that, as authentication takes place exclusively in the enclaves, this non-equivocation mechanism avoids the use of expensive public-key cryptography.

Figure 10 shows the median latency of the two non-equivocation mechanisms between a sender and two receivers. The lowest latency is achieved by the fast path of our implementation of CTBcast and ranges from 2.2 µs to 11 µs depending on the message size. In this mode of operation, CTBcast delivers on unanimous and timely participation of all processes to avoid signatures and prevent equivocation using merely two rounds of Tail Broadcast. CTBcast's slow path (triggered under failures) relies on public-key cryptography, which dominates the latency and raises it to approximately 86 µs. The latency of preventing equivocation using SGX includes accessing the enclave twice, once in the sender and once in the receiver, as well as broadcasting the message.

Due to the lack of SGX in our RDMA experimental setup, we evaluate the cost of accessing the SGX on different hardware to get a good approximation. The SGX hardware is provided by a machine with an Intel i7-7700K CPU running at 4.2 GHz, which is 0.6 GHz higher than the CPU frequency of our RDMA setup. Accessing the enclave (once) ranges from 7 µs to 12.5 µs, which makes preventing equivocation using the SGX take at least 16 µs. The resulting latency is shown in the middle line of Figure 10.

Overall, for both non-equivocation approaches, larger message sizes lead to a linearly higher latency, due to the hashing and communication latency. CTBcast's fast path is up to 6.5× faster than the SGX solution, by taking advantage of the ultra fast communication.

7.5 Impact of CTBcast's Tail on Tail Latency

Figure 11 shows how the size of the tail in CTBcast (parameter t) affects the client's tail latency. We focus on uBFT's fast path and execute Flip with small 64 B requests and larger 2 KiB ones. For both request sizes, we explore four *tail* parameters.

For smaller values of t, we see a latency spike indicative of thrashing as we move to higher percentiles. This spike occurs because CTBcast uses a double buffering mechanism with cryptographic summaries (§5.2) to switch between them; if both buffers fill before a summary occurs (due to a small t), CTBcast stalls. The smaller the t, the sooner the buffers fill, the more often CTBcast stalls, and hence the lower the percentile of the spike. For small requests, a tail t=128 avoids thrashing up to the 99th percentile. For larger requests, t=64 suffices, as filling the buffers takes more time, giving more time for the summary to occur.

7.6 Memory Consumption

Given the fundamental goal of uBFT to operate using finite memory, we monitor the consumption of disaggregated memory and the local memory consumption at the leader replica, while re-running the experiment of Section 7.5.

Table 2: uBFT replica (top) and disaggregated (bottom) memory usage for different CTBcast tails t and request sizes.

Request size	**t = 16**	**t = 32**	**t = 64**	**t = 128**
64 B	0.46 GiB	0.47 GiB	0.49 GiB	0.53 GiB
2 KiB	4.3 GiB	4.5 GiB	4.8 GiB	5.5 GiB
Disag. Mem.	20 KiB	40 KiB	81 KiB	162 KiB

Table 2 summarizes the results for two different request sizes (64 B and 2 KiB) and four different *tail* parameters of CTBcast (16, 32, 64 and 128). For the small 64 B requests, the local memory consumption starts at 0.46 GiB. This is the entire memory that uBFT preallocates at startup and uses during its lifetime. When CTBcast's tail t increases, uBFT's memory consumption increases linearly by ≈1 MiB for each additional message in the tail. For the large 2 KiB requests, the memory consumption starts at 4.3 GiB (t=16) and increases at a rate of ≈11 MiB per message.

uBFT consumes little disaggregated memory. The last row of Table 2 shows the memory used at a single memory node. This amount is independent of the size of requests and depends only on CTBcast's tail t: messages sent over CTBcast are transmitted using our fast message-passing primitive; upon receiving a message, a receiver writes to disaggregated memory only the message's id and its fingerprint (a 32 B cryptographic hash); the register implementation (§6.1) stores two copies, each with an 8 B checksum. To save space, registers use the identifiers of messages as their timestamps.

