The Impact of RDMA on Agreement

Marcos K. Aguilera
VMware
maguilera@vmware.com

Naama Ben-David
CMU
nbendavi@cs.cmu.edu

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Virendra Marathe
Oracle
virendra.marathe@oracle.com

Igor Zablotchi
EPFL
igor.zablotchi@epfl.ch

ABSTRACT

Remote Direct Memory Access (RDMA) is becoming widely available in data centers. This technology allows a process to directly read and write the memory of a remote host, with a mechanism to control access permissions. In this paper, we study the fundamental power of these capabilities. We consider the well-known problem of achieving consensus despite failures, and find that RDMA can improve the inherent trade-off in distributed computing between failure resilience and performance. Specifically, we show that RDMA allows algorithms that simultaneously achieve high resilience and high performance, while traditional algorithms had to choose one or another. With Byzantine failures, we give an algorithm that only requires $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and decides in two (network) delays in common executions. With crash failures, we give an algorithm that only requires $n \geq f_P + 1$ processes and also decides in two delays. Both algorithms tolerate a minority of memory failures inherent to RDMA, and they provide safety in asynchronous systems and liveness with standard additional assumptions.

1 INTRODUCTION

In recent years, a technology known as Remote Direct Memory Access (RDMA) has made its way into data centers, earning a spotlight in distributed systems research. RDMA provides the traditional send/receive communication primitives, but also allows a process to directly read/write remote memory. Research work shows that RDMA leads to some new and exciting distributed algorithms [3, 10, 25, 31, 45, 49].

RDMA provides a different interface from previous communication mechanisms, as it combines message-passing with shared-memory [3]. Furthermore, to safeguard the remote memory, RDMA provides *protection* mechanisms to grant and revoke access for reading and writing data. This mechanism is fine grained: an application can choose subsets of remote memory called *regions* to protect; it can choose whether a region can be read, written, or both; and it can choose individual processes to be given access, where different processes can have different accesses. Furthermore, protections are *dynamic*: they can be changed by the application over time. In this paper, we lay the groundwork for a theoretical understanding of these RDMA capabilities, and we show that they lead to distributed algorithms that are inherently more powerful than before.

While RDMA brings additional power, it also introduces some challenges. With RDMA, the remote memories are subject to failures that cause them to become unresponsive. This behavior differs from traditional shared memory, which is often assumed to be reliable[1]. In this paper, we show that the additional power of RDMA more than compensates for these challenges.

Our main contribution is to show that RDMA improves on the fundamental trade-off in distributed systems between failure resilience and performance—specifically, we show how a consensus protocol can use RDMA to achieve *both* high resilience and high performance, while traditional algorithms had to choose one or another. We illustrate this on the fundamental problem of achieving consensus and capture the above RDMA capabilities as an M&M model [3], in which processes can use both message-passing and shared-memory. We consider asynchronous systems and require safety in all executions and liveness under standard additional assumptions (e.g., partial synchrony). We measure resiliency by the number of failures an algorithm tolerates, and performance by the number of (network) delays in common-case executions. Failure resilience and performance depend on whether processes fail by crashing or by being Byzantine, so we consider both.

With Byzantine failures, we consider the consensus problem called weak Byzantine agreement, defined by Lamport [37]. We give an algorithm that (a) requires only $n \geq 2f_P + 1$ processes (where f_P is the maximum number of faulty processes) and (b) decides in two delays in the common case. With crash failures, we give the first algorithm for consensus that requires only $n \geq f_P + 1$ processes and decides in two delays in the common case. With both Byzantine or crash failures, our algorithms can also tolerate crashes of memory—only $m \geq 2f_M + 1$ memories are required, where f_M is the maximum number of faulty memories. Furthermore, with crash failures, we improve resilience further, to tolerate crashes of a minority of the combined set of memories and processes.

---

[1]There are a few studies of failure-prone memory, as we discuss in related work.

Our algorithms appear to violate known impossibility results: it is known that with message-passing, Byzantine agreement requires $n \geq 3f_P + 1$ even if the system is synchronous [44], while consensus with crash failures require $n \geq 2f_P + 1$ if the system is partially synchronous [27]. There is no contradiction: our algorithms rely on the power of RDMA, not available in other systems.

RDMA's power comes from two features: (1) simultaneous access to message-passing and shared-memory, and (2) dynamic permissions. Intuitively, shared-memory helps resilience, message-passing helps performance, and dynamic permissions help both.

To see how shared-memory helps resilience, consider the Disk Paxos algorithm [29], which uses shared-memory (disks) but no messages. Disk Paxos requires only $n \geq f_P + 1$ processes, matching the resilience of our algorithm. However, Disk Paxos is not as fast: it takes at least four delays. In fact, we show that no shared-memory consensus algorithm can decide in two delays (Section 6).

To see how message-passing helps performance, consider the Fast Paxos algorithm [39], which uses message-passing and no shared-memory. Fast Paxos decides in only two delays in common executions, but it requires $n \geq 2f_P + 1$ processes.

Of course, the challenge is achieving both high resilience and good performance in a single algorithm. This is where RDMA's dynamic permissions shine. Clearly, dynamic permissions improve resilience against Byzantine failures, by preventing a Byzantine process from overwriting memory and making it useless. More surprising, perhaps, is that dynamic permissions help performance, by providing an uncontended instantaneous guarantee: if each process revokes the write permission of other processes before writing to a register, then a process that writes successfully knows that it executed uncontended, without having to take additional steps (e.g., to read the register). We use this technique in our algorithms for both Byzantine and crash failures.

In summary, our contributions are as follows:

- We consider distributed systems with RDMA, and we propose a model that captures some of its key properties while accounting for failures of processes and memories, with support of dynamic permissions.
- We show that the shared-memory part of our RDMA improves resilience: our Byzantine agreement algorithm requires only $n \geq 2f_P + 1$ processes.
- We show that the shared-memory by itself does not permit consensus algorithms that decide in two steps in common executions.
- We show that with dynamic permissions, we can improve the performance of our Byzantine Agreement algorithm, to decide in two steps in common executions.
- We give similar results for the case of crash failures: decision in two steps while requiring only $n \geq f_P + 1$ processes.
- Our algorithms can tolerate the failure of memories, up to a minority of them.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. In Section 3 we formally define the RDMA-compliant M&M model that we use in the rest of the paper, and specify the agreement problems that we solve. We then proceed to present the main contributions of the paper. Section 4 presents our fast and resilient Byzantine agreement algorithm. In Section 5

we consider the special case of crash-only failures, and show an improvement of the algorithm and tolerance bounds for this setting. In Section 6 we briefly outline a lower bound that shows that the dynamic permissions of RDMA are necessary for achieving our results. Finally, in Section 7 we discuss the semantics of RDMA in practice, and how our model reflects these features.

