DARE: High-Performance State Machine Replication on RDMA Networks

(Extended Version)

Marius Poke
Department of Computer Science
ETH Zurich
marius.poke@inf.ethz.ch

Torsten Hoefler
Department of Computer Science
ETH Zurich
htor@inf.ethz.ch

ABSTRACT

The increasing amount of data that needs to be collected and analyzed requires large-scale datacenter architectures that are naturally more susceptible to faults of single components. One way to offer consistent services on such unreliable systems are replicated state machines (RSMs). Yet, traditional RSM protocols cannot deliver the needed latency and request rates for future large-scale systems. In this paper, we propose a new set of protocols based on Remote Direct Memory Access (RDMA) primitives. To asses these mechanisms, we use a strongly consistent key-value store; the evaluation shows that our simple protocols improve RSM performance by more than an order of magnitude. Furthermore, we show that RDMA introduces various new options, such as log access management. Our protocols enable operators to fully utilize the new capabilities of the quickly growing number of RDMA-capable datacenter networks.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance attributes; Fault tolerance

Keywords

Replicated State Machine; RDMA; Performance; Reliability

1. INTRODUCTION

The rapid growth of global data-analytics and web-services requires scaling single logical services to thousands of physical machines. Given the constant mean time between failures per server (≈2 years in modern datacenters [48]) the probability of a single-server failure grows dramatically; for example, a system with 1,000 servers would fail more than once a day if a single server failure causes a global system outage.

Replicated state machines (RSMs) prevent such global outages and can hide server failures while ensuring strong consistency of the overall system. RSMs are often at the core of global-scale services (e.g., in Google's Spanner [15] or Yahoo!'s Zookeeper [28]). However, typical RSM request rates are orders of magnitude lower than the request rates of the overall systems. Thus, highly scalable systems typically utilize RSMs only for management tasks and improve overall performance by relaxing request ordering [17] which implicitly shifts the burden of consistency management to the application layer. Yet, many services, such as airline reservation systems, require a consistent view of the complete distributed database at very high request rates [51].

To ensure strong consistency, RSMs usually rely on a distinguished server—the leader—to order incoming requests [28, 37, 45]. Moreover, before answering a request, the leader must replicate the enclosed state machine update on the remote servers; thus, efficient replication is critical for high performance. Existing TCP or UDP-based RSMs perform these remote updates through messages; thus, the CPUs of the remote servers are unnecessarily involved into the replication process. A more fitting approach is to use remote direct memory access (RDMA): RDMA allows the leader to directly access the memory of the remote servers without involving their CPUs. The benefit of bypassing the remote CPUs is twofold: (1) the synchronization between the leader and the remote servers is reduced; and (2) while the leader handles requests, the remote servers can perform other operations, such as tacking checkpoints.

In this work, we utilize RDMA networking to push the limits of reliable high-performance RSM implementations by more than an order of magnitude. High-performance RDMA network architectures such as InfiniBand or RDMA over Converged Ethernet (RoCE) are quickly adopted in datacenter networking due to their relatively low cost and high performance. However, to exploit the whole potential of RDMA-capable networks, new algorithms need to be carefully designed for remote memory access. Indeed, simply emulating messages over RDMA (e.g., using the IPoIB protocol) leaves most of the performance potential untapped [23]. Our main contribution is a set of protocols for implementing high-performance RSMs by using RDMA techniques in unconventional ways.

We design a novel *wait-free* direct access protocol, called DARE (Direct Access REplication), that uses RDMA features such as QP disconnect (§ 3.2.1) and QP timeouts (§ 3.4, § 4.2.1), in atypical ways to ensure highest performance and reliability. For performance, DARE replicates state machine updates entirely through RDMA (§ 3.3.1); also, it proposes efficient RDMA algorithms for leader election and failure detection. For reliability, DARE builds on a detailed failure-

HPDC'15, June 15–19, 2015, Portland, OR, USA.

ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749267.

model for RDMA networks that considers CPU, NIC, network, and memory failures separately (§ 5). Our implementation of DARE improves the latency compared to existing RSM protocols by up to 35 times and continues operation after a leader failure in less than $35ms$.

In summary, our work makes the following contributions:

- a complete RDMA RSM protocol and open-source reference implementation using InfiniBand (§ 3);
- an RDMA performance model of DARE in a failure-free scenario (§ 3.3.3);
- a failure model for RDMA systems in which we analyze both the availability and reliability of DARE (§ 5);
- a detailed performance analysis showing that our protocol has between 22 and 35 times lower latency than existing approaches (§ 6);
- a demonstration how DARE can be used to implement a strongly-consistent key-value store (§ 6).

2. BACKGROUND

Replicated state machines (RSMs) [49] provide reliable distributed services [32]. They replicate a (potentially infinite) state machine (SM), such as a key-value store (KVS), over a *group of servers*. A server consists of one or more processors that share the same volatile memory; also, it acts as an endpoint in the system's interconnection network. The servers update their SM replicas by applying *RSM operations*. Usually, they store the RSM operations into buffers—the local *logs*. Then, they apply each operation in order. For consistency, the logs must contain the same sequence of operations. Thus, the servers must agree on each RSM operation's position in the log; that is, they need to reach *consensus*.

2.1 Consensus

We consider a group of P servers out of which at most f are faulty. We assume a *fail-stop* model: A *faulty* server operates correctly until it fails and once it fails, it can no longer influence the operation of other servers in the group. A server that did not fail is called *non-faulty*.

In the consensus problem, each server has an *input* and an initially unset *output*. The servers propose their inputs; then, they irreversibly decide upon a value for the outputs such that three conditions are satisfied—agreement, validity and termination. Agreement requires any two non-faulty servers to decide the same; validity requires that if all the inputs are the same the value decided upon is the common input; and termination requires that any non-faulty server decides [7]. The three conditions ensure the main properties of consensus: safety and liveness [34]. *Safety* is provided by agreement and validity; *liveness* is provided by termination.

It is common for consensus protocols to adopt a *leader-based* approach [28, 34, 37, 43, 45]. Leader-based consensus protocols delegate the proposal and decision to a distinguished server—the *leader*. The leader can both propose and decide upon values until the other servers decide to elect a new leader (§ 3.2). Therefore, both conditions required by safety—agreement and validity—can be satisfied if there is no more than one leader at any given time.

The impossibility result of Fischer, Lynch, and Paterson, states that liveness cannot be ensured in an asynchronous model where servers can fail [20]: In an asynchronous model we cannot tell whether a server has failed or it is just running slow. To overcome this result, we augment a synchronous model by a failure detector (FD)—a distributed oracle that provides (possible incorrect) information about faulty servers [12]. In particular, we use a $\diamond\mathcal{P}$ FD, which satisfies both *strong completeness* and *eventual strong accuracy*. Strong completeness requires that eventually every faulty server is suspected to have failed by every non-faulty server. Eventual strong accuracy requires that eventually every non-faulty server is trusted by every non-faulty server. A $\diamond\mathcal{P}$ FD guarantees the termination of leader-based consensus only if a majority of the servers are non-faulty [12]. Therefore, for the remainder of the paper, we consider a group of P servers, out of which up to $f = \left\lfloor \frac{P-1}{2} \right\rfloor$ can fail.

	RDMA/rd	**RDMA/wr**		**UD**	
$o_p = 0.07\mu s$			inline		inline
o $[\mu s]$	0.29	0.26	0.36	0.62	0.47
L $[\mu s]$	1.38	1.61	0.93	0.85	0.54
G $[\mu s/KB]$	0.75	0.76	2.21	0.77	1.92
G_m $[\mu s/KB]$	0.26	0.25	-	-	-

Table 1: LogGP parameters on our system.

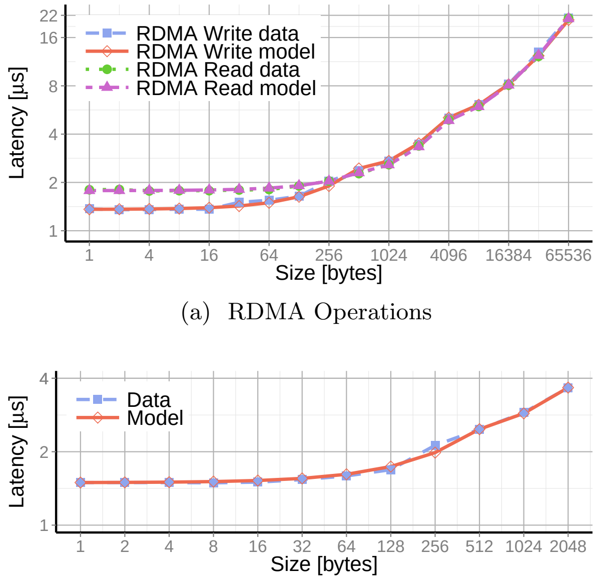
2.2 RDMA overview

Remote Direct Memory Access is an interface that allows servers to access memory in the user-space of other servers. To enable this mechanism, two user-space processes establish so called Queue Pairs (QPs) and connect them. Each QP is a logical endpoint for a communication channel and has a context in the NIC. Before memory can be accessed, a direct mapping from logical addresses in user-space to physical memory addresses is established in the NIC (this memory exposure is performed through the operating system (OS) to guarantee protection). The remote access is then fully performed by the hardware (using a reliable transport channel) without any interaction with the OS at the origin or target of the access. In this way, the NIC can be seen as a separate but limited processor that enables access to remote memory. This mechanism is fundamentally different from existing message-passing mechanisms where messages are processed by the main CPU. Thus, RDMA changes the system's failure-characteristics: a CPU can fail but its memory is still remotely accessible (see Section 5).

