APUS: Fast and Scalable PAXOS on RDMA

Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui

Department of Computer Science, University of Hong Kong

***Abstract*—State machine replication (SMR) enforces the same inputs for a program being replicated on several computers (or replicas), tolerating various failures. Recent SMR protocols have greatly improved the availability of server programs (e.g., `Redis`) that store important data. Unfortunately, existing SMR protocols incur prohibitive performance overhead on server programs, because their protocol latency is high when going through OS and TCP/IP layers, or this latency increases almost linearly to the number of concurrent requests or replicas. This paper presents APUS, a new RDMA-based SMR protocol. APUS achieves high performance by: (1) carefully dividing RDMA workloads among replicas; and (2) making all replicas receive consensus messages purely on local memory, just like making threads receive other threads' data efficiently on bare metal memory.**

APUS is the first SMR protocol that achieves low overhead on diverse real-world server programs. Evaluation on nine widely-used server programs (e.g., `Redis` and `MySQL`) shows that APUS incurred merely a mean overhead of 4.3% on response time and 4.2% on throughput. APUS's protocol latency outperforms a recent fastest RDMA-based SMR protocol by 4.9x. APUS's protocol latency scales almost constantly to the number of concurrent requests and replicas. APUS is deployable: all source code and raw evaluation results are released at http://github.com/icdcs17-p256/apus.

I. INTRODUCTION

State machine replication (SMR) runs the same program on replicas of computers and enforces same inputs for this program as long as a quorum (typically, majority) of replicas behave normally, tolerating various faults such as minor replicas failures. The core of SMR is a distributed consensus protocol (typically, PAXOS [52]). Recent SMR systems [48], [40], [31] have greatly improved the availability of server programs that serve online requests and store important data.

Unfortunately, existing PAXOS protocols incur high performance overhead on server programs. For efficiency, PAXOS typically assigns one replica as the leader to invoke consensus requests, and the other replicas as backups to agree on requests. To agree on an input, at least one message round-trip is required between the leader and a backup. Traditional PAXOS protocols, which are TCP or UDP based, incur high latency as their messages go through network stacks and OS kernels. This latency is acceptable for leader election [25], [6] or heavyweight transactions [31], [48], but undesirable for key-value store servers (e.g., `Redis` and `Memcached`).

Worse, PAXOS consensus latency increases almost linearly [38], [6] to the number of concurrent requests or replicas, because the leader receives linearly increasing consensus messages on TCP/IP. This increased latency aggravates program performance due to two trends. First, when serving concurrent requests, the performance of modern server programs scales well with the increasing number of CPU cores in a computer. Second, advanced SMR systems (e.g., Azure [50]) deploy more replicas (seven or nine) to handle replica failures or upgrades.

To improve PAXOS scalability, one approach is introducing parallel techniques such as multithreading [6], [23] or asynchronous IO [31], [75]. However, the high TCP/IP round-trip latency still exists, and synchronizations in these techniques frequently invoke expensive OS events such as context switches. We ran four parallel consensus protocols [6], [23], [31], [75] on 24-core computers with 40Gbps network, then we spawned 24 concurrent request connections. When the number of replicas increased from three to nine, three protocols' consensus latency increased by 105.4% to 168.3%, and 36.5% to 63.7% of the increase was in OS kernel.

Another scalability approach is maintaining multiple instances of PAXOS, including batching requests [78], partitioning program states [38], [22], [64], [14], [71], splitting consensus leadership [58], [23], and hierarchical replication architecture [47], [38]. These systems have improved throughput. However, the core of these systems, PAXOS, still has an unscalable consensus latency. Prior work [38], [64], [47] explicitly desires a PAXOS protocol with a more scalable consensus latency.

To improve PAXOS performance, Remote Direct Memory Access (RDMA) becomes a promising solution as it becomes common in datacenters. RDMA not only bypasses OS kernel, but it also provides dedicated network hardware to achieve a fast round-trip. For instance, the fastest RDMA operation allows a process to directly write to another process's memory on a remote replica without involving the remote replica's OS kernel or CPU. Such a RDMA round-trip takes only 3 μs [65].

However, due to the limited RDMA operation types, fully exploiting RDMA speed in software systems is widely considered challenging by the community [65], [45], [36], [73]. For instance, DARE [73], a recent fastest PAXOS protocol based on RDMA, presents a sole-leader,

two-round consensus: first, leader does RDMA writes of consensus requests on each remote replica; second, leader does RDMA writes on each replica to update a global variable that points to the latest consensus request. Unfortunately, such a global variable update *serializes* consensus requests: both DARE's and our evaluation (§VIII-B) showed that, as the number of concurrent requests increased, DARE's consensus latency increased quickly.

Our key observation is that we should carefully divide RDMA workloads among the leader and backups, especially in a scalability-sensitive context. Intuitively, we can let both leader and backups do RDMA writes directly on destination replicas' memory, and let all replicas poll their local memory to receive messages.

Although doing so will consume more CPU resources than a sole-leader protocol [73], it has two major benefits. First, both leader and backups participate in consensus, which makes it possible to reach consensus with only one round [61] without updating global variables. Second, all replicas now efficiently receive consensus messages on their local, bare metal memory. An analogy is threads receiving other threads' data via bare metal memory, a fast and scalable computation pattern.

We present APUS,[1] a new RDMA-based PAXOS protocol and its runtime system. In APUS, all replicas directly write to destination replicas' memory and poll messages from local memory to receive messages. Our runtime system handles other technical challenges such as atomic delivery of messages (§IV-A), efficient input logging (§VI-A), and failure recovery (§VI-B).