Due to space limitations, most proofs have been omitted from this version, and appear in the full version of this paper [4].

2 RELATED WORK

RDMA. Many high-performance systems were recently proposed using RDMA, such as distributed key-value stores [25, 31], communication primitives [25, 32], and shared address spaces across clusters [25]. Kaminsky *et al.* [33] provides guidelines for designing systems using RDMA. RDMA has also been applied to solve consensus [10, 45, 49]. Our model shares similarities with DARE [45] and APUS [49], which modify queue-pair state at run time to prevent or allow access to memory regions, similar to our dynamic permissions. These systems perform better than TCP/IP-based solutions, by exploiting better raw performance of RDMA, without changing the fundamental communication complexity or failure-resilience of the consensus protocol. Similarly, Rüsch *et al.* [46] use RDMA as a replacement for TCP/IP in existing BFT protocols.

M&M. Message-and-memory (M&M) refers to a broad class of models that combine message-passing with shared-memory, introduced by Aguilera et al. in [3]. In that work, Aguilera et al. consider M&M models without memory permissions and failures, and show that such models lead to algorithms that are more robust to failures and asynchrony. In particular, they give a consensus algorithm that tolerates more crash failures than message-passing systems, but is more scalable than shared-memory systems, as well as a leader election algorithm that reduces the synchrony requirements. In this paper, our goal is to understand how memory permissions and failures in RDMA impact agreement.

Byzantine Fault Tolerance. Lamport, Shostak and Pease [40, 44] show that Byzantine agreement can be solved in synchronous systems iff $n \geq 3f_P + 1$. With unforgeable signatures, Byzantine agreement can be solved iff $n \geq 2f_P + 1$. In asynchronous systems subject to failures, consensus cannot be solved [28]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [11] or partial synchrony [18, 27]. Many Byzantine agreement algorithms focus on safety and implicitly use the additional assumptions for liveness. Even with signatures, asynchronous Byzantine agreement can be solved only if $n \geq 3f_P + 1$ [16].

It is well known that the resilience of Byzantine agreement varies depending on various model assumptions like synchrony, signatures, equivocation, and the exact variant of the problem to be solved. A system that has non-equivocation is one that can prevent a Byzantine process from sending different values to different processes. Table 1 summarizes some known results that are relevant to this paper.

Our Byzantine agreement results share similarities with results for shared memory. Malkhi *et al.* [41] and Alon *et al.* [5] show bounds on the resilience of strong and weak consensus in a model with reliable memory but Byzantine processes. They also provide consensus protocols, using read-write registers enhanced with

Work	Synchrony	Signatures	Non-Equiv	Strong Validity	Resiliency
[40]	✓	✓	✗	✓	$2f+1$
[40]	✓	✗	✗	✓	$3f+1$
[5, 41]	✗	✓	✓	✓	$3f+1$
[21]	✗	✓	✗	✗	$3f+1$
[21]	✗	✗	✓	✗	$3f+1$
[21]	✗	✓	✓	✗	$2f+1$
This paper	✗	✓	✗ (RDMA)	✗	$2f+1$

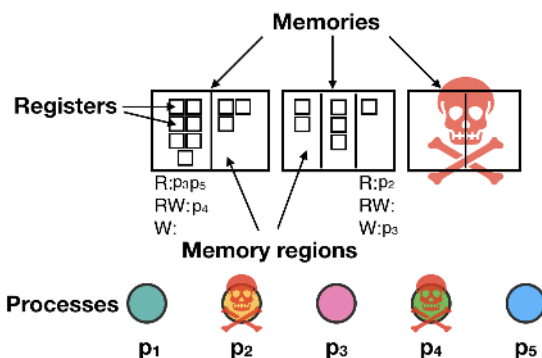**Table 1: Known fault tolerance results for Byzantine agreement.**

Figure 1: Our model with processes and memories, which may both fail. Processes can send messages to each other or access registers in the memories. Registers in a memory are grouped into memory regions that may overlap, but in our algorithms they do not. Each region has a permission indicating what processes can read, write, and read-write the registers in the region (shown for two regions).

sticky bits (write-once memory) and access control lists not unlike our permissions. Bessani *et al.* [12] propose an alternative to sticky bits and access control lists through Policy-Enforced Augmented Tuple Spaces. All these works handle Byzantine failures with powerful objects rather than registers. Bouzid *et al.* [14] show that $3f_P+1$ processes are necessary for strong Byzantine agreement with read-write registers.

Some prior work solves Byzantine agreement with $2f_P+1$ processes using specialized trusted components that Byzantine processes cannot control [19, 20, 22, 23, 34, 48]. Some schemes decide in two delays but require a large trusted component: a coordinator [19], reliable broadcast [23], or message ordering [34]. For us, permission checking in RDMA is a trusted component of sorts, but it is small and readily available.

At a high-level, our improved Byzantine fault tolerance is achieved by preventing equivocation by Byzantine processes, thereby effectively translating each Byzantine failure into a crash failure. Such translations from one type of failure into a less serious one have appeared extensively in the literature [9, 16, 21, 43]. Early work [9, 43] shows how to translate a crash tolerant algorithm into a Byzantine tolerant algorithm in the synchronous setting. Bracha [15] presents a similar translation for the asynchronous setting, in which $n \geq 3f_P+1$ processes are required to tolerate f_P Byzantine failures. Bracha's translation relies on the definition and implementation of a *reliable broadcast* primitive; in this paper we define and implement a similar, but weaker, broadcast primitive that can tolerate more failures due to the capabilities of RDMA.

Faulty memory. Afek *et al.* [2] and Jayanti *et al.* [30] study the problem of masking the benign failures of shared memory or objects. We use their ideas of replicating data across memories. Abraham *et al.* [1] considers honest processes but malicious memory.

Common-case executions. Many systems and algorithms tolerate adversarial scheduling but optimize for common-case executions without failures, asynchrony, contention, etc (e.g., [13, 24, 26, 35, 36, 39, 42]). None of these match both the resilience and performance of our algorithms. Some algorithms decide in one delay but require $n \geq 5f_P+1$ for Byzantine failures [47] or $n \geq 3f_P+1$ for crash failures [17, 24].

3 MODEL AND PRELIMINARIES

We consider a message-and-memory (M&M) model, which allows processes to use both message-passing and shared-memory [3]. The system has n processes $P = \{p_1, \ldots, p_n\}$ and m (shared) *memories*

$M = \{\mu_1, \ldots, \mu_m\}$. Processes communicate by accessing memories or sending messages. Throughout the paper, memory refers to the shared memories, not the local state of processes.