Modern networks, such as InfiniBand, also offer unreliable datagram (UD) messaging semantics that support multicast. We use these to simplify non-performance-critical parts of our protocol such as setup and interaction with clients.

2.3 Modeling RDMA performance

We estimate the performance of RDMA operations through a modified LogGP model [6]. The LogGP model consist of the following parameters: the latency L; the overhead o; the gap between messages g; the gap per byte for long messages G; and the number of processes (or servers) P. We make the common assumption that $o > g$ [6]; also, we assume that control packets, such as write acknowledgments and read requests, are of size one byte. Moreover, we readjust the model to fit the properties of RDMA communication. In particular, we make the following assumptions: (1) the overhead of the target of the access is negligible; (2) the latency of control packets is integrated into the latency of RDMA accesses; (3) for large RDMA accesses, the band-



(a) RDMA Operations

(b) Messages over UD

Figure 1: Evaluation of the RDMA performance models. Both the data and the LogGP parameters are gathered from a 12-node InfiniBand cluster § 6 with an MTU of 4096 bytes.

width increases after transferring the first MTU bytes; (4) for RDMA write operations, L, G, and o depend on whether the data is sent inline; and (5) o_p is the overhead of polling for completion. Table 1 specifies the LogGP parameters for our system, i.e., a 12-node InfiniBand cluster (§ 6).

According to the assumptions above, the time of either writing or reading s bytes through RDMA is estimated by

$$\begin{cases} o_{in} + L_{in} + (s-1)G_{in} + o_p & \text{if inline} \\ o + L + (s-1)G + o_p & \text{if } s \leq m \\ o + L + (m-1)G + (s-m)G_m + o_p & \text{if } s > m, \end{cases} \tag{1}$$

where m is the MTU of the system, G is the gap per byte for the first m bytes, and G_m is the gap per byte after the first m bytes. Besides RDMA operations, DARE uses also unreliable datagrams (UDs). To estimate the time of UD transfers, we use the original LogGP model; thus, the time of sending s bytes over UD is

$$\begin{cases} 2o_{in} + L_{in} + (s-1)G_{in} & \text{if inline} \\ 2o + L + (s-1)G & \text{otherwise.} \end{cases} \tag{2}$$

Both RDMA and UD models fit the data on our system with coefficients of determination larger than 0.99. Also, the good fit of the model is visible in Figures 1a and 1b, where both the data and the model are plotted.

3. THE DARE PROTOCOL

DARE is an RSM protocol that solves consensus through a leader-based approach: A distinguished leader acts as the interface between clients and the SMs. When the leader is suspected to have failed, the servers elect another leader. Each election causes the beginning of a new *term*—a period of time in which at most one leader exits. A server that wins an election during a term becomes the leader of that term. Furthermore, to make progress, DARE requires the existence of a *quorum*; that is, at least $q = \left\lceil \frac{P+1}{2} \right\rceil$ servers must "agree" on the next step. This ensures that after any $f = \left\lfloor \frac{P-1}{2} \right\rfloor$ failures there is still at least one non-faulty server that is aware of the previous step (since $q > f$). That server guarantees the safe continuation of the protocol.

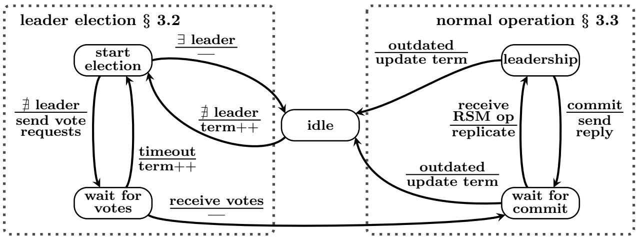


Figure 2: Outline of both leader election and normal operation protocols of DARE. Solid boxes indicate states; arrows indicate transitions. Each transition is described by its precondition (top) and postcondition (bottom).

Existing leader-based RSM protocols and implementations, such as VR [37], Raft [45], and ZooKeeper [28], rely on message passing, often implemented over UDP or TCP channels. DARE replaces the message-passing mechanism with RDMA; it assumes that servers are connected through an interconnect with support for RDMA, such as InfiniBand [1]. To our knowledge, DARE is the first RSM protocol that can exploit the whole potential of RDMA-capable networks. All of the main sub-protocols (see below) entail the design of special methods in order to support remotely accessible data structures; we detail the design of these methods in the following subsections.

DARE outline. We decompose the DARE protocol into three main sub-protocols that contribute to the implementation of an RSM:

- **Leader election:** the servers elect a distinguished server as their leader (§ 3.2).
- **Normal operation:** the leader applies RSM operations in a consistent manner (§ 3.3).
- **Group reconfiguration:** either the group's membership or size changes (§ 3.4).

The first two sub-protocols—leader election and normal operation—are the essence of any leader-based RSM protocol; group reconfiguration is an extension that enables DARE to change the set of servers in the group. Figure 2 shows an outline of both leader election and normal operation.

All servers start in an idle state, in which they remain as long as a leader exists. When a server suspects the leader to have failed (see the heartbeat mechanism in Section 4.2.2) it starts a new election (left side of Figure 2). First, it proposes itself as the leader for the subsequent term by sending vote requests to the other servers. Then, the server either becomes leader after receiving votes from a quorum (itself included) or it starts a new election after timing out (§ 3.2). Note that if another server becomes leader, the server returns to the idle state.

Once a server becomes leader, it starts the normal operation protocol (right of Figure 2). In particular, it must ensure the consistency of the SM replicas. Thus, when it receives an RSM operation the leader replicates it on the other servers with the intention to *commit* it; for safety, an RSM

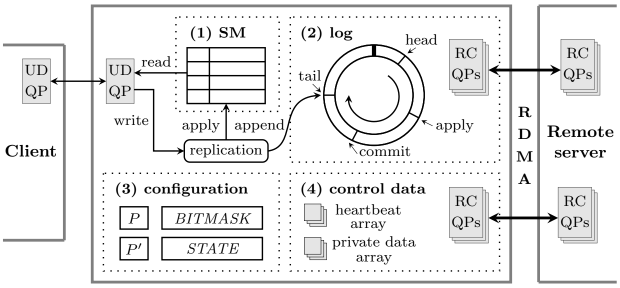


Figure 3: The internal state and interface of a DARE server.

operation is committed when it resides on at least a majority of servers. After an RSM operation is committed, the leader sends a reply to the client that sent the operation. Finally, a leader returns to the idle state if it is *outdated*. An outdated leader is a server that regards itself as leader, although another leader of a more recent term exists; for example, a temporary overload on the current leader can cause a majority of the servers to elect a new leader.

In the remainder of this section, we first present the basics of the DARE protocol (§ 3.1): we specify the internal state of a server; and we outline how both clients and servers interact with each other. Then, we describe in detail the three main sub-protocols of DARE: leader election; normal operation; and group reconfiguration.

3.1 DARE basics

3.1.1 Server internal state

The internal state of each server consists of four main data structures depicted in Figure 3: (1) the client SM; (2) the log; (3) the configuration; and (4) the control data. The *SM* is an opaque object that can be updated by the server through RSM operations received from clients. For consistency, servers apply the RSM operations in the same order; this is achieved by first appending the operations to the log.

The *log* is a circular buffer composed of entries that have sequential indexes; each entry contains the term in which it was created. Usually, log entries store RSM operations that need to be applied to the SM; yet, some log entries are used by DARE for internal operations, such as log pruning (§ 3.3.2) and group reconfiguration (§ 3.4). Similar to RSM operations, log entries are called *committed* if they reside on a majority of servers. The log is described by four dynamic pointers, which follow each other clockwise in a circle:

- **head** points to the first entry in the log; it is updated locally during log pruning (§ 3.3.2);
- **apply** points to the first entry that is not applied to the SM; it is updated locally;
- **commit** points to the first not-committed log entry; it is updated by the leader during log replication (§ 3.3.1);
- **tail** points to the end of the log; it is updated by the leader during log replication (§ 3.3.1).

The *configuration* data structure is a high level description of the group of servers. It contains four fields: the current group size P; a bitmask indicating the active servers; the new group size P'; and an identifier of the current state. The last two fields are needed by DARE for resizing the group without interrupting normal operation [45]. Section 3.4 describes in details the role of these fields in DARE's group reconfiguration protocol.