Similar to recent SMR systems [40], [31], APUS supports general, unmodified server programs. Within APUS, a program runs as is. APUS automatically deploys this program on replicas, intercepts inputs from the program's inbound socket calls (e.g., `recv()`), and invokes PAXOS to enforce same inputs across replicas.

We implemented APUS in Linux. APUS intercepts POSIX inbound socket calls (e.g., `accept()` and `recv()`) to coordinate inputs using the Infiniband RDMA architecture [1]. To help people inspect whether replicas run in sync, on top of APUS's PAXOS protocol, APUS provides an efficient network output checking tool. To recover or add replicas, APUS leverages CRIU [30] to automatically, efficiently checkpoint a program without perturbing consensus in normal case.

We compared APUS with five popular, open source consensus protocols, including four traditional ones (libPaxos [75], ZooKeeper [6], CRANE [31] and S-Paxos [23]), and an RDMA-based one (DARE [73]). We evaluated APUS on nine widely used or studied server programs, including 4 key-value stores (`Redis` [76], `Memcached` [63], `SSDB` [77], and `MongoDB` [66]), a SQL server `MySQL` [13], an anti-virus server `ClamAV` [28], a multimedia server `MediaTomb` [12], an LDAP server OpenLDAP [70], and a transactional database `Calvin` [78]. Evaluation shows that

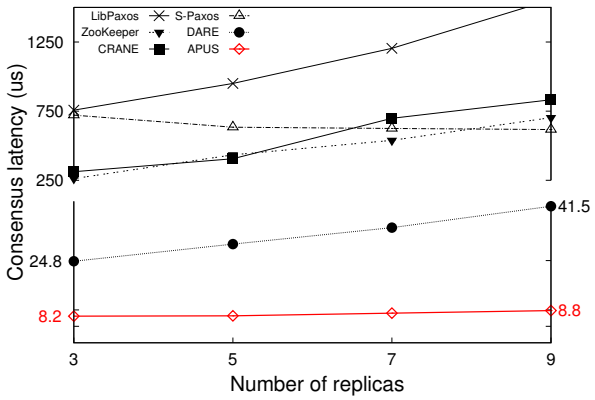

Fig. 1: *Comparing* APUS *to five existing consensus protocols.* All six protocols ran a client with 24 concurrent connections. The Y axis is broken to fit in all protocols.

1. APUS is fast and scalable. Figure 1 shows that APUS's consensus latency outperforms four traditional consensus protocols by at least 32.3x. Its consensus latency stays almost constant to the number of concurrent requests and replicas. Its consensus latency was faster than DARE by 4.9x (§VIII-B).
2. APUS achieves low overhead on diverse real-world server programs. We ran all nine server programs with seven replicas and compared to servers' own unreplicated executions at peak performance. In average, APUS incurred only 4.2% overhead on throughput and 4.3% on response time.
3. APUS is robust on replicas failures and packet losses. Its leader election latency was reasonable.

Our major contribution is the first RDMA-based PAXOS protocol that achieves low performance overhead on diverse server programs. APUS has the potential to largely improve both the scale and performance of many replication systems [50], [47], [31], [40], [22], [23]. For instance, Azure [50] deploys about seven replicas in each PAXOS group, and now it can enjoy much better performance even when serving more concurrent requests. Overall, a fast, scalable, and general SMR service, APUS can significantly promote the deployments of PAXOS and improve both the consistency and fault-tolerance of various systems within a datacenter.

The remaining of this paper is organized as follows. §II introduces PAXOS and RDMA background. §III gives an overview of APUS. §IV presents APUS's consensus protocol with its runtime system. §V describes the network output checking tool. §VI presents implementation details. §VII compares APUS protocol with DARE. §VIII presents evaluation results, §IX discusses related work, and §X concludes.

[1]We name our system after apus, one of the fastest birds.

II. BACKGROUND

A. PAXOS

PAXOS [79], [52], [51], [27], [53], [61] runs the same program and its data on a group of replicas and enforces a strongly consistent order of inputs across replicas. Because a consensus can be achieved as long as a majority of replicas agree, PAXOS is well known for tolerating various faults, including minor replica failures and packet losses. If the leader replica fails, PAXOS elects a new leader from the backups.

To handle replica failovers, standard PAXOS must log inputs in local stable storage. When a new input comes, the PAXOS leader writes this input in local stable storage. The leader then starts a new consensus round among replicas. A backup also writes the received consensus request in local storage if it agrees on this request. Since logging is done by each replica locally, it is scalable.

The consensus latency of PAXOS protocols is notoriously high and unscalable [6], [38]. As datacenters incorporate faster networking hardware and more CPU cores, traditional PAXOS protocols [75], [23], [31], [40], [6] are having fewer performance bottlenecks on network bandwidth and CPU resources.

However, software TCP/IP layers and OS kernels remain performance bottleneck [72]. To quantify this bottleneck, we evaluated four traditional consensus protocols on 24-core computers with 40Gbps network, and we spawned 24 concurrent consensus clients. When changing the replica group size from three to nine, although network and CPUs were not saturate, the consensus latency of three protocols drastically increased by 105.4% to 168.3% (Figure 1), and 36.5% to 63.7% of this increase was in OS kernel. If only one consensus client was spawned, the latency increase on the number of replicas was more gentle (Table II).

This evaluation also shows that both the number of concurrent requests and replicas make PAXOS latency increase drastically. This problem becomes worse as server programs tend to support more concurrent requests and advanced SMR systems such as Azure deploy seven to nine replicas [50] (for upgrades of multiple replicas).