8 RELATED WORK

uBFT uses RDMA to instantiate disaggregated memory. Prior work identifies some benefits and downsides of using RDMA in an adversarial environment. Aguilera *et al.* [2, 6] propose new RDMA-based techniques to enhance the resilience and performance of BFT algorithms. That work is abstract and far from practical solutions: it requires infinite memory and solves only single-shot consensus, stopping short of a solution for an SMR system. Rüsch *et al.* [78] design a Byzantine fault tolerant system that uses RDMA to improve performance, but requires $3f$+1 processes.

Prior work identifies vulnerabilities in the current generation of RDMA hardware and proposes ways to mitigate them [77, 84]. That work is orthogonal to uBFT and could be applied to it. We expect that these problems will eventually be fixed with future NICs.

With a black-box mechanism to prevent equivocation, only $2f$+1 replicas are required for BFT [17, 18, 26]. Several BFT systems achieve that using trusted hardware as the black box: attested append-only memory (A2M) [25] uses a trusted log, TrInc [59] and MinBFT [88] use a trusted counter, Hybster [16] uses Intel's SGX, CheapBFT [51] uses FPGAs, and H-MFT [94] uses trusted hypervisors to implement write-once tables. By separating execution from agreement [91], one can reduce the number of execution replicas to $2f$+1, but $3f$+1 replicas are still required for agreement. Avocado [13] implements a high-performance replicated confidential KV-store using CPU enclaves as the trusted computing base.

Blockchain systems also tolerate Byzantine failures, but their latency is in the seconds or minutes due to their heavy use of cryptography [37, 52], proof of work [35, 71], and/or batching [21, 92]. The recent SplitBFT [65] uses SGX and $3f$+1 replicas to strengthen the safety and confidentiality of blockchains in public clouds.

While most of the prior work is not focused on microsecond-scale latency (and hence came up with different solutions from ours), some recent SMR systems aim for lower latency. Mu [3] is highly optimized and provides microsecond-scale performance, but tolerates only crash failures. SBFT [38] tolerates Byzantine failures and uses a fast path to improve latency, but does not achieve microsecond-scale performance due to its use of cryptography. BFT SMR systems with $3f$+1 replicas can avoid cryptographic signatures, for example, in PBFT's optimized implementations [24]. However, this is impossible in a system with $2f$+1 replicas [26]; the key to uBFT's performance is thus avoiding signatures on the fast path.

Carbink [93] and Hydra [56] build reliable disaggregated memory to improve memory utilization in a cluster, albeit without support for concurrent shared access. MIND [55], GAM [23] and Clover [87], on the other hand, provide reliable shared memory, but they do not tolerate Byzantine writers.

9 DISCUSSION

Does microsecond 2f+1 BFT require disaggregated memory? To achieve microsecond-scale BFT SMR, one must avoid the use of expensive signatures and trusted components in the critical path. uBFT does so via a fast/slow path design that uses disaggregated memory in the slow path. This leads to a small trusted computing base, but there may be other ways to achieve non-equivocation in CTBcast's slow path without affecting fast-path performance.

Can uBFT work with a Byzantine network? uBFT assumes network connections are authenticated and tamper-proof, which is realistic in data centers, where widely deployed protocols such as IPsec and SSL provide such guarantees at line rate. What if such protocols are not available? We can implement simple authenticated channels within uBFT without signatures in the critical path, by augmenting messages with an HMAC. With BLAKE3, creating or verifying 256-bits HMACs takes ≈100 ns. As a result, we expect less than 2 µs of additional overhead on the fast path of uBFT.

What about uBFT's throughput? Any system can provide a throughput that is inverse of its latency. For uBFT, that amounts to ≈91 kops for small 32 B requests. uBFT doubles this figure, by exploiting the slack between the processing of events in a consensus slot to interleave two requests with minimal latency penalty. Throughput can be further optimized using well-known techniques, such as batching [31] and running parallel consensus instances on multiple cores [16], but we have not done so.