Finally, the *control data* consists of a set of arrays that have an entry per server. One such array is the *private data array* that is used by servers as reliable storage (§ 3.2.3). Another example is the *heartbeat array* used by the leader to maintain its leadership (§ 4.2.2). We specify the rest of the arrays as we proceed with the description of the protocol.

In-memory data structures: benefits and challenges. The internal state of a DARE server consists of in-memory data structures. The benefit of an in-memory state is twofold. First, accessing in-memory data has lower latency than accessing on-disk data. Second, in-memory data structures can be remotely accessed through RDMA. In particular, in DARE, the leader uses the commit and tail pointers to manage the remote logs directly through RDMA. Thus, since the target servers are not active, they can perform other operations such as saving the SM on stable storage for higher reliability. Also, RDMA accesses are performed by the hardware without any interaction with the OS; this often leads to higher performance as compared to message passing [23].

Yet, the in-memory approach entails that the entire state is volatile; thus, when high reliability is required, DARE uses *raw replication*. Raw replication makes an item of data reliable by scattering copies of it among different nodes. Thus, this approach can tolerate a number of simultaneous node failures equal to the numbers of copies. In Section 5, we discuss reliability in more details.

3.1.2 Communication interface

DARE relies on both unreliable and reliable communication. Unreliable communication is implemented over UD QPs, which support both unicast (i.e., one-to-one) and multicast (i.e., one-to-many) transfers. The multicast support makes UD QPs practical in the context of a dynamic group membership, where the identity of the servers may be unknown. Thus, we implement the interaction between group members and clients over UD QPs. Note that new servers joining the group act initially as clients and, thus, they also use the UD QPs to access the group (§ 3.4).

The InfiniBand architecture specification's Reliable Connection (RC) transport mechanism does not lose packets [29]; therefore, DARE implements reliable communication over RC QPs. Since the servers need remote access to both the log and the control data, any pair of servers is connected by two RC QPs: (1) a control QP that grants remote access to the control data; and (2) a log QP that grants remote access to the local log (see Figure 3).

3.2 Leader election

We adopt a traditional leader election protocol to RDMA semantics: A server sends vote requests and it waits for votes from at least $\lfloor P/2 \rfloor$ servers, before it becomes the leader. In addition, a server cannot vote twice in the same term. Thus, DARE guarantees at most one leader per term, which satisfies the safety requirement of a leader-based consensus protocol (see Section 2.1).

In the remainder of this section, we describe our RDMA design of the voting mechanism. Figure 4 outlines this mechanism during a successful leader election in a group of three

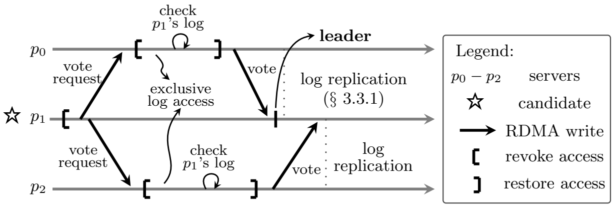


Figure 4: The voting mechanism during leader election.

servers. Although our approach is similar to a message-passing one, it requires special care when managing the log accesses. In particular, by using RDMA semantics, the leader bypasses the remote CPUs when accessing their logs; as a result, the servers are unaware of any updates of their logs. This hinders the ability of a server to participate in elections. Thus, we first outline how DARE uses *QP state transitions* to allow servers to manage the remote access to their own memory; then, we describe the voting mechanism.

3.2.1 Managing log access

Once a QP is created, it needs to be transitioned through a sequence of states to become fully operational; moreover, at any time the QP can be locally reset to the original state, which is non-operational. Thus, DARE servers can decide on either exclusive local access or shared remote access. For exclusive local access, the QP is reset to the original non-operational state; while for remote log access, the servers move the QP in the ready-to-send state [29], which is fully-operational. Besides managing access to their logs, DARE servers use QP state transitions for both connecting and disconnecting servers during group reconfiguration (§ 3.4).

3.2.2 Becoming a candidate

The leader election protocol starts when a server suspects the leader to have failed. In Figure 4, server p_1 starts an election by revoking remote access to its log; this ensures that an outdated leader cannot update the log. Then, it proposes itself as a *candidate* for the leadership of the subsequent term. That is, it sends vote requests to the other servers: It updates its corresponding entry in the *vote request array* (one of the control data arrays) at all other servers by issuing RDMA write operations (see Figure 4). An entry in the vote request array consists of all the information a server requires to decide if it should vote for the candidate: the candidate's current term and both the index and the term of the candidate's last log entry (see Section 3.2.3).

Depending on the internal state of the candidate, we distinguish between three possible outcomes (depicted in the left side of Figure 2): (1) the candidate becomes leader if it receives the votes from at least $\lfloor P/2 \rfloor$ servers; (2) it decides to support the leadership of another candidate more suited to become leader (§ 3.2.3); or (3) otherwise, it starts another election after a timeout period. Note that the candidate restores remote log access for every server from which it received a vote; this ensures that a new leader can proceed with the log replication protocol (§ 3.3.1). Also, two or more candidates may receive collectively enough votes to become leader, yet, not enough individually; thus, a new election starts. If the candidates are synchronized, this scenario can repeat several times. reducing the availability (or even the liveness) of the protocol. To avoid this, DARE servers use randomized timeouts [45].

3.2.3 Answering vote requests

Servers not aware of a leader periodically check the vote request array for incoming requests. They only consider requests for the leadership of a higher (more recent) term than their own; on receiving a valid request, the servers increase their own term. In Figure 4, servers p_0 and p_2 receive vote requests from candidate p_1. Both servers grant their vote after first checking that the candidate's latests log entry is at least as recent as their own; an entry is more recent than another if it has either a higher term or the same term but a higher index [45]. By checking a candidate's latest log entry, the servers ensure that the log of a possible leader contains the most recent entry among a majority of servers; in Section 4.1, we show that this property is essential for DARE's safety. Also, note that while performing the check, both servers need exclusive access to their own logs (see Figure 4).

A server's volatile internal state introduces an additional challenge. The server may fail after voting for a candidate, and then, recover during the same term. If after recovery, it receives a vote request from another candidate, but for the same term, the server could grant its vote. Thus, two servers may become leaders during the same term, which breaks the safety of our protocol. To avoid such scenarios, prior to answering vote requests, each server makes its decision reliable by replicating it via the private data array (§ 3.1.1).

RDMA vs. MP: leader election. Our RDMA design of leader election increases DARE's availability. When the leader fails, the RSM becomes unavailable until a new leader is elected. First, for leader election to start, the servers need to detect the failure (§ 4.2.2). Then, the election time depends on the period a candidate waits for votes before restarting the election. This period needs to be large enough for the vote requests to reach the servers and at least $\lfloor P/2 \rfloor$ votes to arrive back at the candidate. The RDMA-capable interconnect allows us to reduce this period; thus, our design increases the RSM's availability. In addition, due to the in-memory approach, servers do not need to write their vote on stable storage before answering a vote request.

3.3 Normal operation

The normal operation protocol entails the existence of a sole leader that has the support of at least a majority of servers (including itself). The leader is responsible for three tasks: serving clients; managing the logs; and, if needed, reconfiguring the group. In the remainder of this section, we describe the first two tasks; we defer the discussion of group reconfiguration to Section 3.4. First we specify how clients interact with DARE. Then, we present a log replication protocol designed entirely for RDMA (§ 3.3.1); also, we outline a log pruning mechanism that prevents the overflowing of the logs. Finally, we provide an RDMA performance model of DARE during normal operation.

Client interaction. Clients interact with the group of servers by sending requests through either multicast or unicast. To identify the leader of the group, clients send their first request via multicast. Multicast requests are considered only by the leader. Once the leader replies, clients send subsequent requests directly to the leader via unicast. However, if the request is not answered in a predefined period of time, clients re-send the request through multicast. Note that this is also the case when sending requests to an outdated leader, since it cannot answer client requests (see below). Also, the current implementation assumes that a client waits for a reply before sending the subsequent request. Yet, DARE handles different clients asynchronously: The leader can execute, at the same time, requests from multiple clients. This increases the protocol's throughput (see Figure 8b).

Write requests. Regardless of the nature of the SM, clients can send either write or read requests. *Write requests* contain RSM operations that alter the SM; for example, a write request may consist of updating the value of a key in a KVS. Such operations need to be applied to all SM replicas in the same order. Therefore, when receiving a write request, the leader stores the RSM operation into an entry that is appended to the log. Then, it replicates the log entry on other servers with the purpose to commit it (see Section 3.3.1). As a safety requirement, each DARE server applies only RSM operations stored in committed log entries.