The system is asynchronous in that it can experience arbitrary delays. We expect algorithms to satisfy the safety properties of the problems we consider, under this asynchronous system. For liveness, we require additional standard assumptions, such as partial synchrony, randomization, or failure detection.

Memory permissions. Each memory consists of a set of *registers*. To control access, an algorithm groups those registers into a set of (possibly overlapping) *memory regions*, and then defines permissions for those memory regions. Formally, a memory region mr of a memory μ is a subset of the registers of μ. We often refer to mr without specifying the memory μ explicitly. Each memory region mr has a *permission*, which consists of three disjoint sets of processes R_{mr}, W_{mr}, RW_{mr} indicating whether each process can read, write, or read-write the registers in the region. We say that p has read permission on mr if $p \in R_{mr}$ or $p \in RW_{mr}$; we say that p has write permission on mr if $p \in W_{mr}$ or $p \in RW_{mr}$. In the special case when $R_{mr} = P \setminus \{p\}$, $W_{mr} = \emptyset$, $RW_{mr} = \{p\}$, we say that mr is a Single-Writer Multi-Reader (SWMR) region—registers in mr correspond to the traditional notion of SWMR registers. Note that a register may belong to several regions, and a process may have access to the register on one region but not another—this models the existing RDMA behavior. Intuitively, when reading or writing data, a process specifies the region and the register, and the system uses the region to determine if access is allowed (we make this precise below).

Permission change. An algorithm indicates an initial permission for each memory region mr. Subsequently, the algorithm may wish to change the permission of mr during execution. For that, processes can invoke an operation *changePermission(mr, new_perm)*, where *new_perm* is a triple (R, W, RW). This operation returns no results and it is intended to modify R_{mr}, W_{mr}, RW_{mr} to R, W, RW.

To tolerate Byzantine processes, an algorithm can restrict processes from changing permissions. For that, the algorithm specifies a function *legalChange*(*p*, *mr*, *old_perm*, *new_perm*) which returns a boolean indicating whether process *p* can change the permission of *mr* to *new_perm* when the current permissions are *old_perm*. More precisely, when *changePermission* is invoked, the system evaluates *legalChange* to determine whether *changePermission* takes effect or becomes a no-op. When *legalChange* always returns false, we say that the *permissions are static*; otherwise, the *permissions are dynamic*.

Accessing memories. Processes access the memories via operations *write*(*mr*, *r*, *v*) and *read*(*mr*, *r*) for memory region *mr*, register *r*, and value *v*. A *write*(*mr*, *r*, *v*) by process *p* changes register *r* to *v* and returns *ack* if $r \in mr$ and *p* has write permission on *mr*; otherwise, the operation returns *nak*. A *read*(*mr*, *r*) by process *p* returns the last value successfully written to *r* if $r \in mr$ and *p* has read permission on *mr*; otherwise, the operation returns *nak*. In our algorithms, a register belongs to exactly one region, so we omit the *mr* parameter from write and read operations.

Sending messages. Processes can also communicate by sending messages over a set of directed links. We assume messages are unique. If there is a link from process *p* to process *q*, then *p* can send messages to *q*. Links satisfy two properties: *integrity* and *no-loss*. Given two correct processes *p* and *q*, integrity requires that a message *m* be received by *q* from *p* at most once and only if *m* was previously sent by *p* to *q*. No-loss requires that a message *m* sent from *p* to *q* be eventually received by *q*. In our algorithms, we typically assume a fully connected network so that every pair of correct processes can communicate. We also consider the special case when there are no links (see below).

Executions and steps. An execution is as a sequence of process steps. In each step, a process does the following, according to its local state: (1) sends a message or invokes an operation on a memory (read, write, or changePermission), (2) tries to receive a message or a response from an outstanding operation, and (3) changes local state. We require a process to have at most one outstanding operation on each memory.

Failures. A memory *m* may fail by crashing, which causes subsequent operations on its registers to hang without returning a response. Because the system is asynchronous, a process cannot differentiate a crashed memory from a slow one. We assume there is an upper bound f_M on the maximum number of memories that may crash. Processes may fail by crashing or becoming Byzantine. If a process crashes, it stops taking steps forever. If a process becomes Byzantine, it can deviate arbitrarily from the algorithm. However, that process cannot operate on memories without the required permission. We assume there is an upper bound f_P on the maximum number of processes that may be faulty. Where the context is clear, we omit the *P* and *M* subscripts from the number of failures, *f*.

Signatures. Our algorithms assume unforgeable signatures: there are primitives *sign*(*v*) and *sValid*(*p*, *v*) which, respectively, signs a value *v* and determines if *v* is signed by process *p*.

Messages and disks. The model defined above includes two common models as special cases. In the *message-passing* model, there are no memories (*m* = 0), so processes can communicate only by sending messages. In the *disk* model [29], there are no links, so processes can communicate only via memories; moreover, each memory has a single region which always permits all processes to read and write all registers.

Consensus

In the consensus problem, processes propose an initial value and must make an irrevocable decision on a value. With crash failures, we require the following properties:
- **Uniform Agreement.** If processes *p* and *q* decide v_p and v_q, then $v_p = v_q$.
- **Validity.** If some process decides *v*, then *v* is the initial value proposed by some process.
- **Termination.** Eventually all correct processes decide.

We expect Agreement and Validity to hold in an asynchronous system, while Termination requires standard additional assumptions (partial synchrony, randomization, failure detection, etc). With Byzantine failures, we change these definitions so the problem can be solved. We consider weak Byzantine agreement [37], with the following properties:
- **Agreement.** If correct processes *p* and *q* decide v_p and v_q, then $v_p = v_q$.
- **Validity.** With no faulty processes, if some process decides *v*, then *v* is the input of some process.
- **Termination.** Eventually all correct processes decide.

Complexity of algorithms. We are interested in the performance of algorithms in *common-case executions*, when the system is synchronous and there are no failures. In those cases, we measure performance using the notion of *delays*, which extends message-delays to our model. Under this metric, computations are instantaneous, each message takes one delay, and each memory operation takes two delays. Intuitively, a delay represents the time incurred by the network to transmit a message; a memory operation takes two delays because its hardware implementation requires a round trip. We say that a consensus protocol is *k-deciding* if, in common-case executions, some process decides in *k* delays.