B. RDMA

RDMA architectures (e.g., Infiniband [1] and RoCE [3]) become common within a datacenter due to its ultra low latency, high throughput, and its decreasing prices. The ultra low latency of RDMA not only comes from its kernel bypassing feature, but also its dedicated network stack implemented in hardware. Therefore, RDMA is considered the fastest kernel bypassing technique [45], [65], [73]; it is several times faster than software-only kernel bypassing techniques (e.g., DPDK [2] and Arrakis [72]).

RDMA has three operation types, from fast to slow: one-sided read/write operations, two sided send/recv operations, and IPoIB (IP over Infiniband). IPoIB can run unmodified socket programs, but it is several times slower than the other two types. A one-sided RDMA write operation can directly write from one replica's memory to a remote replica's memory without involving the remote OS kernel or CPU. Prior work [65] shows that one-sided operations are up to 2x faster than two-sided operations [46], so APUS uses one-sided operations (or "WRITE" in this paper). On a WRITE succeeds, the remote NIC sends an RDMA ACK to local NIC.

A one-sided RDMA communication between a local and a remote NIC needs a Queue Pair (QP), including a send queue and a receive queue. Each QP has a Completion Queue (CQ) to store ACKs. A QP belongs to a type of "XY": X can be R (reliable) or U (unreliable), and Y can be C (connected) or U (unconnected). HERD [45] shows that WRITEs on RC and UC OPs incur almost same latency, so APUS uses RC QPs.

Normally, to ensure a WRITE resides in remote memory, the local replica normally busily polls an ACK from the CQ before it proceeds (or *signaling*). Polling ACK is time consuming as it involves synchronization between the NICs on both sides of a CQ. We looked into the ACK pollings in a fastest RDMA-based PAXOS protocol DARE [73], and we found that, although DARE is highly optimized (its leader maintains one global CQ to receive all backups' ACKs in batches), polling ACKs is still time consuming: when the CQ was empty, it took 0.039$\sim$0.12 μs; when the CQ has one or more ACKs, it took 0.051$\sim$0.42 μs.

Fortunately, depending on protocol logic, one can do *selective signaling* [45]: it only checks for an ACK after pushing a number of WRITEs. Because APUS's protocol logic does not rely on RDMA ACKs, it just occasionally invokes selective signaling to clean up ACKs.

III. APUS OVERVIEW

APUS follows a typical SMR deployment scenario: it runs replicas of a server program in a datacenter. Replicas connect with each other using RDMA QPs. Client programs located in LAN or WAN. The APUS leader handles client requests and runs its RDMA-based protocol to coordinate inputs across replicas.

Figure 2 shows APUS's architecture. APUS intercepts a server program's inbound socket calls (e.g., `recv()`) using a Linux technique called LD_PRELOAD. APUS involves four key components: a PAXOS consensus protocol for input coordination (in short, the *coordinator*), an output checking protocol (the *checker*), a circular in-memory consensus log (the *log*), and a guard process that handles checkpointing and recovering a server's process state and file system state (the *guard*).

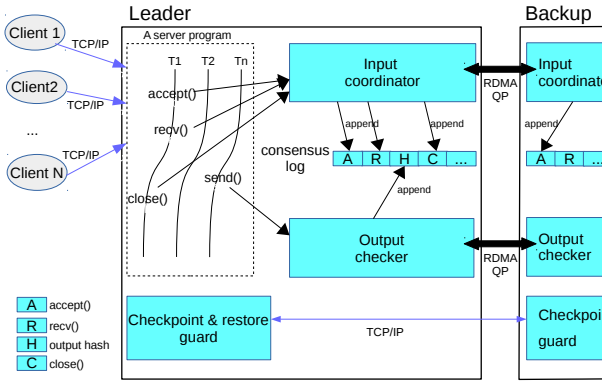

Fig. 2: APUS *Architecture (key components are in blue).*

The coordinator is invoked when a server program thread calls an inbound socket call to manage a client socket connection (e.g., `accept()` and `close()`) or to receive inputs from the connection (e.g., `recv()`). On the leader side, APUS executes the actual Libc socket call, extracts the returned value or inputs of this call, stores it in local SSD, appends a log entry to its local consensus log, and then invokes the coordinator for a new consensus request on "executing this socket call".

The coordinator runs a consensus algorithm (§IV), which WRITEs the local entry to backups' remote logs in parallel and polls the local log entry to wait quorum. When a quorum is reached, the leader thread simply finishes intercepting this call and continues with its server execution. As the leader's server threads execute more calls, APUS enforces the same consensus log and thus the same socket call sequence across replicas.

On each backup side, the coordinator uses a APUS internal thread called *follower* to poll its consensus log for new consensus requests. If the coordinator agrees the request, the follower stores the log entry in local SSD and then WRITEs a consensus reply to the remote leader's corresponding log entry. A backup does not need to intercept a server's socket calls because the follower will just follow the leader's consensus requests on executing what socket calls and then forward these calls to its local server program.

The output checker is occasionally invoked as the leader's server program executes outbound socket calls (e.g., `send()`). For every 1.5KB (MTU size) of accumulated outputs per connection, the checker unions the previous hash with current outputs and computes a new CRC64 hash. After a fixed number of hashes are generated, the checker then invokes consensus across replicas, which compares the hash at its global hash index on the leader side.

This output consensus is based on the input consensus algorithm (§IV) except that backups carry their hash at the same hash index back to the leader. For this particular output consensus, the leader first waits quorum. It then waits for a few μs in order to collect more remote hashes. It then compares remote hashes it has.

If a hash divergence is detected, the leader optionally invokes the local guard to forward a "rollback" command to the diverged replica's guard. The diverged replica's guard then rolls back and restores the server program to a latest checkpoint before the last successful output check (§V). The replica then restores and re-executes socket calls to catch up. As output hash generations are fast and an output consensus is invoked occasionally, we did not observe performance impact on the checker.