10 CONCLUSION

uBFT is the first BFT SMR system to achieve microsecond-scale latency, $2f$+1 replicas, practically bounded memory, and a small non-tailored trusted computing base. To do that, uBFT introduces Consistent Tail Broadcast to prevent equivocation, and a matching consensus algorithm. uBFT uses disaggregated memory as a trusted computing base, which our prototype implements using RDMA. We applied uBFT to three applications to show that BFT can be realistic in data centers even in latency-critical settings.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable comments, as well as our anonymous artifact evaluators for reviewing our prototype implementation. Finally, we thank our colleagues Jovan Komatovic and Pierre-Louis Roman for their feedback. This work has been supported in part by SNSF Early Postdoc.Mobility grant P2ELP2_195126.

11 CONSENSUS ALGORITHM

This section gives the pseudocode of uBFT's consensus. Due to space limitations, correctness arguments are given in the extended version of this paper [5]. Algorithm 2 describes uBFT's operation under a stable leader. Algorithm 3 describes view changes. Algorithm 4 describes how summaries let uBFT handle the gaps caused by CTBcast's tail-validity. Finally, Algorithm 5 gives the explicit requirements for messages to pass CTBcast's Byzantine checks.

Algorithm 2: Common Case (stable leader)

```
CTBcasts FIFO-deliver and block upon a Byzantine message

upon Init:
  view = 0
  next_slot = 0
  checkpoint = (app_state: Initial, open_slots: [0, 99])_Σ
  for each replica:
    state[replica] = {
      view = 0, seal_view = ⊥, new_view = ⊥,
      prepares: Map<slot, PREPARE> = {},
      commits: Map<slot, COMMIT> = {},
      checkpoint = (Initial, [0, 99])_Σ }

def Propose(req):
  wait (leader(view) == me and next_slot in checkpoint.
      ↪ open_slots and NEW_VIEW broadcast if view > 0)
  CTBcast-bcast <PREPARE, view, next_slot++, req>

upon CTBcast-dlvr <PREPARE, v, s, r> from p as P:
  state[p].prepares[s] = P
  if v != view or s ∉ checkpoint.open_slots: return
  TBcast-bcast <WILL_CERTIFY, v, s> # Fast path
  TBcast-bcast <CERTIFY, sign(P)> # Slow path

# Fast path
upon TBcast-dlvr <WILL_CERTIFY, v, s> from 2f+1:
  if v != view or s ∉ checkpoint.open_slots: return
  TBcast-bcast <WILL_COMMIT, v, s>

upon TBcast-dlvr <WILL_COMMIT, v, s> from 2f+1:
  if v != view or s ∉ checkpoint.open_slots: return
  trigger once Decide(s, state[leader(v)].prepares[s].req)

# Slow path
upon TBcast-dlvr <CERTIFY, <P., v, s, _>_σ> from f+1 as P_Σ:
  if v != view or s ∉ checkpoint.open_slots: return
  CTBcast-bcast <COMMIT, P_Σ>

upon CTBcast-dlvr <COMMIT, P_Σ> from p as C:
  state[p].commits[P_Σ.slot] = C
  if dlvred f+1 COMMIT with a matching PREPARE:
    trigger once Decide(P_Σ.slot, P_Σ.req)

# Checkpoints
after having decided on all checkpoint.open_slots:
  wait for all decided requests to be applied to the App
  next_cp = (App.Snapshot(), checkpoint.open_slots + 100)
  TBcast-bcast <CERTIFY_CHECKPOINT, sign(next_cp)>

upon TBcast-dlvr <CERTIFY_CHECKPOINT, c_σ> from f+1 as C_Σ:
  MaybeCheckpoint(C_Σ)

upon CTBcast-dlvr <CHECKPOINT, C_Σ> from p as CP:
  state[p].checkpoint = CP
  forget state[p].commits and prepares ∉ C_Σ.open_slots
  MaybeCheckpoint(C_Σ)

def MaybeCheckpoint(C_Σ):
  if C_Σ supersedes checkpoint:
    checkpoint = C_Σ
    App.BringUpToSpeed(checkpoint)
    TBcast-bcast <CHECKPOINT, checkpoint>
```