Write requests may contain RSM operations that are not idempotent (i.e., they change the SM replicas every time they are applied). DARE ensures that each RSM operation is applied only once by enforcing *linearizable semantics* [27] through unique request IDs (as other RSM protocols). Furthermore, to increase the throughput of strongly consistent writes, DARE executes write requests in batches: The leader first appends the RSM operations of all consecutively received write requests to the log; then, it replicates all the entries at once.

Read requests. *Read requests* contain RSM operations that do not alter the SM; for example, a read request may consist of getting the value of a key from a KVS. For such operations, replication is not required: For efficiency, the leader answers read requests directly from the local SM. Yet, to ensure that reads do not return stale data, DARE imposes two constraints: (1) an outdated leader cannot answer read requests; and (2) a leader with an outdated SM cannot answer read requests.

First, to verify whether a leader is outdated, DARE uses a property of leader election—any successful election requires at least a majority of servers to increase their terms (cf. § 3.2.3). Therefore, before answering a read request, the leader reads the term of at least $\lfloor P/2 \rfloor$ servers; if it finds no term larger than its own, then it can safely answer the read request. As an optimization, the leader verifies whether it is outdated only once for a batch of consecutively received read requests; thus, DARE's read throughput increases.

Second, before answering a read request, the leader must ensure that all RSM operations stored in committed log entries are applied to the local SM. In Section 4.1, we show that the leader's log contains all the committed entries that store RSM operations not yet applied by all non-faulty servers. However, a new leader may not be aware of all the committed entries (§ 3.3.1); thus, the local SM is outdated. As a solution, a new leader appends to its log an entry with no RSM operation. This starts the log replication protocol that commits also all the preceding log entries.

3.3.1 Log replication

The core of the normal operation protocol is *log replication*—the leader replicates RSM operations on the other servers with the intention to commit them (see right side of Figure 2). In DARE, log replication is performed entirely through RDMA: The leader writes its own log entries into the logs of the remote servers. Yet, after a new leader is elected, the logs can contain not-committed entries. These entries may differ from the ones stored at the same position in the leader's log; for example, Figure 5a shows the logs after server p_1 becomes the leader of a group of three servers. Before a newly elected leader can replicate log entries, it must first remove all the remote not-committed entries. Therefore, we split the log replication protocol into two phases: (1) log adjustment; and (2) direct log update.

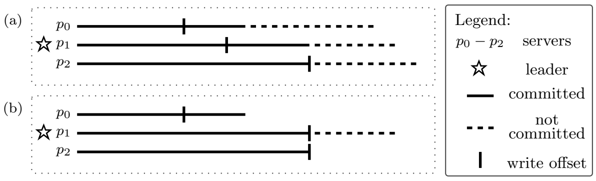


Figure 5: The logs in a group of three servers: (a) after server p_1 is elected leader; and (b) after log adjustment. For clarity, the apply pointers are omitted.

Log adjustment. One naïve approach to adjust a remote log is to set its tail pointer to the corresponding commit pointer. However, this may remove committed entries (see server p_0's log in Figure 5a); as a result, a committed entry may no longer be replicated on a majority of servers, which breaks the safety of our protocol. A correct approach sets the remote tail pointer to the first not-committed entry; note that due to the "lazy" update of the commit pointers (see below), a server may not be aware of all its committed log entries. Thus, to adjust a remote log, the leader performs two subsequent RDMA accesses (labeled by `a` and `b` in Figure 6): (1) it reads the remote not-committed entries; and (2) it sets the remote tail pointer to the offset of the first non-matching entry when compared to its own log. In addition, the leader updates its own commit pointer. Figure 5b shows the logs of the three servers after the log adjustment phase is completed.

Reading the remote not-committed entries could be rather costly in case of large RSM operations. Yet, the RSM operations are not needed for the leader to identify the first non-matching entry. Thus, as an optimization, before granting log access to a new leader, each server creates a buffer with only the index and the term of all of its not-committed entries. As a result, the leader avoids large data transfers over the network.

Direct log update. The second phase of log replication consists of three RDMA accesses (labeled by `c`, `d` and `e` in Figure 6). First, for each adjusted remote log, the leader writes all entries between the remote and the local tail pointers. Second, the leader updates the tail pointers of all the servers for which the first access completed successfully. To commit log entries, the leader sets the local commit pointer to the minimum tail pointer among at least a majority of servers (itself included). Finally, for the remote servers to apply the just committed entries, the leader "lazily" updates the remote commit pointers; by lazy update we mean that there is no need to wait for completion.

Asynchronous replication. During log replication, the leader handles the remote logs asynchronously. Figure 6 shows the RDMA accesses during log replication in a group of three servers. Once the leader receives the confirmation that server p_0's log is adjusted, it starts updating it, although it is not yet aware that server p_2's log is adjusted

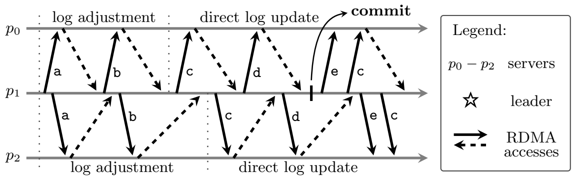


Figure 6: The RDMA accesses during log replication: (`a`) read the remote not-committed entries; (`b`) write the remote tail pointer; (`c`) write the remote log; (`d`) write the remote tail pointer; and (`e`) write the remote commit pointer.

(the access is delayed). When the delayed access completes, the leader can also start the direct log update phase for server p_2. Furthermore, the leader commits its log entries after updating the tail pointer of server p_0 (since $P = 3$).

RDMA vs. MP: log replication. Replicating the logs through RDMA has several benefits over the more traditional message-passing approach. RDMA accesses remove the overhead on the target, which has two consequences: first, the leader commits RSM operations faster; and second, the servers are available for other tasks, such as recovery (§ 3.4). Moreover, RDMA allows for servers with a faulty CPU, but with both NIC and memory working, to be remotely accessible during log replication, hence, increasing both availability and reliability (§ 5). Finally, RDMA allows for efficient log adjustment. In DARE, log adjustment entails two RDMA accesses regardless of the number of non-matching log entries; yet, in Raft [45] for example, the leader must send a message for each non-matching log entry.

3.3.2 Log pruning: removing applied entries

Every server applies the RSM operations stored in the log entries between its apply and commit pointers; once an operation is applied, the server advances its apply pointer. When an RSM operation is applied by all the non-faulty servers in the group, the entry containing it can be removed from the log. Thus, the leader advances its own head pointer to the smallest apply pointer in the group; then, it appends to the log an `HEAD` entry that contains the new head pointer. Servers update their head pointer only when they encounter a committed `HEAD` entry; thus, all subsequent leaders will be aware of the updated head pointer. Furthermore, when the log is full the leader blocks until the remote servers advance their apply pointers. To avoid waiting, the leader can remove the server with the lowest apply pointer on the grounds that it hinders the performance of the entire group (cf. [16]).

3.3.3 An RDMA performance model

DARE is designed as a high-performance RSM protocol. Its performance is given by the request latency—the amount of time clients need to wait for requests to be answered. Client requests have two parts: (1) the UD transfer, which entails both sending the request and receiving a reply; and (2) the RDMA transfer, which consist of the leader's remote memory accesses. We use Equations (1) and (2) from Section 2.3 to estimate the latency of both UD and RDMA transfers; for readability, we consider the gap per byte G only for the s bytes of either the read or written data.

The UD transfer entails two messages: one short that is sent inline (request for reads and replies for writes); and one long that transfers the data. Thus, the latency of the UD transfer is bounded (due to the above simplification) by

$$t_{UD} \geq 2o_{in} + L_{in} + \begin{cases} 2o_{in} + L_{in} + (s-1)G_{in} & \text{if inline} \\ 2o + L + (s-1)G & \text{otherwise} \end{cases}.$$

The latency introduced by RDMA accesses depends on the request type. For read requests, the leader needs to wait for at least $q-1$ RDMA reads to complete ($q = \left\lceil \frac{P+1}{2} \right\rceil$). Thus, the latency of the RDMA transfer in case of read requests is bounded by

$$t_{RDMA/rd} \geq (q-1)o + \max\{fo, L\} + (q-1)o_p,$$

where f is the maximum number of faulty servers; also, the max function indicates the overlap between the overhead of issuing the last f reads and the latency of the $(q-1)$st one.

For write requests, the leader needs to go through the steps of log replication. Yet, the logs are adjusted only once per term; thus, assuming a fairly stable leader, the latency of log adjustment is negligible. During the direct log update phase, the leader accesses the logs of at least $q-1$ servers; for each, it issues three subsequent RDMA write operations (see Figure 6). Thus, the latency of the RDMA transfer in case of a write request is bounded by

$$t_{RDMA/wr} \geq 2(q-1)o_{in} + L_{in} + 2(q-1)o_p +$$
$$\begin{cases} (q-1)o_{in} + \max\{fo_{in}, L_{in} + (s-1)G_{in}\} & \text{if inline} \\ (q-1)o + \max\{fo, L + (s-1)G\} & \text{otherwise} \end{cases},$$

where similar to read requests, the max function indicates the overlap between the last f log update operations and the latency of the $(q-1)$st one. Figure 8a compares these bounds with measurements gathered on our system (§ 6).