IV. THE RDMA-BASED PAXOS PROTOCOL

This section presents APUS's consensus protocol with its runtime system, including the RDMA-based consensus algorithm in normal case (§IV-A), handling concurrent connections (§IV-B), leader election (§IV-C), and reliability guarantees (§IV-D).

A. Normal Case Algorithm

APUS' input consensus protocol contains three roles (§III). First, the PAXOS consensus log. Second, a server program thread on the leader replica (in short, a leader thread) which invokes a consensus request. For efficiency, APUS lets leader threads concurrently invoke consensus requests when they call inbound socket calls (e.g., `recv()`). Third, a backup's APUS internal follower thread (§III) which processes consensus requests.

```
struct log_entry_t {
  consensus_ack reply[MAX]; // Per replica consensus reply.
  viewstamp_t vs;
  viewstamp_t last_committed;
  int node_id;
  viewstamp_t conn_vs; // client connection ID.
  int call_type; // socket call type.
  size_t data_sz; // data size in the call.
  char data[0]; // data, with a canary value in the last byte.
} log_entry;
```

Fig. 3: APUS*'s log entry for each socket call.*

Figure 3 shows the format of a log entry in APUS's consensus log. Most fields are regular as those in a typical PAXOS protocol [61] except three ones: the `reply` array, the client connection ID `conn_vs`, and the type ID of a socket call `call_type`. The `reply` array is for backups to WRITE their consensus replies to the leader. The `conn_vs` is for identifying which connection this socket call belongs to (see IV-B). The `call_type` identifies four types of socket calls in APUS: the `accept()` type (e.g., `accept()`), the `recv()` type (e.g., `recv()` and `read()`), the `send()` type (e.g., `send()`), and the `close()` type (e.g., `close()`).

Figure 4 shows the input consensus algorithm. Suppose a leader thread invokes a consensus request when it calls a socket call with the `recv()` type. A consensus request includes four steps. The first step (**L1**) is executing the actual Libc socket call, because APUS needs to

get the actual return values or received data bytes of this call and then replicates them in remote replicas' logs.

The second step (**L2**) is local preparation, including assigning a global viewstamp to locate this entry in the consensus log, building a log entry structure for this call, and writing this entry to a local fast storage (§VI-A).

Third, each leader thread concurrently invokes a consensus via the third step (**L3**): WRITE the log entry to remote backups. This step is thread-safe because each leader thread works on its own entry and backups' corresponding remote entries. An **L3** WRITE immediately returns after copying the entry to its local QP between the leader and each backup. In our evaluation, copying a log entry to local QP took no more than 0.2 μs. Therefore, the WRITEs to all backups are done almost in parallel (all **L3** arrows in Figure 3).

The fourth step (**L4**) is the leader thread polling on its `reply` field in its local log entry for backups' consensus replies. Once consensus is reached, the leader thread finishes intercepting this `recv()` socket call and continues with its server application logic.

On a backup side, one tricky synchronization issue is that an efficient way is needed to make the leader's RDMA WRITEs and backups' polls atomic. For instance, while a leader thread is doing a WRITE on `vs` to a remote backup, the backup's follower thread may be reading `vs` concurrently, causing a corrupted read value.

To address this issue, one existing approach [35], [45] leverages the left-to-right ordering of RDMA WRITEs and puts a special non-zero variable at the end of a fix-sized log entry because they mainly handle key-value stores with fixed value length. As long as this variable is non-zero, the RDMA WRITE ordering guarantees that the log entry WRITE is complete. However, because APUS aims to support general server programs with largely variant received data lengths, this approach cannot be applied in APUS.

Another approach is using atomic primitives provided by RDMA hardware, but a prior evaluation [80] has shown that RDMA atomic primitives are much slower than normal RDMA WRITEs and local memory reads.

APUS tackles this issue by adding a canary value after the actual `data` array. Because APUS uses a QP with the type of RC (reliable connection) (§II), the follower always first checks the canary value according to `data_size` and then starts a standard PAXOS consensus reply decision [61]. Our efficient, synchronization-free approach guarantees that the follower always reads a complete log entry.

A follower thread in a backup replica polls from the latest un-agreed log entry, and it does three steps to agree on consensus requests in order without allowing any consensus gaps [61]. First (**B1**), it does a regular PAXOS view ID checking to see whether the leader is

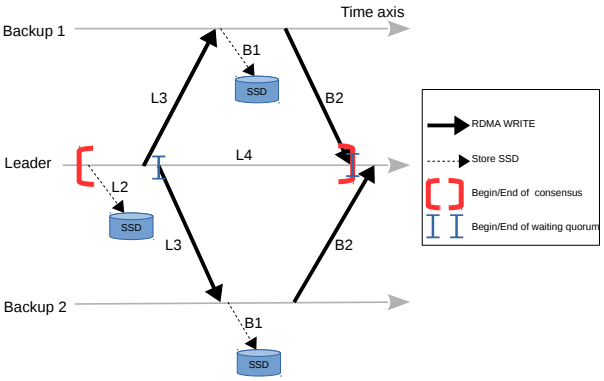

Fig. 4: APUS *consensus algorithm in normal case.*

up-to-date, it then stores the log entry in its local SSD. Second (**B2**), it does a WRITE to send back a consensus reply to the leader's `reply` array element according to its own node ID. Backups perform these two steps in parallel (see Figure 3).

Third (**B3**, not shown in Figure 4), the follower does a regular PAXOS check on `last_committed` and executes all socket calls that it has not executed before this viewstamp. It then "executes" each log entry by sending the socket calls to the local server program.