Algorithm 3: View Change

```
upon suspicion of leader(view): ChangeView()

def ChangeView():
  for each <WILL_COMMIT, v|_{v==view}, s> bcast via TBcast:
    wait to have broadcast a matching COMMIT or CHECKPOINT
  CTBcast-bcast <SEAL_VIEW, ++view>

upon CTBcast-dlvr <SEAL_VIEW, v> from p as SV:
  state[p].seal_view = SV
  state[p].view = v
  send_leader(v) <CRTFY_VC, v, sign((p, state[p]\new_view))>

upon dlvr f+1 matching <CRTFY_VC, v|_{v==view}, s_σ> about f+1
    ↪ replicas as C:
  if me != leader(view): return
  CTBcast-bcast <NEW_VIEW, C>
  MaybeCheckpoint(highest checkpoint in C)
  for s in checkpoint.open_slots:
    CTB-bcast <PREPARE, v, s, MustPropose(s, C)>
  next_slot = checkpoint.open_slots.last + 1

upon CTBcast-dlvr <NEW_VIEW, certificates> from p as NV:
  state[p].new_view = NV
  while view != NV.view + 1: ChangeView()

def MustPropose(slot, certificates):
  if slot > max open slot in certificates: return Any
  return latest committed req for slot in certificates or ⊥
```

Algorithm 4: CTBcast Summaries

```
after CTBcast-dlvr the message with id % tail == 0 from p:
  send_p <CERTIFY_SUMMARY, sign((p, id, state[p]))>

every tail invocations of CTBcast-bcast:
  block calls to CTBcast-bcast

upon dlvr <CERTIFY_SUMMARY, (me, id, _)_σ> from f+1 as S_Σ:
  TBcast-bcast <SUMMARY, S_Σ>
  unblock calls to CTBcast-bcast

upon TBcast-dlvr <SUMMARY, (p, id, history)_Σ>:
  when a gap is detected in the dlvry of CTBcast from p:
    if the latest message dlvred from p is lower than id:
      dlvr in order p's missed CTBcast messages in history
          ↪ without running the Byzantine checks (Alg. 5)
      continue dlvring p's CTBcast messages after id
```

Algorithm 5: CTBcast's Byzantine Checks

```
def valid <PREPARE, v, s, r> from p:
  state[p].view == v and leader(v) == p and
    s in state[p].checkpoint.open_slots and
    p never prepared slot s before in v and
    (v == 0 or (state[p].new_view != ⊥ and
      r == MustPropose(s, state[p].new_view)))

def valid <COMMIT, P_Σ> from p as C:
  P_Σ.slot in state[p].checkpoint.open_slots and
    P_Σ.view == state[p].view and
    state[p].commits[P_Σ.slot] != C

def valid <CHECKPOINT, C_Σ> from p:
  C_Σ supersedes state[p].checkpoint

def valid <SEAL_VIEW, v> from p:
  state[p].view < v

def valid <NEW_VIEW, certificates> from p:
  leader(state[p].view) == p and
    it is p's first non-CHECKPOINT message in this view and
    each certificate is about a different replica and
    each certificate is signed by f+1 different replicas and
    each certificate is about view state[p].view
```

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-Scalable Byzantine Fault-Tolerant Services. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct 2005), 59–74. https://doi.org/10.1145/1095809.1095817

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto, ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 409–418. https://doi.org/10.1145/3293611.3331601

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Berkeley, CA, USA, Article 34, 18 pages. https://www.usenix.org/conference/osdi20/presentation/aguilera

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2022. Source code for uBFT: Microsecond-scale BFT using Disaggregated Memory. https://doi.org/10.5281/zenodo.7330354

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2022. uBFT: Microsecond-scale BFT using Disaggregated Memory (Extended Version). https://doi.org/10.48550/ARXIV.2210.17174