3.4 Group reconfiguration

DARE is intended for dynamical environments where servers can fail at any time. Thus, the group of servers can modify both its membership and its size; we refer to this as *group reconfiguration*. DARE handles group reconfigurations through the configuration data structure (§ 3.1.1). A configuration can be in three states: a stable state that entails a group of P servers with the non-faulty servers indicated by a bitmask; an extended state used for adding servers to a full group (see below); and a transitional state that allows for the group to be resized without interrupting normal operation [45]. The last two states require the new group size P' to be set.

We define three operations that are sufficient to describe all group reconfiguration scenarios: (1) remove a server; (2) add a server; and (3) decrease the group size. For example, since a server's internal state is volatile, a transient failure entails removing a server followed by adding it back; also, increasing the size entails adding a server to a full group. The three operations can be initiated only by a leader in a stable configuration. Each operation may entail multiple phases. Yet, each phase contains the following steps: the leader modifies its configuration; then, it appends to the log an entry that contains the updated configuration (a `CONFIG` entry); and once the `CONFIG` entry is committed, the phase completes and a possible subsequent phase can start. Note that when a server encounters a `CONFIG` log entry, it updates its own configuration accordingly regardless of whether the entry is committed. In the remainder of this section, we first describe how DARE implements the three operations; then, we outline how a server recovers its internal state. If not stated otherwise, we assume all configurations to be stable.

Removing a server. A server may be removed in one of the following cases: the log is full and cannot be pruned (§ 3.3.2); the group size is decreased; or the leader suspects the server to have failed. The leader detects failed (or unavailable) servers by using the QP timeouts provided by the RC transport mechanism [29]. In all cases, removing a server is a single-phase operation. First, the leader disconnects its QPs (§ 3.2.1) with the server. Then, it updates the bitmask of its configuration accordingly; also, it adds a `CONFIG` log entry with the updated configuration. Finally, once the log entry is committed, the server is removed.

Adding a server. Adding a server to a group is similar to removing a server; the only difference is that the QPs with the server must be connected instead of disconnected. Yet, if the group is full, adding a server requires first to increase the group size, which, without previously adding a server, decreases the fault-tolerance of the group. Intuitively, this is because the new group starts already with a failure since the new server is not yet added. Thus, adding a server to a full group is a three-phase operation: (1) adding the server; (2) increasing the group size; (3) stabilizing the configuration.

First, the leader establishes a reliable connection with the server; also, it creates an extended configuration with $P' = P + 1$. This configuration allows the added server to recover; yet, the server cannot participate in DARE's sub-protocols. Second, the leader increases the group size without interrupting normal operation [45]. In particular, it moves the configuration to a transitional state, in which all servers are participating in DARE's sub-protocols. The servers form two groups—the original group of P server and the new group of P' servers; majorities from both groups are required for both electing a leader and committing a log entry, such as the `CONFIG` entry containing the transitional configuration. Finally, the leader stabilizes the configuration: It sets P to the new size P' and it moves back into the stable state.

Decreasing the group size. Usually, adding more servers leads to higher reliability (see Figure 7); yet, it also decreases the performance, since more servers are required to form a majority. Thus, DARE allows the group size to be decreased. Decreasing the group size is a two-phase operation: first, the leader creates a transitional configuration that contains both the old and the new sizes; and then, it stabilizes it by removing the extra servers from the end of the old configurations. Note that the leader can be one of the servers that need to be removed; in this case, the leader removes itself once the decrease operation completes.

Recovery. When added to the group, a server needs to recover its internal state before participating in DARE's sub-protocols; in particular, it needs to retrieve both the SM and the log. To retrieve the SM, the new server asks any server, except for the leader, to create a snapshot of its SM; then, it reads the remote snapshot. Once the SM is recovered, the server reads the committed log entries of the same server. After it recovers, the server sends a vote to the leader as a notification that it can participate in log replication. Note that the recovery is performed entirely through RDMA.

RDMA vs. MP: recovery. Our RDMA approach reduces the impact of recovery on normal operation. The reason for this is twofold. First, contrary to message-passing RSMs, in DARE, the leader manages the logs directly without involving the CPUs of the remote servers; thus, servers can create a snapshot of their SM without interrupting normal operation. Second, the new server can retrieve both the SM and the log of a remote server directly through RDMA.

4. DARE: SAFETY AND LIVENESS

4.1 Safety argument

In addition to the safety requirement of consensus (§ 2.1), DARE guarantees the *RSM safety* property: each SM replica applies the same sequence of RSM operations. DARE exhibits similar properties as existing RSM protocols; thus, we use an analogous approach to show that the RSM safety property holds at all times. In particular, we show that DARE satisfies both *log matching* and *leader completeness* properties, which implies that it satisfies the RSM safety property [45].

Log matching. The log matching property states that two logs with an identical entry have all the preceding entries identical as well. We show that this property holds through complete induction on the term number. Initially, all logs are empty; therefore, the base case holds. Then, in the inductive step, we prove that if the property holds until the beginning of a given term, then it holds throughout that term. DARE guarantees that there is at most one leader during the term. If no leader exists, the logs remain unmodified throughout the term; hence, the property holds.

Thus, we assume the existence of a leader. According to the log replication protocol, the leader must adjust a remote log before updating it (§ 3.3.1). Once a log is adjusted, its most recent entry is identical to an entry in the leader's log; thus, according to the inductive step, all the preceding entries are identical as well. Moreover, to update the remote logs, the leader copies sequences of entries from its own log. Therefore, the log matching property holds at all times.

Leader completeness. The leader completeness property states that every leader's log contains all already-committed entries; note that, to account for log pruning, we consider only log entries that store RSM operations not yet applied by all non-faulty servers. To guarantee the leader completeness property, DARE relies on the following properties: (1) a leader requires votes from at least $\lfloor P/2 \rfloor$ servers (§ 3.2.2); (2) the leader's log contains the most recent entry among a majority of servers (§ 3.2.3); (3) a committed entry is replicated on a majority of servers (§ 3.3.1); and (4) the above log matching property. We refer the reader to [45], where the authors argue that, given the four properties above, the leader completeness property holds.

DARE servers apply only RSM operations enclosed in committed entries; also, a server applies the RSM operation enclosed in a log entry only after it applies all the RSM operations stored in log entries with lower indexes. Finally, the leader completeness property ensures that every committed entry eventually resides in the log of every non-faulty server. Therefore, DARE guarantees the RSM safety property.

4.2 Liveness argument

In theory, DARE's liveness is guaranteed by the following condition: there is a time after which a leader $\mathcal{L}$ answers all client requests. This implies (cf. § 3.3) that leader $\mathcal{L}$ must have the support of at least a majority of the servers (itself included). The eventual strong accuracy property of a $\diamond\mathcal{P}$ FD [12] ensures that eventually all the non-faulty servers, which are at least a majority (cf. § 2.1), always trust each other; without loss of generality, we assume one of these non-faulty servers to be $\mathcal{L}$. Thus, from a theoretical point of view, DARE guarantees liveness.

Yet, in practice, no leader can be trusted forever by other servers; it is sufficient though for a non-faulty leader to have the support of a majority of the servers for "sufficiently long" [4] so that it can answer at least one client request. Then, DARE must ensure that every time a leader fails (or loses the support of a majority) another leader is elected. We split this requirement into two conditions: (1) all the non-faulty servers eventually detect a faulty leader; and (2) once a faulty leader is detected, a new leader is eventually elected. The former condition results from the strong completeness property of a $\diamond\mathcal{P}$ FD (i.e., eventually, every faulty server is suspected to have failed by every non-faulty server). Moreover, the former condition ensures that sufficient servers (i.e., at least a majority) participate in the leader election sub-protocol; this, together with the randomized timeouts [45] used to restart a leader election, ensure the latter condition.

In the remainder of this section, we first describe how we utilize InfiniBand's timeout mechanisms [29] to obtain a model of partial synchrony [12] required by a $\diamond\mathcal{P}$ FD. Then, we outline how to implement a $\diamond\mathcal{P}$ FD with RDMA-semantics.

4.2.1 Synchronicity in RDMA networks

Synchronicity in the context of processors implies that there is a fixed bound on the time needed by a processor to execute any operation; intuitively, this guarantees the responsiveness of non-faulty processors. Since DARE uses RDMA for log replication, the processors are the NICs of the servers. These NICs are special-purpose processors that are in charge solely of the delivery of network packets at line-rate; that is, NICs avoid nondeterministic behavior, such as that introduced by preemption in general-purpose CPUs. Therefore, we can assume a bound on the execution time of NIC operations.