B. Handling Concurrent Connections

Unlike traditional PAXOS protocols which mainly handle single-threaded programs due to the deterministic state machine assumption in SMR, APUS aims to support both single-threaded as well as multithreaded server programs running on multi-core machines. Therefore, a strongly consistent mechanism is needed to map every concurrent client connection on the leader and to its corresponding connection on backups. A naive approach could be matching a leader connection's socket descriptor to the same one on a backup, but backups' servers may return nondeterministic descriptors due contentions on systems resources.

Fortunately, PAXOS already makes viewstamps [61] of socket calls strongly consistent across replicas. For TCP connections, APUS adds the `conn_vs` field, the viewstamp of the the first socket call in each connection (i.e., `accept()`) as the connection ID for log entries. Then, APUS maintains a hash map on each local replica to map this connection ID to local socket descriptors.

C. Leader Election

Leader election on RDMA raises a main issue: because backups do not communicate frequently with each other in normal case, thus a backup does not know the remote memory locations where the other backups are polling. Writing to a wrong remote memory location may cause the other backups to miss all leader election messages. A recent system [73] establishes an extra control QP to handle leader election, complicating deployments.

APUS addresses this issue with a simple, clean design. It runs a leader election with the same consensus log and the same QP. In normal case, the leader does WRITEs to remote logs as heartbeats with a period of T. Each consensus log maintains a control data structure called `elect[MAX]`, one element for each replica. Normal case operations and heartbeats use the other parts of the consensus log but leave this `elect` array alone. Once backups miss heartbeats from the leader for 3*T, they close the outdated leader's QPs and elect a new leader by making follower threads poll the `elect` array.

Backups start a standard PAXOS leader election algorithm [61] with three steps. Each replica writes to its own `elect` element at remote replicas. First, backups propose a new view with a standard two-round PAXOS consensus [51] by including both the view and the index of the latest log entry. The other backups also propose their views and poll on this array in order to follow other proposals or confirm itself as the winner. The backup whose log is more up-to-date will win. A log is more up-to-date if its latest entry has either a higher view or the same view but a higher index.

Second, the winner proposes itself as a leader candidate using this array, another two-round PAXOS consensus. Third, after the second step reaches a quorum, the new leader notifies remote replicas itself as the new leader and it starts to WRITE periodic heartbeats. Overall, APUS safely avoids multiple "leaders" to corrupt consensus logs, because only one leader is elected in each view, and backups always close an outdated leader's QPs before electing a new leader.

D. Reliability Guarantee

To minimize protocol-level bugs, APUS's protocol derives from a viewstamp-based protocol [61]. We made this design choice because PAXOS is notoriously difficult to understand [68], [51], [52], [79], implement [27], [61], or verify [82], [39]. Deriving from a viewstamp-based protocol [61] helps us incorporate these readily mature understanding, engineering experience, and the theoretically verified safety rules into our protocol.

Although APUS's PAXOS protocol works on a RDMA network, the reliability of APUS does not rely on a lossless RDMA (e.g., network devices can fail). APUS still provides same fault-tolerance guarantee as the standard PAXOS model, where a stable storage exists, but replicas may fail, and server programs may crash.

V. OUTPUT CHECKING PROTOCOL

Most server programs are multithreaded and they may run into nondeterminism (e.g., concurrency errors [56]), which may cause replicas to diverge. APUS provides a fast output checking protocol as a best-effort tool for program developers to detect replica divergence. We found this tool useful (§VIII-D). If divergence is detected, developers can manually restore them with APUS checkpoints (§VI-B). A fully automatic approach can be running a deterministic multithreading tool [32], [55] within APUS [31], and we leave it for future work.

A main technical challenge for comparing outputs across replicas is that network outputs and their physical timings are miscellaneous. For example, when we ran `Redis` simply on pure SET workloads, we found that different replicas reply the numbers of "OK" replies for SET operations randomly: one replica may send four of them in one `send()` call, while another replica may only send one of them in each `send()` call. Therefore, comparing outputs on each `send()` call among replicas may not only yield wrong results, but may slow down server programs among replicas.

To tackle this challenge, APUS presents a bucket-based hash computation mechanism. When a server calls a `send()` call, APUS puts the sent bytes into a local, per-connection bucket with 1.5KB (MTU size). Whenever a bucket is full, APUS computes a new CRC64 hash on a union of the current hash and this bucket. Such a hash computation mechanism encodes accumulated network outputs. Then, for every T_{comp} (by default, 10K in APUS) local hash values are generated, APUS invokes the output checking protocol once to check this hash across replicas. Because this protocol is invoked rarely, we did not observed its performance impact.

To compare a hash across replicas, APUS's output checking protocol runs the same as the input coordination protocol (§IV-A) except that the follower thread on each backup replica carries this hash value in the `reply` written back into the leader's corresponding log entry.

VI. IMPLEMENTATION DETAILS

A. Parallel Input Logging

To handle replica fail-overs, a standard PAXOS protocol should provide a persistent input logging storage. APUS uses the PAXOS viewstamp of each input as key and its input data as value. APUS stores this key-value pair in Berkeley DB (BDB) with a BTree access method [20], as our evaluation this method fastest.

However, if more inputs are inserted, the BTree height will increase, which will cause the key-value insertion latency to largely increase.

To handle this issue, we implemented a parallel logging approach [21]: instead of maintaining a single BDB store, we maintain an array of BDB stores. We use an index to indicate the current active store and insert new inputs. Once the number of insertions reach a threshold, we move the index to the next empty store in the array and recycle preceding stores. This implementation made APUS logging latency efficient: 2.8$\sim$8.7 μs (§VIII-C).