4 BYZANTINE FAILURES

We now consider Byzantine failures and give a 2-deciding algorithm for weak Byzantine agreement with $n \geq 2f_P + 1$ processes and $m \geq 2f_M + 1$ memories. The algorithm consists of the composition of two sub-algorithms: a slow one that always works, and a fast one that gives up under hard conditions.

The first sub-algorithm, called *Robust Backup*, is developed in two steps. We first implement a primitive called *non-equivocating broadcast*, which prevents Byzantine processes from sending different values to different processes. Then, we use the framework of Clement *et al.* [21] combined with this primitive to convert a message-passing consensus algorithm that tolerates crash failures into a consensus algorithm that tolerates Byzantine failures. This yields Robust Backup.[2] It uses only static permissions and assumes memories are split into SWMR regions. Therefore, this sub-algorithm works in the traditional shared-memory model with SWMR registers, and it may be of independent interest.

---

[2]The attentive reader may wonder why at this point we have not achieved a 2-deciding algorithm already: if we apply Clement *et al.* [21] to a 2-deciding crash-tolerant algorithm (such as Fast Paxos [13]), will the result not be a 2-deciding Byzantine-tolerant algorithm? The answer is no, because Clement *et al.* needs non-equivocated broadcast, which incurs at least 6 delays.

The second sub-algorithm is called *Cheap Quorum*. It uses dynamic permissions to decide in two delays using one signature in common executions. However, the sub-algorithm gives up if the system is not synchronous or there are Byzantine failures.

Finally, we combine both sub-algorithms using ideas from the Abstract framework of Aublin et al. [8]. More precisely, we start by running Cheap Quorum; if it aborts, we run Robust Backup. There is a subtlety: for this idea to work, Robust Backup must decide on a value v if Cheap Quorum decided v previously. To do that, Robust Backup decides on a *preferred value* if at least $f+1$ processes have this value as input. To do so, we use the classic crash-tolerant Paxos algorithm (run under the Robust Backup algorithm to ensure Byzantine tolerance) but with an initial set-up phase that ensures this safe decision. We call the protocol *Preferential Paxos*.

4.1 The Robust Backup Sub-Algorithm

We develop Robust Backup using the construction by Clement *et al.* [21], which we now explain. Clement *et al.* show how to transform a message-passing algorithm $\mathcal{A}$ that tolerates f_P crash failures into a message-passing algorithm that tolerates f_P Byzantine failures in a system where $n \geq 2f_P + 1$ processes, assuming unforgeable signatures and a non-equivocation mechanism. They do so by implementing trusted message-passing primitives, *T-send* and *T-receive*, using non-equivocation and signature verification on every message. Processes include their full history with each message, and then verify locally whether a received message is consistent with the protocol. This restricts Byzantine behavior to crash failures.

To apply this construction in our model, we show that our model can implement non-equivocation and message passing. We first show that shared-memory with SWMR registers (and no memory failures) can implement these primitives, and then show how our model can implement shared-memory with SWMR registers.

Consider a shared-memory system. While Clement *et al.* receive and verify a message separately, we combine the two steps into one primitive called *non-equivocating broadcast*.

DEFINITION 1. *Non-equivocating broadcast is defined in terms of two primitives, broadcast(k, m) and deliver(k, m, q). When a process p invokes broadcast(k, m) we say that p broadcasts (k, m). When a process p invokes deliver(k, m, q) we say that p delivers (k, m) from q. Each correct process p must invoke broadcast$(k, *)$ with k one higher than p's previous invocation (and first invocation with k=1). The following holds:*

(1) *If a correct process p broadcasts (k, m), then all correct processes eventually deliver (k, m) from p.*
(2) *If p and q are correct processes, p delivers (k, m) from r, and q delivers (k, m') from r, then m=m'.*
(3) *If a correct process delivers (k, m) from p, p must have broadcast (k, m).*

A correct implementation of non-equivocating broadcast replaces the send and receive primitives with broadcast and deliver respectively in Clement *et al.*'s implementation of T-send and T-receive.

We now show how to implement non-equivocating broadcast in shared-memory. The idea of the algorithm is that before delivering a message (k, m) from q, each process p checks that no other process

Algorithm 2: Non-Equivocating Broadcast

```
1   SWMR slots[n,M,n]; initialized to ⊥. slots[p] is array of SWMR(
          ↪ p) registers.

3   Code for process p to broadcast (k,m)
4     write(slots[p,j,p], sign((k,m)));

6   Code for process p
7     Last[n] is local array with last k delivered from each process
8     while true {
9       for q in Π {
10        try_deliver(q); }  }

12    try_deliver(q) {
13      k = Last[q]
14      val = (key,msg) = read(slots[q,k,q]);
15      if (val == ⊥ || !sValid(p, val) || key≠k)
16        return; //q hasn't written anything or is Byzantine. Will
                ↪ retry later.
17      Write(slots[p,k,q], val);
18      for i in Π {
19        otherV = (otherK, otherM) = read(slots[i,k,q]);
20        if (otherV ≠ val && otherV ≠ ⊥ && sValid(q, otherV) &&
                ↪ otherK == key)
21          return;  } //q is Byzantine; no delivery.
22      deliver(k,msg, q)
23      Last[q] += 1 }
```

saw a different value from q. More specifically, each process p has n memory slots per sequence number, that only p can write to, but all processes can read from. These slots are initialized to $\perp$, and p uses them to write the values that it has seen. To broadcast its k-th message, p simply writes a signed version of the message in slot (k, p) of its memory. To deliver a message m from process q with sequence number k, process p does three things: (1) p reads slot (q, k) from q's memory. If p reads $\perp$ from q's (k, q) slot, then q has not yet sent any message with sequence number k; p retries at a later time. If p reads a value that is not signed by q, it also restarts, pretending that it did not see any value. (2) Otherwise, if p read some signed value m from q's (k, q) slot, p writes m into slot (k, q) in its own memory, and (3) p reads slot (k, q) in every other process's memory. If, for every other process r, p reads either m or $\perp$ in (k, q) in r's memory, then p delivers q's message. Otherwise, since other processes cannot forge q's signature, this means that q has tried to equivocate, as some other process saw a different value when it read from q. p handles this case by ignoring the value from q.

LEMMA 4.1. *Non-equivocating broadcast is implementable in shared-memory with SWMR regular registers.*

The result of Clement et al. [21] and Lemma 4.1 immediately imply the following result.