Synchronous communication requires a bound on the time within which any packet is delivered to a non-faulty server. Time can generally not be bound in complex networks, however, datacenter networks usually deliver packets within a tight time bound. InfiniBand's reliable transport mechanism does not lose packets and notifies the user if the transmission experiences unrecoverable errors [29]: It uses Queue Pair timeouts that raise unrecoverable errors in the case of excessive contention in the network or congestion at the target. Thus, the RC service of InfiniBand offers a communication model where servers can ascertain deterministically if a packet sent to a remote server was acknowledged within a bounded period of time.

4.2.2 Leader failure detection

The $\diamond\mathcal{P}$ FD used by DARE to detect failed leaders is based on an heartbeat mechanism implemented with RDMA semantics. Initially, every server suspects every other server to have failed. Then, the leader starts sending periodic heartbeats by writing its own term in the remote heartbeat arrays (§ 3.1.1). Every other server checks its heartbeat array regularly, with a period Δ: First, it selects the heartbeat with the most recent term; then, it compares this term with its own. If the terms are equal, then the leader is non-faulty; thus, the server extends its support. If its own term is smaller, then a change in leadership occurred; thus, the server updates its own term to indicate its support. Otherwise, the server assumes the leader has failed; thus, the strong completeness property holds [12].

Component	**AFR**	**MTTF**	**Reliability**
Network [18, 25]	1.00%	$876,000$	4-nines
NIC [18, 25]	1.00%	$876,000$	4-nines
DRAM [26]	39.5%	$22,177$	2-nines
CPU [26]	41.9%	$20,906$	2-nines
Server [25, 48]	47.9%	$18,304$	2-nines

Table 2: Worst case scenario reliability data. The reliability is estimated over a period of 24 hours and expressed in the "nines" notation; the MTTF is expressed in hours.

In addition, when a server finds an heartbeat with a term smaller than its own, it first increments Δ to ensure that eventually a non-faulty leader will not be suspected; thus, the eventual strong accuracy property holds [12]. Then, it informs the owner of the heartbeat that it is an outdated leader, so it can return to the idle state (see Figure 2).

Finally, once a server votes for a candidate, it resets the time elapsed since the latest check of the heartbeat array; thus, the candidate has a chance to become leader before the server starts a new leader election.

5. FINE-GRAINED FAILURE MODEL

As briefly mentioned in Section 2.2, RDMA requires a different view of a failing system than message passing. In message passing, a failure of either the CPU or OS (e.g., a software failure) disables the whole node because each message needs both CPU and memory to progress. In RDMA systems, memory may still be accessed even if the CPU is blocked (e.g., the OS crashed) due to its OS bypass nature.

To account for the effects of RDMA, we propose a failure model that considers each individual component—CPU, main memory (DRAM), and NIC—separately. We make the common assumption that each component can fail independently of the other components in the system [3, 11, 41]; for example, the CPU may execute a failed instruction in the OS and halt, the NIC may encounter too many bit errors to continue, or the memory may fail ECC checks. We also make the experimentally verified assumption that a CPU/OS failure does not influence the remote readability of the memory. Finally, we assume that the network (consisting of links and switches) can also fail.

Various sources provide failure data of systems and system components [18, 21, 24, 26, 39, 44, 46, 50]. Yet, systems range from very reliable ones with AFRs per component below 0.2% [39] to relatively unreliable ones with component failure log events at an annual rate of more than 40% [26] (here we assume that a logged error impacted the function of the device). Thus, it is important to observe the reliability of the system that DARE is running on and adjust the parameters of our model accordingly. For the sake of presentation, we pick the *worst case* for DARE, i.e., the highest component errors that we found in the literature. Table 2 specifies this for the main components over a period of one 24 hours.

Availability: zombie servers. We refer to servers with a blocked CPU, but with both a working NIC and memory as *zombie servers.* Zombie servers account for roughly

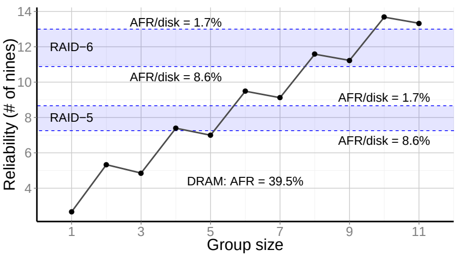


Figure 7: DARE's reliability over a period of 24 hours. Also, the reliability achieved by disks with RAID technologies.

half of the failure scenarios (cf. Table 2). Due to their nonfunctional CPU, zombie servers cannot participate in some protocols, such as leader election. Yet, DARE accesses remote memory through RDMA operations that consume no *receive request* on the remote QP, and hence, no *work completions* [29]. Therefore, a zombie server's log can be used by the leader during log replication increasing DARE's availability. Note that the log can be used only temporarily since it cannot be pruned and eventually the leader will remove the zombie server. Moreover, even in case of permanent CPU failures, zombie servers may provide sufficient time for recovery without losing availability.

Reliability. As briefly mentioned in Section 3.1.1, our design exploits the concept of memory reliability through raw replication. In particular, DARE uses raw replication in two situations: (1) explicitly by a server before answering a vote request during leader election; and (2) implicitly by the leader during log replication. In both situations, at least $q = \lceil \frac{P+1}{2} \rceil$ replicas are created. Thus, DARE's reliability is given by the probability that no more than $q-1$ servers experience a memory failure (cf. Table 2, the failures probabilities of both NIC and network are negligible).

To compute the reliability of our protocol, we assume that the components are part of non-repairable populations: Having experienced a failure, the same component can rejoin the system; however, it is treated as a new individual of the population. The reliability of non-repairable populations is estimated by *lifetime distribution models* (LDMs) [2]. An LDM is a probability density function $f(t)$ defined over the time range $t = (0, \infty)$. The corresponding cumulative distribution function $F(t)$, gives the probability that a randomly selected individual fails by time t. We denote the probability of a single component to fail during a time interval $(t_{j-1}, t_j]$ by h_j. We compute h_j from the probability to fail by the end of the interval, conditioned by the probability to be alive at the beginning of the interval. We refer to this time interval as the *epoch* j; note that an epoch is bounded by the expected lifetime of the components. Thus, (cf. Bayes' theorem) $h_j = \frac{F(t_j)-F(t_{j-1})}{1-F(t_{j-1})}$. Further, we introduce the discrete random variable X_j to denote the *number of components of a certain type that fail during epoch j*. Also, we consider k_j components to be available at the beginning of the epoch. Since all k_j components have the same failure probability h_j, the random variable X_j follows a binomial distribution (i.e., $X_j \sim B(k_j, h_j)$).

Components that failed during an epoch may not recover until the beginning of the subsequent epoch; thus, k_j varies. We express this variation through a discrete random variable K_j that denotes *the number of available components at the beginning of epoch j*. We simplify the analysis of K_j, by assuming that the recovery time of a component is less than the length of an epoch; such an assumption is valid since the only side effect of increasing the epoch length is higher failure probabilities. Under this assumption, $K_j = k_1 - X_j$, where k_1 is the total number of available components.

Therefore, we can express the probability of x components failing during epoch j by the following probability mass function (PMF):

$$Pr(X_j = x) = \sum_{k=x}^{k_1} Pr(K_j = k) \binom{k}{x} h_j^x (1-h_j)^{k-x},$$

where $Pr(K_j = k) = Pr(X_{j-1} = k_1 - k)$ gives the probability that $k \leq k_1$ components are available at the beginning of epoch j. The formula for $Pr(K_j = k)$ entails a recurrence with the initial value given by $Pr(X_1 = x) = \binom{k_1}{x} h_1^x (1-h_1)^{k_1-x}$.

Of particular interest to our analysis of reliability is the total number of unavailable components during an epoch j. We denote this number by the discrete random variable U_j, which consists of the combined failures in both j and $j-1$ epochs (i.e., $U_j = X_{j-1} + X_j$). Thus, we express the probability that $u \leq k_1$ components are unavailable during epoch j by the following PMF (cf. Bayes' theorem):

$$Pr(U_j = u) = \sum_{x=0}^{u} Pr(X_{j-1} = x) \binom{k_1 - x}{u - x} h_j^{u-x} (1-h_j)^{k_1-u}.$$

Finally, we estimate the probability that during an epoch j no more than κ (out of k_1) components are unavailable by the following cumulative distributed function:

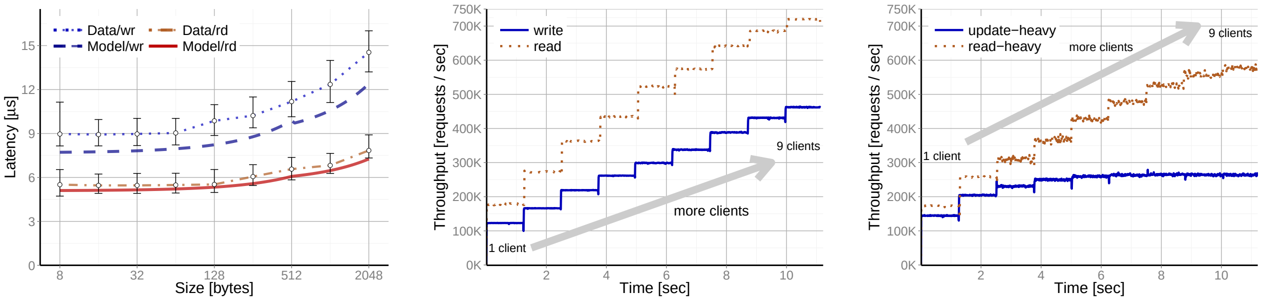
$$\rho_j(\kappa, k_1) = Pr(U_j \leq \kappa) = \sum_{u=0}^{\kappa} Pr(U_j = u).$$

Therefore, $\rho_j(q-1, P)$ expresses DARE's reliability. To compute h_j, we use the data from Table 2 under the assumption that all components are modeled by exponential LDMs. Figure 7 plots the reliability as a function of the group size. Of particular interest is the decrease in reliability when the group size increases from an even to an odd value. This is expected since the group has one more server, but the size of a quorum remains unchanged. Also, Figure 7 compares the reliability of our in-memory approach with the one achieved by stable storage; the disk AFRs are according to [46]. We observe that for a group size of 7, DARE can achieve higher reliability than disks with RAID-5 [13], while 11 servers are sufficient to overpass the reliability of disks with RAID-6 [47].

6. EVALUATION

We evaluate the performance of DARE in a practical setting. We use a 12-node InfiniBand cluster; each node has an Intel E5-2609 CPU clocked at 2.40GHz. The cluster is connected with a single switch using a single Mellanox QDR NIC (MT27500) at each node. Moreover, the nodes are running Linux, kernel version 3.12.18. DARE is implemented[1] in C and relies on two libraries: *libibverbs*, an implementation of the RDMA verbs for InfiniBand; and *libev*, a high-performance event loop. Each server runs an instance of

[1] DARE: `http://spcl.inf.ethz.ch/Research/Parallel_Programming/DARE`



(a) Write (/wr) and read (/rd) latency

(b) Write and read throughput

(c) Real-world inspired workloads

Figure 8: Evaluation of DARE

DARE; yet, each server is single-threaded. Finally, to compile the code, we used GCC version 4.8.2.

We consider a key-value store (KVS) as the client SM: Clients access data through 64-byte keys. Moreover, since clients send requests through UD, the size of a request is limited by the network's MTU (i.e., 4096 bytes). Henceforth, we only state the size of the data associated with a key. The structure of this section is as follows: first, we evaluate both the latency and the throughput of DARE; then we analyze the throughput for different workloads; finally, we compare the performance of DARE with other protocols and implementations, such as ZooKeeper [28].

Latency. DARE is designed as a high-performance RSM. Figure 8a shows the latency of both write and read requests (gets and puts in the context of a KVS). In the benchmark, a single client reads and writes objects of varying size to/from a group of five servers. Each measurement is repeated 1,000 times; the figure reports the median and both the 2nd and the 98th percentiles. DARE has a read latency of less than $8\mu s$; the write latency is, with $15\mu s$, slightly higher because of the higher complexity of log replication. Also, Figure 8a evaluates the model described in Section 3.3.3. Of particular interest is the difference between our model and the measured write latency. In practice, the small RDMA overhead ($\approx 0.3\mu s$) implies that a slight computational overhead may cause more than $\lfloor P/2 \rfloor$ servers to go through log replication; as a result, the write latency increases.

Throughput. We analyze DARE's throughput in a group of three servers that receives requests from up to nine clients. We calculate the throughput by sampling the number of answered requests in intervals of $10ms$. For 2048-byte requests, DARE achieves a maximum throughput of 760 MiB/s for reads and 470 MiB/s for writes. Furthermore, Figure 8b shows how the throughput (for 64-byte requests) increases with the number of clients. The reason for the increase is twofold: (1) DARE handles requests from different clients asynchronously; and (2) DARE batches requests together to reduce the latency. Therefore, with 9 clients, DARE answers over 720.000 read requests per seconds and over 460.000 write requests per second.

Further, we study the effect of a dynamic group membership on DARE's performance. In particular, Figure 9a shows the write throughput (for 64-byte requests) during a series of group reconfiguration scenarios. First, two servers are subsequently joining an already full group causing the size to increase; this implies that more servers are needed for a majority; hence, the throughput decreases. Also, note that the two joins cause a brief drop in throughput, but no unavailability. Then, the leader fails causing a short period of unavailability (i.e., around $30ms$) until a new leader is elected; this is followed by a brief drop in performance when the leader detects and removes the previously failed leader.

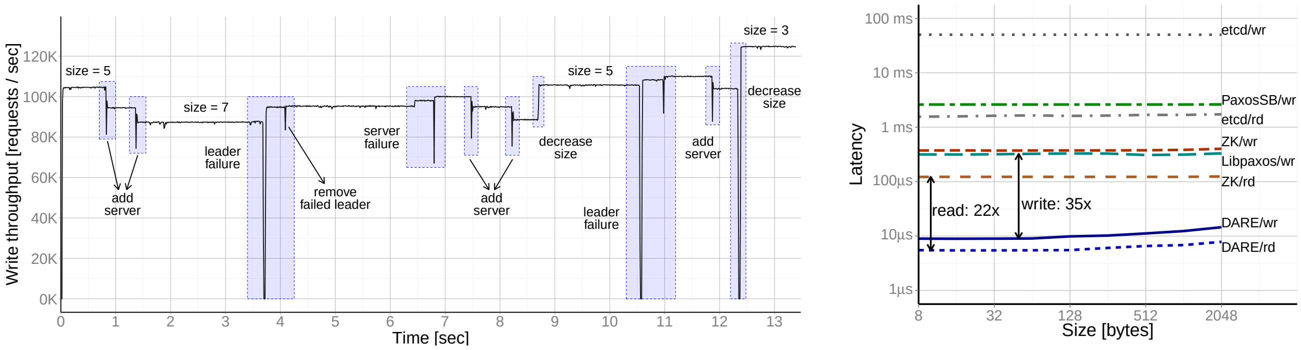
Next, a server fails; the leader handles the failure in two steps, both bringing an increase in throughput. First, it stops replicating log entries on the server since its QPs are inaccessible; second, after a number of failed attempts to send an heartbeat (we use two in our evaluation), the leader removes the server. The removal of the failed server is followed by two other servers joining the group; note that these joins have similar effects as the previous ones. Once the group is back to a full size, the leader receives a request to decrease the size, which implies an increase in throughput.

In the new configuration, the leader fails again having a similar effect as the previous leader failure. After a new leader is elected, another server joins the group. Finally, the new leader decreases the group size to three. However, this operations entails the removal of two servers, one of them being the leader. Thus, the group is shortly unavailable until a new leader is elected.

Workloads. The results of Figure 8b are valid for either read-only or write-only client SMs; yet, usually, this is not the case. Figure 8c shows DARE's throughput when applying real-world inspired workloads to a group of three servers. In particular, we use two workloads: read-heavy and update-heavy [14]. The read-heavy workload consist of 95% reads; it is representative for applications such as photo tagging. The update-heavy workload consist of 50% writes; it is representative for applications such as an advertisement log that records recent user activities. For read-heavy workload, the throughput slightly fluctuates when more than one client sends requests. This is because DARE ensures linearizable semantics [27]; in particular, the leader cannot answer read requests until it answers all the preceding write requests. Also, by interleaving read and write requests, DARE cannot take advantage of batching; thus, for update-heavy workloads, the throughput saturates faster.

DARE vs. other RSMs. We conclude DARE's evaluation by comparing it with other state of the art RSM protocols and implementations. In particular, we measure the latency of four applications: ZooKeeper (ZK), a service for coordinating processes of distributed applications [28]; `etcd`[2], a key-value store that uses Raft [45] for reliability;

[2] `etcd` version 0.4.6: `https://github.com/coreos/etcd`



(a)

(b)

Figure 9: (a) DARE's write throughput during group reconfiguration. (b) DARE and other RSM protocols: write (/wr) and read (/rd) latency.

and PaxosSB [31] and Libpaxos[3], both implementations of the Paxos protocol [33,34], providing support only for writes.

For each application, we implemented a benchmark that measures the request latency in a similar manner as for DARE—a single client sends requests of varying size to a group of five servers. All applications use TCP/IP for communication; to allow a fair comparison, we utilize TCP/IP over InfiniBand ("IP over IB"). Also, for the applications that rely on stable storage, we utilize a RamDisk (an in-memory filesystem) as storage location. Figure 9b shows the request latency of the four applications on our system. For ZooKeeper, we observe a minimal read latency of $\approx 120\mu s$; the put performance depends on the disk performance, and with a RamDisk, it oscillates around $380\mu s$. In the case of `etcd`, a read requests takes around $1.6ms$, while a write request takes almost $50ms$. For both PaxosSB and Libpaxos, we measured only the write latency. While PaxosSB answers a write requests in around $2.6ms$, Libpaxos, with around $320\mu s$, attains a write latency lower than ZooKeeper.