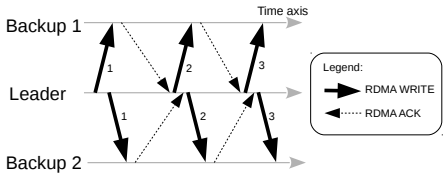

Fig. 5: ***DARE's RDMA-based protocol.*** **It is a sole-leader, two-round protocol with three steps: (1) the leader WRITEs a consensus request to all backups' consensus logs and waits for ACKs to check if they succeed; (2) for the successful backups in (1), leader does WRITEs to update tail pointer of their consensus logs; and (3) on receiving a majority of ACKs in (2), a consensus is reached, leader does WRITEs to notify backups.**

B. Checkpoint and Restore

We proactively design APUS's checkpoint mechanism to incur little performance impact in normal case. A checkpoint operation is invoked periodically in one backup replica, so the leader and other backups can still reach consensus on new inputs rapidly.

A guard process is running on each replica to checkpoint and restore the local server program. It assigns one backup replica's guard to checkpoint the local server program's process state and file system state of current working directory within a one-minute duration.

Such a checkpoint operation and its duration are not sensitive to normal case performance because the other backups can still reach quorum rapidly. Each checkpoint is associate with a last committed socket call viewstamp of the server program. After each checkpoint, the backup dispatches the checkpoint zip file to the other replicas.

Specifically, APUS leverages CRIU [30], a popular, open source tool, to checkpoint a server program's process state (e.g., CPU registers and memory). Since CRIU does not support checkpointing RDMA connections, APUS's guard first sends a "close RDMA QP" request to an APUS internal thread, lets this thread closes all remote RDMA QPs, and then invokes CRIU.

VII. COMPARING APUS WITH DARE

DARE [73] (Figure 5) is one of the fastest PAXOS protocols. It is RDMA-based and most relevant to APUS. DARE is designed for both a small number of concurrent requests and replicas. DARE follows a sole-leader manner with two-rounds: first, leader does RDMA writes of consensus requests on each replica; second, leader does RDMA writes on each replica to update a global variable that points to the latest request (tail of consensus log) in each backup. DARE's both rounds are essential in case a backup becomes a new leader.

DARE is not designed to work with large concurrent requests or replicas for two reasons. First, its second round needs to update a global variable for each replica, which *serializes* concurrent consensus requests, an essential feature for PAXOS performance [61]. For instance, two ongoing DARE consensus requests may interleave and cause the variable to point to an earlier request. Therefore, the consensus of DARE's new requests can not start until the consensus of prior requests finish, confirmed in both DARE's and our evaluation (§VIII-B). Second, a sole-leader protocol is difficult to add a stable storage or checkpoint design. Therefore, DARE lacks durability (§II), another essential PAXOS feature [57].

Overall, APUS differs from DARE in three aspects. First, APUS's protocol has only one RDMA round (Figure 4), and DARE has two rounds. Second, APUS scales well on concurrent client connections and replicas (§VIII-B), and DARE [73] mentioned that their design choices do not include scalability. Third, although APUS is a durable protocol and DARE is a volatile one, APUS is faster than DARE by 4.9x in average. §VIII-B compares APUS and DARE performance in detail.

VIII. EVALUATION

Our evaluation machines include nine RDMA-enabled, Dell R430 servers as PAXOS replicas. Each server has Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. All NICs are Mellanox ConnectX-3 Pro 40Gbps connected with Infiniband [1]. The `ping` latency between every two replicas is 84 μs (the IPoIB round-trip latency).

We compared APUS with five open source consensus protocols, including four traditional ones (libPaxos [75], ZooKeeper [6], CRANE [31] and S-Paxos [23]) and an RDMA-based one (DARE [73]). S-Paxos is designed to achieve scalable throughput on more replicas.

Program	Benchmark	Workload/input description
`ClamAV`	clamscan [8]	Files in `/lib` from a replica
`MediaTomb`	ApacheBench [11]	Transcoding videos
`Memcached`	mcperf [7]	50% set, 50% get operations
`MongoDB`	YCSB [10]	Insert operations
`MySQL`	Sysbench [9]	SQL transactions
OpenLDAP	Self	LDAP queries
`Redis`	Self	50% set, 50% get operations
`SSDB`	Self	Eleven operation types
`Calvin`	Self	SQL transactions

TABLE I: ***Benchmarks and workloads.*** **"Self" in the Benchmark column means we used a program's own benchmark.**

We evaluated APUS on nine widely used or studied server programs, including 4 key-value stores `Redis`, `Memcached`, `SSDB`, `MongoDB`; `MySQL`, a SQL server; `ClamAV`, an anti-virus server that scans files and delete malicious ones; `MediaTomb`, a multimedia storage server that stores and transcodes video and audio files; OpenLDAP, an LDAP server; `Calvin`, a widely studied transactional database system. All these programs are multithreaded except `Redis` (but it can serve concurrent requests via Libevent). Table I shows the benchmarks and workloads. These servers all update or store important data and files, thus PAXOS' strong fault-tolerance is especially attractive to these programs.

The rest of this section focuses on these questions:

§VIII-A: How does APUS's performance compare to traditional protocols?

§VIII-B: How does APUS compare to DARE?

§VIII-C: What is the performance overhead of running APUS with server programs? How does it scale with the number of concurrent requests?

§VIII-D: How fast is APUS on handling checkpoints and electing a new leader?

A. Comparing with traditional consensus protocols

We ran APUS and four traditional consensus protocols using their own client programs or popular client programs with 100K requests of similar sizes. For each protocol, we ran a client with 24 concurrent connections on a 24-core machine located in LAN, and we used up to nine replicas. Both 24 concurrent connections and nine replicas are common high values [6], [31], [40], [73].