THEOREM 4.2. *There exists an algorithm for weak Byzantine agreement in a shared-memory system with SWMR regular registers, signatures, and up to f_P process crashes where $n \geq 2f_P + 1$.*

In particular, we can implement weak Byzantine agreement by taking any correct consensus algorithm $\mathcal{A}$ for the classic crash-only message passing model, and replacing all its sends and receives by non-equivocating broadcast and deliver (respectively)

that also attach a process's entire execution history to each message. We call this method of communication *trusted sends and receives*, or simply T-send and T-receive primitives. Clement et al. [21] show that implementing such T-send and T-receive primitives with non-equivocation and signatures yields a Byzantine-tolerant replacement for classic sends and receives. In the full version of this paper, we present a pseudocode for T-send and T-receive with non-equivocating broadcast.

Non-equivocation in our model. To convert the above algorithm to our model, where memory may fail, we use the ideas in [2, 7, 30] to implement failure-free SWMR regular registers from the fail-prone memory, and then run weak Byzantine agreement using those regular registers. To implement an SWMR register, a process writes or reads all memories, and waits for a majority to respond. When reading, if p sees exactly one distinct non-$\perp$ value v across the memories, it returns v; otherwise, it returns $\perp$.

DEFINITION 2. *Let $\mathcal{A}$ be a message-passing algorithm.* Robust Backup($\mathcal{A}$) *is the algorithm $\mathcal{A}$ in which all send and receive operations are replaced by T-send and T-receive operations (respectively) implemented with non-equivocating broadcast.*

From the result of Clement et al. [21], Lemma 4.1, and the above handling of memory failures, it is easy to see that with $\mathcal{A}$ being a correct consensus algorithm for the crash-only setting, Robust Backup($\mathcal{A}$) solves weak Byzantine agreement with the desired fault tolerance in our dynamic permission M&M model. This is summarized in the follow theorem.

THEOREM 4.3. *There exists an algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

4.2 The Cheap Quorum Sub-Algorithm

We now give an algorithm that decides in two delays in common executions in which the system is synchronous and there are no failures. It requires only one signature for a fast decision, whereas the best prior algorithm requires $6f_P + 2$ signatures and $n \geq 3f_P + 1$ [8]. Our algorithm, called Cheap Quorum, is not in itself a complete consensus algorithm; it may abort in some executions. If Cheap Quorum aborts, it outputs an *abort value*, which is used to initialize the Robust Backup so that their composition preserves weak Byzantine agreement. This composition is inspired by the Abstract framework of Aublin *et al.* [8].

The algorithm has a special process ℓ, say $\ell = p_1$, which serves both as a *leader* and a *follower*. Other processes act only as *followers*. The memory is partitioned into $n + 1$ regions denoted *Region*[p] for each $p \in \Pi$, plus an extra one for p_1, *Region*[ℓ] in which it proposes a value. Initially, *Region*[p] is a regular SWMR region where p is the writer. Unlike in Algorithm 2, some of the permissions are dynamic; processes may remove p_1's write permission to *Region*[ℓ] (i.e., the *legalChange* function returns false to any permission change requests, except for ones revoking p_1's permission to write on *Region*[ℓ]).

Processes initially execute under a *normal* mode in common-case executions, but may switch to *panic* mode if they intend to abort, as in [8]. The pseudo-code of the normal mode is in Algorithm 3.

Algorithm 3: Cheap Quorum normal operation—code for process p

```
1   Leader code
2   propose(v) {
3       sign(v);
4       status = Value[ℓ].write(v);
5       if (status == nak) Panic_mode();
6       else decide(v); }
7
8   Follower code
9   propose(w){
10      do {v = read(Value[ℓ]);
11          for all q ∈ Π do pan[q] = read(Panic[q]);
12      } until (v ≠ ⊥ || pan[q] == true for some q || timeout);
13      if (v ≠ ⊥ && sValid(p1,v)) {
14          sign(v);
15          write(Value[p],v);
16          do {for all q ∈ Π do val[q] = read(Value[q]);
17              if |{q : val[q] == v}| ≥ n then {
18                  Proof[p].write(sign(val[1..n]));
19                  for all q ∈ Π do prf[q] = read(Proof[q]);
20                  if |{q : verifyProof(prf[q]) == true}| ≥ n {
                         ↪ decide(v); exit; } }
21              for all q ∈ Π do pan[q] = read(Panic[q]);
22          } until (pan[q] == true for some q || timeout); }
23      Panic_mode();}
```

Algorithm 4: Cheap Quorum panic mode—code for process p

```
1   panic_mode(){
2       Panic[p] = true;
3       changePermission(Region[ℓ], R: Π, W: {}, RW: {});
             ↪    // remove write permission
4       v = read(Value[p]);
5       prf = read(Proof[p]);
6       if (v ≠ ⊥){  Abort with ⟨v, prf⟩; return; }
7       LVal = read(Value[ℓ]);
8       if (LVal ≠ ⊥) {Abort with ⟨LVal, ⊥⟩; return;}
9       Abort with ⟨myInput, ⊥⟩; }
```

Region[p] contains three registers *Value*[p], *Panic*[p], *Proof*[p] initially set to $\perp$, *false*, $\perp$. To propose v, the leader p_1 signs v and writes it to *Value*[ℓ]. If the write is successful (it may fail because its write permission was removed), then p_1 decides v; otherwise p_1 calls *Panic_mode*(). Note that all processes, including p_1, continue their execution after deciding. However, p_1 never decides again if it decided as the leader. A follower q checks if p_1 wrote to *Value*[ℓ] and, if so, whether the value is properly signed. If so, q signs v, writes it to *Value*[q], and waits for other processes to write the same value to *Value*[$*$]. If q sees $2f + 1$ copies of v signed by different processes, q assembles these copies in a *unanimity proof*, which it signs and writes to *Proof*[q]. q then waits for $2f + 1$ unanimity proofs for v to appear in *Proof*[$*$], and checks that they are valid, in which case q decides v. This waiting continues until a timeout expires[3], at which time q calls *Panic_mode*(). In *Panic_mode*(), a process p sets *Panic*[p] to *true* to tell other processes it is panicking; other processes periodically check to see if they should panic too. p then removes write permission from *Region*[ℓ], and decides on a value to abort: either *Value*[p] if it is non-$\perp$, *Value*[ℓ] if it is non-$\perp$,

---

[3]The timeout is chosen to be an upper bound on the communication, processing and computation delays in the common case.
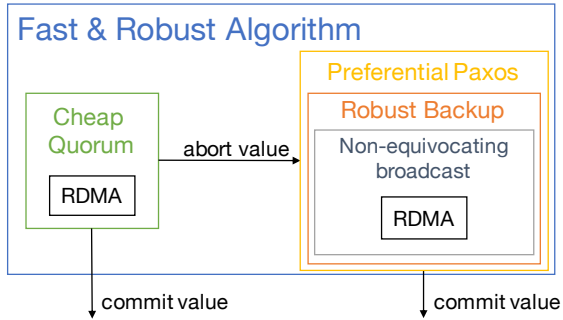
Figure 5: Interactions of the components of the Fast & Robust Algorithm.

or p's input value. If p has a unanimity proof in *Proof*[p], it adds it to the abort value.