[6] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. 2021. Frugal Byzantine Computing. In *35th International Symposium on Distributed Computing (DISC '21) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:19. https://doi.org/10.4230/LIPIcs.DISC.2021.3

[7] Marcos K. Aguilera and Michael Walfish. 2009. No Time for Asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (Monte Verità, Switzerland) *(HotOS '09)*. USENIX Association, Berkeley, CA, USA, 3. https://www.usenix.org/conference/hotos-xii/no-time-asynchrony

[8] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. 2008. Matrix Signatures: From MACs to Digital Signatures in Distributed Systems. In *Proceedings of the 22nd International Symposium on Distributed Computing* (Arcachon, France) *(DISC '08)*. Springer-Verlag, Berlin, Heidelberg, 16–31. https://doi.org/10.1007/978-3-540-87779-0_2

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[10] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack W. Stokes, Geoff Outhred, and Lechao Diwu. 2020. PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, Berkeley, CA, USA, 797–816. https://www.usenix.org/conference/nsdi20/presentation/arzani

[11] InfiniBand Trade Association. 2020. InfiniBand Architecture, General Specifications, Memory Placement Extensions. Retrieved November 11, 2022 from https://cw.infinibandta.org/document/dl/8594

[12] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS '13)*. IEEE Computer Society, USA, 297–306. https://doi.org/10.1109/ICDCS.2013.53

[13] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *2021 USENIX Annual Technical Conference (USENIX ATC '21)* (Virtual Event). USENIX Association, Berkeley, CA, USA, 65–79. https://www.usenix.org/conference/atc21/presentation/bailleu

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (Mar 2017), 48–54. https://doi.org/10.1145/3015146

[15] Motti Beck and Michael Kagan. 2011. 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* (San Francisco, CA, USA) *(DC-CaVES '11)*. International Teletraffic Congress, San Francisco, CA, USA, 9–15. https://dl.acm.org/doi/10.5555/2043535.2043537

[16] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 222–237. https://doi.org/10.1145/3064176.3064213

[17] Naama Ben-David, Benjamin Y. Chan, and Elaine Shi. 2022. Revisiting the Power of Non-Equivocation in Distributed Protocols. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (Salerno, Italy) *(PODC '22)*. Association for Computing Machinery, New York, NY, USA, 450–459. https://doi.org/10.1145/3519270.3538427

[18] Naama Ben-David and Kartik Nayak. 2021. Brief Announcement: Classifying Trusted Hardware via Unidirectional Communication. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event) *(PODC '21)*. Association for Computing Machinery, New York, NY, USA, 191–194. https://doi.org/10.1145/3465084.3467948

[19] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Atlanta, GA, USA) *(DSN '14)*. IEEE Computer Society, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

[20] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 645–650. https://www.usenix.org/conference/atc18/presentation/boucher

[21] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. https://doi.org/10.48550/ARXIV.1807.04938

[22] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer Publishing Company, New York, NY, USA. https://doi.org/10.1007/978-3-642-15260-3

[23] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.* 11, 11 (Jul 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[24] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. https://www.usenix.org/legacy/publications/library/proceedings/osdi99/castro.html

[25] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, WA, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 189–204. https://doi.org/10.1145/1294261.1294280

[26] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. 2012. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing* (Madeira, Portugal) *(PODC '12)*. Association for Computing Machinery, New York, NY, USA, 301–308. https://doi.org/10.1145/2332432.2332490

[27] Yann Collet. 2022. xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash

[28] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086

[29] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 779–793. https://doi.org/10.1145/3460120.3484821

[30] Ian Cutress. 2019. CXL Specification 1.0 Released: New Industry High-Speed Interconnect From Intel. Retrieved May 25, 2022 from https://www.anandtech.com/show/14068/cxl-specification-1-released-new-industry-high-speed-interconnect-from-intel

[31] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

[32] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. 2004. The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing* (St. John's, NL, Canada) *(PODC '04)*. Association for Computing Machinery, New York, NY, USA, 338–346. https://doi.org/10.1145/1011767.1011818