All four traditional protocols were run on IPoIB (§II-B). Figure 1 shows that the consensus latency of three traditional protocols increased almost linearly to the number of replicas (except S-Paxos). S-Paxos batches requests from replicas and invokes consensus when the batch is full. More replicas can take shorter time to form a batch, so S-Paxos incurred a slightly better consensus latency with more replicas. Nevertheless, its latency was always over 600 μs. APUS's consensus latency outperforms these four protocols by at least 32.3x.

Proto-#Rep	Latency	First	Major	Process	Sys
libPaxos-3	81.6	74.0	81.6	2.5	5.1
libPaxos-9	208.3	145.0	208.3	12.0	51.3
ZooKeeper-3	99.0	67.0	99.0	0.84	31.2
ZooKeeper-9	129.0	76.0	128.0	3.6	49.4
CRANE-3	78.0	69.0	69.0	13.0	0
CRANE-9	148.0	83.0	142.0	30.0	35.0
S-Paxos-3	865.1	846.0	846.0	20.0	0
S-Paxos-9	739.1	545.0	731.0	35.0	159.1

TABLE II: ***Performance breakdown of traditional protocols on leader with only one client.*** **The "Proto-#Rep" column is the protocol name and replica group size; "Latency" is the consensus latency; "First" is the latency of leader's first received consensus reply; "Major" is the latency of leader's consensus; "Process" is leader's time spent in processing all replies; and "Sys" is leader's time spent in systems (OS kernel, network libraries, and JVM) between the "First" and the "Major" reply. All times are in μs.**

To find the scalability bottleneck in traditional protocols, we spawned only one client and broke down their consensus latency on leader (Table II). From three to nine replicas, the consensus latency (the "Latency" column) of these protocols increased more gently than that on 24 concurrent clients. For instance, when the number of replicas increased from three to nine, ZooKeeper latency increased by 30.3% with one client; this latency increased by 168.3% with 24 clients (Figure 1). This indicates that concurrent consensus requests are the major scalability bottleneck for these protocols.

Specifically, three protocols had scalable latency on the arrival of their first consensus reply (the "First" column), which implies that network is not saturate. libPaxos is an exception because its two-round protocol consumed much bandwidth. However, on leader, there is a big gap between the arrival of the first consensus reply and the "majority" reply (the "Major" column). Given that the replies' CPU processing time was small (the "Process" column), we can see that various systems layers, including OS kernels, network libraries, and language runtimes (e.g., JVM), are another major scalable bottleneck (the "Sys" column). This indicates that RDMA is useful on bypassing systems layers.

Note that both CRANE and S-Paxos's leader handles consensus replies rapidly, so they two had same "First" and "Major" arrival times (i.e., "Sys" times were 0).

B. Comparing with DARE

We compared APUS and DARE [73] on key-value stores. APUS ran a popular one, `Redis`; DARE supported a 335-line, RDMA-based key-value server written by their authors. Figure 6 shows APUS and DARE's consensus latency on variant concurrent clients. Both APUS and DARE ran seven replicas with randomly arriving, update-heavy (50% SETs and 50% GETs) and read-heavy (10% SETs and 90% GETs) workloads. DARE performance on two workloads were different because it handles GETs with only one consensus round [73]. APUS handles all requests with the same protocol. When there was only one client, APUS was slightly slower than DARE because DARE does not store inputs persistently. In average, APUS's consensus latency was faster than DARE by 4.9x. We think this owes to two reasons.

First, APUS is a one-round protocol and DARE is a two-round protocol (for SETs), so DARE's "actual-consensus" time was 53.2% higher than APUS. Even using read-heavy workloads (DARE uses one-round for GETs) with APUS, APUS's actual consensus time was still slightly faster than DARE's on more than six clients, because APUS avoids expensive ACK pollings (§II-B).

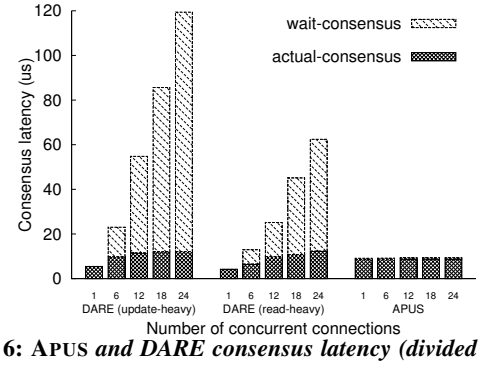

Fig. 6: APUS ***and DARE consensus latency (divided into two parts) on variant concurrent clients.*** **"Wait-consensus" is the time an input request spent on waiting consensus to start. "Actual-consensus" is the time spent on running consensus.**