In the full version of this paper [4], we prove the correctness of Cheap Quorum, and in particular we show the following two important agreement properties:

LEMMA 4.4 (CHEAP QUORUM DECISION AGREEMENT). *Let p and q be correct processes. If p decides v_1 while q decides v_2, then $v_1 = v_2$.*

LEMMA 4.5 (CHEAP QUORUM ABORT AGREEMENT). *Let p and q be correct processes (possibly identical). If p decides v in Cheap Quorum while q aborts from Cheap Quorum, then v will be q's abort value. Furthermore, if p is a follower, q's abort proof is a correct unanimity proof.*

The above construction assumes a fail-free memory with regular registers, but we can extend it to tolerate memory failures using the approach of Section 4.1, noting that each register has a single writer process.

4.3 Putting it Together: the Fast & Robust Algorithm

The final algorithm, called Fast & Robust, combines Cheap Quorum (§4.2) and Robust Backup (§4.1), as we now explain. Recall that Robust Backup is parameterized by a message-passing consensus algorithm $\mathcal{A}$ that tolerates crash-failures. $\mathcal{A}$ can be any such algorithm (e.g., Paxos).

Roughly, in Fast & Robust, we run Cheap Quorum and, if it aborts, we use a process's abort value as its input value to Robust Backup. However, we must carefully glue the two algorithms together to ensure that if some correct process decided v in Cheap Quorum, then v is the only value that can be decided in Robust Backup.

For this purpose, we propose a simple wrapper for Robust Backup, called *Preferential Paxos*. Preferential Paxos first runs a set-up phase, in which processes may adopt new values, and then runs Robust Backup with the new values. More specifically, there are some *preferred* input values $v_1 \ldots v_k$, ordered by priority. We guarantee that every process adopts one of the top $f + 1$ priority inputs. In particular, this means that if a majority of processes get the highest priority value, v_1, as input, then v_1 is guaranteed to be the decision value. The set-up phase is simple; all processes send each other their input values. Each process p waits to receive $n - f$ such messages, and adopts the value with the highest priority that it sees.

This is the value that p uses as its input to Paxos. The pseudocode for Preferential Paxos is given in the full version of our paper [4], where we also prove the following lemma about Preferential Paxos:

LEMMA 4.6 (PREFERENTIAL PAXOS PRIORITY DECISION). *Preferential Paxos implements weak Byzantine agreement with $n \geq 2f_P + 1$ processes. Furthermore, let $v_1, \ldots, v_n$ be the input values of an instance C of Preferential Paxos, ordered by priority. The decision value of correct processes is always one of $v_1, \ldots, v_{f+1}$.*

We can now describe Fast & Robust in detail. We start executing Cheap Quorum. If Cheap Quorum aborts, we execute Preferential Paxos, with each process receiving its abort value from Cheap Quorum as its input value to Preferential Paxos. We define the priorities of inputs to Preferential Paxos as follows.

DEFINITION 3 (INPUT PRIORITIES FOR PREFERENTIAL PAXOS). *The input values for Preferential Paxos as it is used in Fast & Robust are split into three sets (here, p_1 is the leader of Cheap Quorum):*

- *$T = \{v \mid v$ contains a correct unanimity proof $\}$*
- *$M = \{v \mid v \notin T \wedge v$ contains the signature of $p_1\}$*
- *$B = \{v \mid v \notin T \wedge v \notin M\}$*

The priority order of the input values is such that for all values $v_T \in T$, $v_M \in M$, and $v_B \in B$, $priority(v_T) > priority(v_M) > priority(v_B)$.

Figure 5 shows how the various algorithms presented in this section come together to form the Fast & Robust algorithm. In the full version, we show that Fast & Robust is correct, with the following key lemma:

LEMMA 4.7 (COMPOSITION LEMMA). *If some correct process decides a value v in Cheap Quorum before an abort, then v is the only value that can be decided in Preferential Paxos with priorities as defined in Definition 3.*

THEOREM 4.8. *There exists a 2-deciding algorithm for Weak Byzantine Agreement in a message-and-memory model with up to f_P Byzantine processes and f_M memory crashes, where $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$.*

5 CRASH FAILURES

We now restrict ourselves to crash failures of processes and memories. Clearly, we can use the algorithms of Section 4 in this setting, to obtain a 2-deciding consensus algorithm with $n \geq 2f_P + 1$ and $m \geq 2f_M + 1$. However, this is overkill since those algorithms use sophisticated mechanisms (signatures, non-equivocation) to guard against Byzantine behavior. With only crash failures, we now show it is possible to retain the efficiency of a 2-deciding algorithm while improving resiliency. In Section 5.1, we first give a 2-deciding algorithm that allows the crash of all but one process ($n \geq f_P + 1$) and a minority of memories ($m \geq 2f_M + 1$). In Section 5.2, we improve resilience further by giving a 2-deciding algorithm that tolerates crashes of a minority of the combined set of memories and processes.

5.1 Protected Memory Paxos

Our starting point is the Disk Paxos algorithm [29], which works in a system with processes and memories where $n \geq f_P + 1$ and $m \geq$

Algorithm 6: Protected Memory Paxos—code for process p

```
1   Regions: for i=1..m, Region[i] is all of memory i,
2                     with initial permission (R:
                      ↪ Π − {p₁}, W: ∅, RW: {p₁} )
3   Registers: for i=1..m, p ∈ Π,
4                 slot[i,p]: tuple (minProp,accProp, value)
                  ↪ in Region[i] // in memory i
5   Ω: failure detector that returns current leader

7   propose(v) {
8     repeat forever {
9       wait until Ω == p; // wait to become leader
10      propNr = choose number higher than any proposal
                ↪ numbers seen before;
11      if (p ≠ p₁){
12          pfor i=1..m {  // for each memory i
13              changePermission(Region[i], R:{Π-{p}},W:
                    ↪ ∅, RW:{p}); //get write perm
14              write1Success[i] = write(slot[i,p], {
                    ↪ propNr, ⊥, ⊥})
15              for all q ∈ Π do localInfo[i,q] = read(
                    ↪ slot[i,q])  }
16          wait for completion of m - fM iterations of
                    ↪ pfor loop
17          if (!write1Success[i] for some i) then
                    ↪ continue;
18          if (localInfo[i,q].minProp > propNr for some
                    ↪ i,q) continue;
19          if (localInfo[i,q].value = ⊥ for all i,q)
                    ↪ myValue = v
20          else myValue = localInfo[i,q].value where i,q
                    ↪   maximizes localInfo[i,q].accProp }
21      pfor i=1..m do write2Success[i] = write(slot[i,p
                ↪ ], {propNr, propNr, myValue})
22      wait for completion of m - fM iterations of pfor
                ↪ loop
23      if !write2Success[j] for some j then continue;
24      decide myValue   }}
```