In addition, Figure 9b compares our protocol against all four applications. The latency of DARE is at least 22 times lower for read accesses and 35 times lower for write accesses. Also, we evaluate DARE's write throughput against ZooKeeper; in particular, we set up an experiment were 9 clients send requests to a group of three servers. With a write throughput of ≈ 270 MiB/s, ZooKeeper is around $1.7x$ below the performance achieved by DARE. Finally, we compare DARE with the Chubby lock service [10]. Yet, since we cannot evaluate it on our system, we use the latency measurement from the original paper [10]. Chubby achieves read latencies of under 1 ms and write latencies of around 5-10 ms. Thus, DARE's performance is more than two orders of magnitude higher.

7. RELATED WORK

The importance of practical resilient services sparked numerous proposals for protocols and implementations so that we can only discuss the most related approaches here. Starting from Lamport's original works on Paxos for solving consensus [33,34], several practical systems have been proposed [9,15,31]. Although the Paxos protocol ensures both safety and liveness, it is difficult both to understand and to implement. This led to a series of practical RSM protocols and implementations, that are at their heart similar to Paxos but simplify the protocol to foster understanding and implementability, such as Viewstamped Replication [37], Raft [45], ZooKeeper [28], and Chubby [10]. In general, DARE is different from these approaches in that it is designed for RDMA semantics instead of relying on messages. In this way, we achieve lowest latency in combination with high throughput which benefits client applications that cannot pipeline requests and require strong consistency.

Other attempts were made to increase the performance of key-value stores through RDMA; yet, they are using RDMA either as an optimization for message passing [30] or only for reading data [19,40]. DARE uses RDMA for both read and write requests. In addition, unique features of RDMA, such as queue pair management, allow us to design wait-free protocols for normal operation.

Furthermore, a series of optimizations were proposed for solving consensus, such as distributing the load by allowing multiple servers to answer requests and answering not-interfering requests out-of-order [35,38,42]. While DARE does not consider these optimizations, we believe they could be added to our design with moderate effort. Also, highly scalable systems, such as distributed hash tables, offer high throughput by relaxing the consistency model [17,36]. DARE's scalability is limited since it guarantees strong consistency for both reads and writes.

Finally, our fine-grained failure model, where remote CPUs can fail but their memory is still usable, is very similar to Disk Paxos [22]. Our work focuses on the fast implementation of this model with minimal overheads over realistic RDMA systems. Also, the idea of an in-memory RSM has been analyzed before [5,9,37]. Yet, most practical systems utilize disks for saving the state on stable storage [28,45], resulting in higher latency; while we can do this as well, we argue that it may not be necessary to achieve high reliability. Other systems reduce the latency by replacing disk storage with NVRAM; for example, CORFU uses a cluster of flash devices to offer a shared log abstraction [8]; however, it requires an RSM for reconfiguration.

8. DISCUSSION

Does it scale to thousands of servers? Leader-based RSM protocols are limited in scalability due to their dependency on consensus. Thus, DARE is intended to store meta-

[3] Libpaxos3: `https://bitbucket.org/sciascid/libpaxos.git`

data of more complex operations. A strategy to increase scalability would be partitioning data into multiple (reliable) DARE groups and delivering client requests through a routing mechanism [51]. Yet, routing requests that involve multiple groups would require consensus.

What about stable storage? Using stable storage, such as RAID systems, can further increase data reliability. Yet, waiting for requests to commit to disk would be too slow for normal latency-critical operations. While our protocol can easily be extended to facilitate this (one could even use additional InfiniBand disk storage or NVRAM targets directly), we currently only consider to periodically save the SM to disk. In case of a very unlikely catastrophic failure (more than half of the servers fail), one may still be able to retrieve from disk the slightly outdated SM. This is consistent with the behavior of most file-system caches today.

Can weaker consistency requirements be supported? DARE reads could be sped up significantly if any server could answer requests (not only the leader). This would also disencumber the leader who could process writes faster; yet, clients may read an outdated version of the data.

9. CONCLUSION

We demonstrate how modern RDMA networks can accelerate linearizable state machine replication by more than an order of magnitude compared to traditional TCP/IP-based protocols. Our RDMA-based RSM protocol and prototype DARE provide wait-free log replication and utilize RDMA features in innovative ways. For example, the queue pair connection management is extensively used to control log access during leader election and in failure situations.

An analysis of DARE's reliability shows that only five DARE servers are more reliable and 35x faster than storing the data on a RAID-5 system. This is mainly because an RDMA system is potentially more resilient—despite a fail-stop failure of the CPU or OS, a server is still accessible via RDMA. Our implementation allows operators to offer strongly consistent services in datacenters; we expect that RDMA-based protocols will quickly become a standard for performance-critical applications.

Acknowledgements. This work was supported by Microsoft Research through its Swiss Joint Research Centre. We thank our shepherd Jay Lofstead, the anonymous reviewers; Miguel Castro for his insightful feedback; Timo Schneider for helpful discussions; and the Systems Group at ETH Zurich for providing us the InfiniBand machine used for evaluation.

10. REFERENCES

[1] OpenFabrics Alliance (OFA). OpenFabrics Enterprise Distribution (OFED). `www.openfabrics.org`.

[2] NIST/SEMATECH e-Handbook of Statistical Methods. `http://www.itl.nist.gov/div898/handbook/`, 2012.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 1–14, New York, NY, USA, 2002. ACM.

[4] M. Aguilera. Stumbling over consensus research: Misunderstandings and issues. In B. Charron-Bost, F. Pedone, and A. Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 59–72. Springer Berlin Heidelberg, 2010.

[5] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[6] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[7] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[8] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: A shared log design for flash clusters. In *Proc. 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 1–1, Berkeley, CA, USA, 2012.

[9] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Reconstructing Paxos. *SIGACT News*, 34:2003, 2003.

[10] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006.

[11] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999.

[12] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[13] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[15] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[18] J. Domke, T. Hoefler, and S. Matsuoka. Fail-in-place Network Design: Interaction Between Topology, Routing Algorithm and Failures. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 597–608, Piscataway, NJ, USA, 2014. IEEE Press.

[19] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proc. 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, Apr. 1985.

[21] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–7, Berkeley, CA, USA, 2010.

[22] E. Gafni and L. Lamport. Disk Paxos. *Distrib. Comput.*, 16(1):1–20, Feb. 2003.

[23] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, New York, NY, USA, 2013. ACM.

[24] P. Gill, N. Jain, and N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, Aug. 2011.

[25] Global Scientific Information and Computing Center. Failure History of TSUBAME2.0 and TSUBAME2.5, 2014.

[26] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and Tolerating Heterogeneous Failures in Large Parallel Systems. In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 45:1–45:11, New York, NY, USA, 2011. ACM.

[27] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'10, pages 11–11, Berkeley, CA, USA, 2010.

[29] InfiniBand Trade Association. *InfiniBand Architecture Specification: Volume 1, Release 1.2.1.* 2007.

[30] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proc. 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011.

[31] J. Kirsch and Y. Amir. Paxos for System Builders: An Overview. In *Proc. 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 3:1–3:6, New York, NY, USA, 2008. ACM.

[32] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95 – 114, 1978.

[33] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[34] L. Lamport. Paxos Made Simple. *SIGACT News*, 32(4):51–58, Dec. 2001.

[35] L. Lamport. Generalized Consensus and Paxos, 2005.

[36] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table. In *Proc. 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 775–787, Washington, DC, USA, 2013.

[37] B. Liskov and J. Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[38] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008.

[39] C. D. Martino, F. Baccanico, Z. Kalbarczyk, R. Iyer, J. Fullop, and W. Kramer. Lessons Learned From the Analysis of System Failures at Petascale: The Case of Blue Waters. Jun 2014.

[40] C. Mitchell, Y. Geng, and J. Li. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proc. 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013.

[41] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010.

[42] I. Moraru, D. G. Andersen, and M. Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[43] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(01):95–107, 2001.

[44] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs. In *Proc. 6th Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM.

[45] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, USENIX ATC'14, Philadelphia, PA, June 2014.

[46] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007.

[47] J. S. Plank, A. L. Buchsbaum, and B. T. Vander Zanden. Minimum Density RAID-6 Codes. *Trans. Storage*, 6(4):16:1–16:22, June 2011.

[48] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and Modeling of a Non-blocking Checkpointing System. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 19:1–19:10, Los Alamitos, CA, USA, 2012.

[49] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[50] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007.

[51] P. Unterbrunner, G. Alonso, and D. Kossmann. High Availability, Elasticity, and Strong Consistency for Massively Parallel Scans over Relational Data. *The VLDB Journal*, 23(4):627–652, Aug. 2014.