[33] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. https://doi.org/10.48550/ARXIV.2203.08989

[34] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, CA, USA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[35] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '16)*. USENIX Association, Berkeley, CA, USA, 45–59. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[36] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv.* 54, 6, Article 126 (Jul 2021), 36 pages. https://doi.org/10.1145/3456631

[37] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[38] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Portland, OR, USA) *(DSN '19)*. IEEE Computer Society, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[39] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC '22)* (Carlsbad, CA, USA). USENIX Association, Berkeley, CA, USA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[40] Tomasz Gromadzki and Jan Marian Michalski. 2019. Persistent Memory Replication Over Traditional RDMA Part 1: Understanding Remote Persistent Memory. Retrieved November 11, 2022 from https://www.intel.com/content/www/us/en/developer/articles/technical/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html

[41] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. 2022. uKharon: A Membership Service for Microsecond Applications. In *2022 USENIX Annual Technical Conference (USENIX ATC '22)* (Carlsbad, CA, USA). USENIX Association, Berkeley, CA, USA, 101–120. https://www.usenix.org/conference/atc22/presentation/guerraoui

[42] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. *ACM Trans. Storage* 14, 3, Article 23 (Oct 2018), 26 pages. https://doi.org/10.1145/3242086

[43] Divya Gupta, Lucas Perronne, and Sara Bouchenak. 2016. BFT-Bench: A Framework to Evaluate BFT Protocols. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 109–112. https://doi.org/10.1145/2851553.2858667

[44] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores That Don't Count. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, MI, USA) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3458336.3465297

[45] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/3102980.3103005

[46] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '10)*. USENIX Association, Berkeley, CA, USA, 11 pages. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[47] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. USENIX Association, Virtual Event, 239–256. https://www.usenix.org/conference/osdi21/presentation/ibanez

[48] Danga Interactive. 2022. Memcached. https://memcached.org/

[49] Hai Jin, Rajkumar Buyya, and Toni Cortes. 2002. An Introduction to the InfiniBand Architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (1st ed.). John Wiley and Sons, Inc., Hoboken, NJ, USA, 616–632. https://doi.org/10.1109/9780470544839

[50] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, IL, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[51] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/2168836.2168866

[52] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC '16)*. USENIX Association, Berkeley, CA, USA, 279–296. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

[53] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4, Article 7 (Jan 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[54] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (Jul 1982), 382–401. https://doi.org/10.1145/357172.357176

[55] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

[56] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2019. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. https://doi.org/10.48550/ARXIV.1910.09727

[57] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, GA, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[58] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting Failures in Distributed Systems with the FALCON Spy Network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2043556.2043583

[59] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '09)*. USENIX Association, Berkeley, CA, USA, 14 pages. https://www.usenix.org/conference/nsdi-09/trinc-small-trusted-hardware-large-distributed-systems

[60] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (Jan 2019), 139–151. https://doi.org/10.1109/TC.2018.2860009

[61] Isis Agora Lovecruft, Henry De Valence, and Aumasson. 2022. ed25519-dalek: Fast and efficient Rust implementation of ed25519 key generation, signing, and verification in Rust. https://github.com/dalek-cryptography/ed25519-dalek

[62] Mads Frederik Madsen and Søren Debois. 2020. On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (Virtual Event) *(PODC '20)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/3382734.3405731

[63] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, CA, USA) *(OSDI '08)*. USENIX Association, Berkeley, CA, USA, 369–384. https://www.usenix.org/conference/osdi-08/mencius-building-efficient-replicated-state-machines-wans

[64] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Trans. Dependable Secur. Comput.* 3, 3 (Jul 2006), 202–215. https://doi.org/10.1109/TDSC.2006.35

[65] Ines Messadi, Markus Horst Becker, Kai Bleeke, Leander Jehl, Sonia Ben Mokhtar, and Rüdiger Kapitza. 2022. SplitBFT: Improving Byzantine Fault Tolerance Safety Using Trusted Compartments. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 56–68. https://doi.org/10.1145/3528535.3531516