$2f_M + 1$. This is our resiliency goal, but Disk Paxos takes four delays in common executes. Our new algorithm, called Protected Memory Paxos, removes two delays; it retains the structure of Disk Paxos but uses permissions to skip steps. Initially some fixed leader $\ell = np_1$ has exclusive write permission to all memories; if another process becomes leader, it takes the exclusive permission. Having exclusive permission permits a leader ℓ to optimize execution, because ℓ can do two things simultaneously: (1) write its consensus proposal and (2) determine whether another leader took over. Specifically, if ℓ succeeds in (1), it knows no leader ℓ' took over because ℓ' would have taken the permission. Thus ℓ avoids the last read in Disk Paxos, saving two delays. Of course, care must be taken to implement this without violating safety.

The pseudocode of Protected Memory Paxos is in Algorithm 6. Each memory has one memory region, and at any time exactly one process can write to a region. Each memory i holds a register $slot[i, p]$ for each process p. Intuitively, $slot[i, p]$ is intended for p to write, but p may not have write permission to do that if it is not the leader—in that case, no process writes $slot[i, p]$.

When a process p becomes leader, it executes a special phase (the first leader p_1 can skip this phase), where p acquires exclusive write permission for a majority of memories, writes a new proposal number in its slot in a majority of memories, and then reads all slots in a majority of memories. If any of p's writes fail or p finds a proposal with a higher proposal number, then p gives up. Otherwise,

p adopts the value with highest proposal number. In the next phase, p writes its value to its slot in a majority of memories. If a write fails, p gives up (then the current leader restarts the algorithm). If p succeeds, this is where we optimize time: p can simply decide, whereas Disk Paxos must read the memories again.

The code ensures that some correct process eventually decides, but it is easy to extend it so all correct processes decide [18]. Also, the code shows one instance of consensus, with p_1 as initial leader. With many consensus instances, the leader terminates one instance and becomes the default leader in the next.

THEOREM 5.1. *Consider a message-and-memory model with up to* f_P *process crashes and* f_M *memory crashes, where* $n \geq f_P + 1$ *and* $m \geq 2f_M + 1$. *There exists a 2-deciding algorithm for consensus.*

5.2 Aligned Paxos

We now further enhance the failure resilience. We show that memories and processes are equivalent *agents*, in that it suffices for a majority of the agents (processes and memories together) to remain alive to solve consensus. Our new algorithm, *Aligned Paxos*, achieves this resiliency. To do so, the algorithm relies on the ability to use both the messages and the memories in our model; permissions are not needed. The key idea is to align a message-passing algorithm and a memory-based algorithm to use any majority of agents. We align Paxos [38] and Protected Memory Paxos so that their decisions are coordinated. More specifically, Protected Memory Paxos and Paxos have two phases. To align these algorithms, we factor out their differences and replace their steps with an abstraction that is implemented differently for each algorithm. The result is our *Aligned Paxos* algorithm, which has two phases, each with three steps: *communicate*, *hear back*, and *analyze*. Each step treats processes and memories separately, and translates the results of operations on different agents to a common language. We implement the steps using their analogues in Paxos and Protected Memory Paxos[4]. The pseudocode of Aligned Paxos is given in the full version of our paper [4].

6 DYNAMIC PERMISSIONS ARE NECESSARY FOR EFFICIENT CONSENSUS

In §5.1, we showed how dynamic permissions can improve the performance of Disk Paxos. Are dynamic permissions necessary? We prove that with shared memory (or disks) alone, one cannot achieve 2-deciding consensus, even if the memory never fails, it has static permissions, processes may only fail by crashing, and the system is partially synchronous in the sense that eventually there is a known upper bound on the time it takes a correct process to take a step [27]. This result applies a fortiori to the Disk Paxos model [29].

THEOREM 6.1. *Consider a partially synchronous shared-memory model with registers, where registers can have arbitrary static permissions, memory never fails, and at most one processes may fail by crashing. No consensus algorithm is 2-deciding.*

---

[4]We believe other implementations are possible. For example, replacing the Protected Memory Paxos implementation for memories with the Disk Paxos implementation yields an algorithm that does not use permissions.

Proof. Assume by contradiction that A is an algorithm in the stated model that is 2-deciding. That is, there is some execution E of A in which some process p decides a value v with 2 delays. We denote by R and W the set of objects which p reads and writes in E respectively. Note that since p decides in 2 delays in E, R and W must be disjoint, by the definition of operation delay and the fact that a process has at most one outstanding operation per object. Furthermore, p must issue all of its read and writes without waiting for the response of any operation.

Consider an execution E' in which p reads from the same set R of objects and writes the same values as in E to the same set W of objects. All of the read operations that p issues return by some time t_0, but the write operations of p are delayed for a long time. Another process p' begins its proposal of a value $v' \neq v$ after t_0. Since no process other than p' writes to any objects, E' is indistinguishable to p' from an execution in which it runs alone. Since A is a correct consensus algorithm that terminates if there is no contention, p' must eventually decide value v'. Let t' be the time at which p' decides. All of p's write operations terminate and are linearized in E' after time t'. Execution E' is indistinguishable to p from execution E, in which it ran alone. Therefore, p decides $v \neq v'$, violating agreement. □

Theorem 6.1, together with the Fast Paxos algorithm of Lamport [39], shows that an atomic read-write shared memory model is strictly weaker than the message passing model in its ability to solve consensus quickly. This result may be of independent interest, since often the classic shared memory and message passing models are seen as equivalent, because of the seminal computational equivalence result of Attiya, Bar-Noy, and Dolev [7]. Interestingly, it is known that shared memory can tolerante more failures when solving consensus (with randomization or partial synchrony) [6, 16], and therefore it seems that perhaps shared memory is strictly stronger than message passing for solving consensus. However, our result shows that there are aspects in which message passing is stronger than shared memory. In particular, message passing can solve consensus faster than shared memory in well-behaved executions.

7 RDMA IN PRACTICE

Our model is meant to reflect capabilities of RDMA, while providing a clean abstraction to reason about. We now give an overview of how RDMA works, and how features of our model can be implemented using RDMA.

RDMA enables a remote process to access local memory directly through the network interface card (NIC), without involving the CPU. For a piece of local memory to be accessible to a remote process p, the CPU has to *register* that memory region and associate it with the appropriate connection (called *Queue Pair*) for p. The association of a registered memory region and a queue pair is done indirectly through a *protection domain*: both memory regions and queue pairs are associated with a protection domain, and a queue pair q can be used to access a memory region r if q and r and in the same protection domain. The CPU must also specify what access level (read, write, read-write) is allowed to the memory region in each protection domain. A local memory area can thus be registered and associated with several queue pairs, with the same or

different access levels, by associating it with one or more protection domains. Each RDMA connection can be used by the remote server to access registered memory regions using a unique region-specific key created as a part of the registration process.

As highlighted by previous work [45], failures of the CPU, NIC and DRAM can be seen as independent (e.g., arbitrary delays, too many bit errors, failed ECC checks, respectively). For instance, *zombie servers* in which the CPU is blocked but RDMA requests can still be served account for roughly half of all failures [45]. This motivates our choice to treat processes and memory separately in our model. In practice, if a CPU fails permanently, the memory will also become unreachable through RDMA eventually; however, in such cases memory may remain available long enough for ongoing operations to complete. Also, in practical settings it is possible for full-system crashes to occur (e.g., machine restarts), which correspond to a process and a memory failing at the same time—this is allowed by our model.

Memory regions in our model correspond to RDMA memory regions. Static permissions can be implemented by making the appropriate memory region registration before the execution of the algorithm; these permissions then persist during execution without CPU involvement. Dynamic permissions require the host CPU to change the access levels; this should be done in the OS kernel: the kernel creates regions and controls their permissions, and then shares memory with user-space processes. In this way, Byzantine processes cannot change permissions illegally. The assumption is that the kernel is not Byzantine. Alternatively, future hardware support similar to SGX could even allow parts of the kernel to be Byzantine.

Using RDMA, a process p can grant permissions to a remote process q by registering memory regions with the appropriate access permissions (read, write, or read/write) and sending the corresponding key to q. p can revoke permissions dynamically by simply deregistering the memory region.

For our non-equivocation algorithm, each process can register the two dimensional array of values in read-only mode with a protection domain. All the queue pairs used by that process are also created in the context of the same protection domain. Additionally, the process can preserve write access permission to its row via another registration of just that row with the protection domain, thus enabling single-writer multiple-reader access. Thereafter the non-equivocation algorithm can be implemented trivially by using RDMA reads and writes by all processes. Non-equivocation with unreliable memories is similarly straightforward since failure of the memory ensures that no process will be able to access the memory.

For Cheap Quorum, the static memory region registrations are straightforward as above. To revoke the leader's write permission, it suffices for a region's host process to deregister the memory region. Panic messages can be relayed using RDMA message sends.

In our crash-only consensus algorithm, we leverage the capability of registering overlapping memory regions in a protection domain. As in above algorithms, each process uses one protection domain for RDMA accesses. Queue pairs for connections to all other processes are associated with this protection domain. The process' entire slot array is registered with the protection domain in read-only mode. In addition, the same slot array can be dynamically registered (and deregistered) in write mode based on incoming write

permission requests: A proposer requests write permission using an RDMA message send. In response, the acceptor first deregisters write permission for the immediate previous proposer. The acceptor thereafter registers the slot array in write mode and responds to the proposer with the new key associated with the newly registered slot array. Reads of the slot array are performed by the proposer using RDMA reads. Subsequent second phase RDMA write of the value can be performed on the slot array as long as the proposer continues to have write permission to the slot array. The RDMA write fails if the acceptor granted write permission to another proposer in the meantime.

REFERENCES

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed computing (DIST)*, 18(5):387–408, 2006.

[2] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 47–58, August 1992.

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

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The Impact of RDMA on Agreement. *ArXiv preprint arXiv:1905.12143*, 2019.

[5] Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. Tight bounds for shared memory systems accessed by byzantine processes. *Distributed computing (DIST)*, 18(2):99–109, 2005.

[6] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[8] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[9] Rida Bazzi and Gil Neiger. Optimally simulating crash failures in a byzantine environment. In *International Workshop on Distributed Algorithms (WDAG)*, pages 108–128. Springer, 1991.

[10] Jonathan Behrens, Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel, Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, and Robbert Van Renesse. Derecho: Group communication at the speed of light. Technical report, Technical Report. Cornell University, 2016.

[11] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30. ACM, 1983.

[12] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):419–432, 2009.

[13] Romain Boichat, Partha Dutta, Svend Frolund, and Rachid Guerraoui. Reconstructing paxos. *SIGACT News*, 34(2):42–57, March 2003.

[14] Zohir Bouzid, Damien Imbs, and Michel Raynal. A necessary condition for byzantine k-set agreement. *Information Processing Letters*, 116(12):757–759, 2016.

[15] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[16] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[17] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, pages 42–50. Springer, 2001.

[18] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[19] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference (ATC)*, pages 287–292, 2008.

[20] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–204, 2007.

[21] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (limited) power of non-equivocation. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 301–308. ACM, 2012.

[22] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *International Symposium on Reliable Distributed Systems (SRDS)*, pages 174–183, 2004.

[23] Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous byzantine consensus with 2f+ 1 processes. In *ACM symposium on applied computing (SAC)*, pages 475–480. ACM, 2010.

[24] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN)*, pages 137–146. IEEE Computer Society, 2006.

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

[26] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *International Conference on Dependable Systems and Networks (DSN)*, pages 22–27. IEEE, 2005.

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

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

[29] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed computing (DIST)*, 16(1):1–20, 2003.

[30] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, May 1998.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, volume 16, pages 185–201, 2016.

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

[34] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conference on Computer Systems (EuroSys)*, pages 295–308, 2012.

[35] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2):45–63, 2001.

[36] Klaus Kursawe. Optimistic byzantine agreement. In *International Symposium on Reliable Distributed Systems (SRDS)*, pages 262–267, October 2002.

[37] Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM (JACM)*, 30(3):668–676, July 1983.

[38] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[39] Leslie Lamport. Fast paxos. *Distributed computing (DIST)*, 19(2):79–103, 2006.

[40] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[41] Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by byzantine processes. *Distributed computing (DIST)*, 16(1):37–48, 2003.

[42] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(3):202–215, July 2006.

[43] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[44] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

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

[46] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. Towards low-latency byzantine agreement protocols using RDMA. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 146–151. IEEE, 2018.

[47] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing (DISC)*, pages 438–450, September 2008.

[48] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

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