[66] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Portland, OR, USA) *(SIGMETRICS '15)*. Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/2745844.2745848

[67] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for

Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3278532.3278566

[68] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[69] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA, USA) *(USENIX ATC '13)*. USENIX Association, Berkeley, CA, USA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[70] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, PA, USA) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[71] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. Retrieved October 28, 2022 from https://bitcoin.org/bitcoin.pdf

[72] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD Failures in Datacenters: What, When and Why?. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Antibes Juan-les-Pins, France) *(SIGMETRICS '16)*. Association for Computing Machinery, New York, NY, USA, 407–408. https://doi.org/10.1145/2896377.2901489

[73] Eric Newhuis. 2022. Liquibook: Open source order matching engine. https://github.com/enewhuis/liquibook

[74] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. 2022. BLAKE3. https://github.com/BLAKE3-team/BLAKE3

[75] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done about It?. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Seattle, WA, USA) *(USITS '03)*. USENIX Association, Berkeley, CA, USA, 15 pages. https://www.usenix.org/conference/usits-03/why-do-internet-services-fail-and-what-can-be-done-about-it

[76] Red Hat. 2020. RHEL for Real Time Timestamping. Retrieved May 25, 2022 from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping

[77] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security '21)* (Virtual Event). USENIX Association, Berkeley, CA, USA, 4277–4292. https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger

[78] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. 2018. Towards Low-Latency Byzantine Agreement Protocols Using RDMA. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE Computer Society, Los Alamitos, CA, USA, 146–151. https://doi.org/10.1109/DSN-W.2018.00054

[79] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (Nov 1984), 277–288. https://doi.org/10.1145/357401.357402

[80] Salvatore Sanfilippo. 2022. Redis. https://github.com/redis/redis

[81] David Schneider. 2012. The microsecond market. *IEEE Spectrum* 49, 6 (2012), 66–81. https://doi.org/10.1109/MSPEC.2012.6203974

[82] Omid Shahmirzadi, Sergio Mena, and Andre Schiper. 2009. Relaxed Atomic Broadcast: State-Machine Replication Using Bounded Memory. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS '09)*. IEEE Computer Society, Niagara Falls, NY, USA, 3–11. https://doi.org/10.1109/SRDS.2009.25

[83] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '11)*. USENIX Association, Berkeley, CA, USA, 309–322. https://www.usenix.org/conference/nsdi11/sharing-data-center-network

[84] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA: Efficient NIC-Based Authentication and Encryption for Remote Direct Memory Access. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (Virtual Event) *(USENIX ATC '20)*. USENIX Association, Berkeley, CA, USA, Article 47, 14 pages. https://www.usenix.org/conference/atc20/presentation/taranov

[85] Mellanox Technologies. 2015. RDMA Aware Networks Programming User Manual. Rev 1.7. Retrieved May 25, 2022 from https://docs.nvidia.com/networking/spaces/viewspace.action?key=RDMAAwareProgrammingv17

[86] Mellanox Technologies. 2022. VMA: Linux user space library for network socket acceleration based on RDMA compatible network adaptors. https://github.com/Mellanox/libvma

[87] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference* (Virtual Event) *(USENIX ATC '20)*. USENIX Association, Berkeley, CA, USA, Article 3, 16 pages. https://www.usenix.org/conference/atc20/presentation/tsai

[88] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan 2013), 16–30. https://doi.org/10.1109/TC.2011.221

[89] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. https://doi.org/10.48550/ARXIV.2207.03027

[90] Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. 2019. The Synchronous Data Center. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 142–148. https://doi.org/10.1145/3317550.3321442

[91] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/945445.945470

[92] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto, ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[93] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)* (Carlsbad, CA, USA). USENIX Association, Berkeley, CA, USA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

[94] Danyang Zhuo, Qiao Zhang, Dan R. K. Ports, Arvind Krishnamurthy, and Thomas Anderson. 2014. Machine Fault Tolerance for Reliable Datacenter Systems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (Beijing, China) *(APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/2637166.2637235