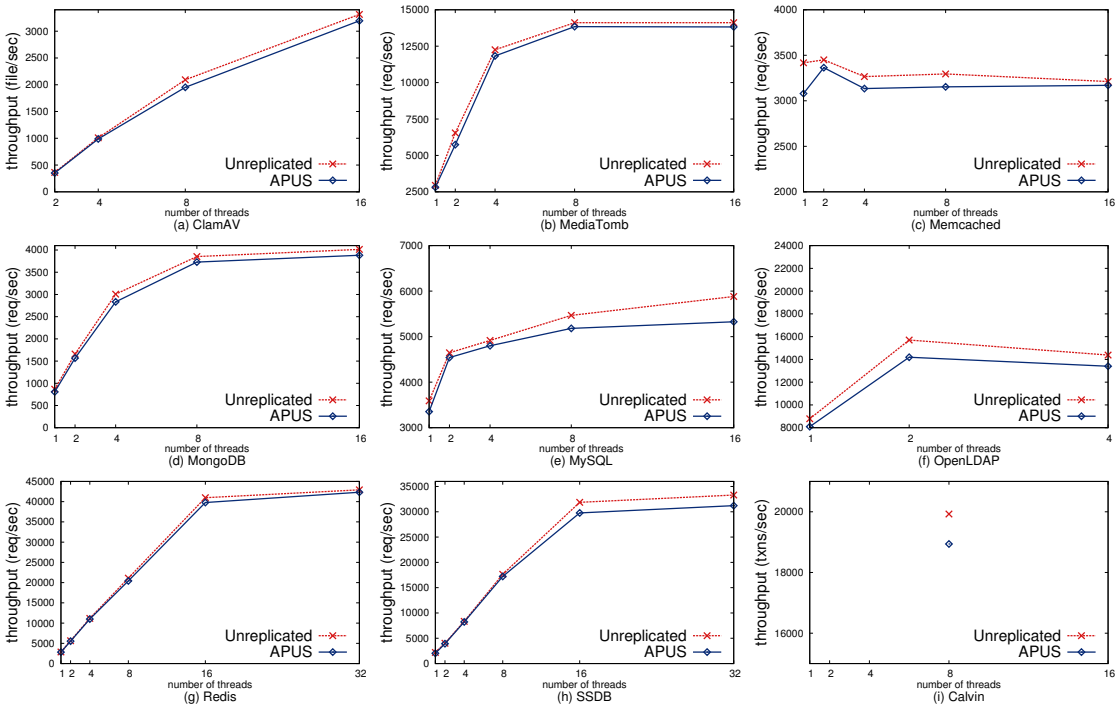

Fig. 7: APUS *throughput compared to the unreplicated execution.*

Second, DARE's second consensus round updates a global variable for each backup and serializes consensus requests (§VII). Although DARE mitigates this limitation by batching same SET or GET types, randomly arriving requests often break batches, causing a large "wait-consensus" time (a new batch can not start consensus until prior batches reach consensus). DARE evaluation [73] confirmed such a high wait duration: with three replicas and nine concurrent clients, DARE's throughput on real-world inspired workloads (50% SET and 50% GET arriving randomly) was 43.5% lower than that on 100% SET workloads. APUS's "wait-consensus" was almost 0 as it enables concurrent consensus requests (§IV-A).

DARE evaluation also showed that, with 100% SET workloads, its throughput decreased by 30.1% when the number of replicas increased from three to seven. We reproduced a similar result: we used the same workloads and 24 concurrent clients, and we varied the number of replicas from three to nine. We found that APUS consensus latency increased merely by 7.3% and DARE increased by 67.3% (shown in Figure 1).

Overall, these results indicate that DARE is better on smaller number of concurrent requests and replicas (e.g., leader election [25], [6]), and APUS is better on larger number of concurrent requests or replicas (e.g., replicating server programs [40], [31]).

C. Performance Overhead

To stress APUS, we used nine replicas to run all nine server programs without modifying them. We used up to 32 concurrent client connections (most evaluated programs reached peak throughput at 16), and then we measured mean response time and throughput in 50 runs.

We turned on output checking (§V) and didn't observe an performance impact. Only two programs (`MySQL` and OpenLDAP) have different output hashes caused by physical times (an approach [61] can be leveraged to enforce same physical times across replicas).

Figure 7 shows APUS's throughput. For `Calvin`, we only collected the 8-thread result because `Calvin` uses this constant thread count in their code to serve client requests. Compared to these server programs' unreplicated executions, APUS merely incurred a mean throughput overhead of 4.2% (note that in Figure 7, the Y-axises of most programs start from a large number). As the number of threads increases, all programs' unreplicated executions got a performance improvement except `Memcached`. Prior work [40] also showed that `Memcached` itself scaled poorly. Overall, APUS scaled as well as unreplicated executions on concurrent requests.

To understand APUS's performance overhead, we broke down its consensus latency on the leader replica. Table III shows these statistics per 10K requests, 8 or max (if less than 8) threads. According to the consensus algorithm in Figure 4, for each socket call, APUS's leader does an "L2": SSD write (the "SSD time" column in Table III) and an "L4": quorum waiting phase (the "quorum time" column). L4 implies backups' performance because each backup stores consensus requests in local SSD and then WRITEs a reply to leader.

Program	# Calls	Input	SSD time	Quorum time
ClamAV	30,000	37.0	7.9 μs	10.9 μs
MediaTomb	30,000	140.0	5.0 μs	17.4 μs
Memcached	10,016	38.0	4.9 μs	7.0 μs
MongoDB	10,376	490.6	7.8 μs	9.2 μs
MySQL	10,009	28.8	5.1 μs	7.8 μs
OpenLDAP	10,016	27.3	5.5 μs	6.4 μs
Redis	10,016	40.5	2.8 μs	6.0 μs
SSDB	10,016	47.0	3.0 μs	6.2 μs
Calvin	10,002	128.0	8.7 μs	10.8 μs

TABLE III: ***Leader's input consensus events per 10K requests, 8 threads.*** **The "# Calls" column means the number of socket calls that went through APUS input consensus; "Input" means average bytes of a server's inputs received in these calls; "SSD time" means the average time spent on storing these calls to stable storage; and "Quorum time" means the average time spent on waiting quorum.**

By adding the last two columns in Table III, APUS consensus latency took only 8.8 μs to 22.4 μs. This small consensus latency makes it achieve reasonable throughputs in Figure 7. Moreover, Figure 6 and Table III indicate that APUS incurred low overhead on server programs' response time. APUS's mean response time overhead was 4.3%. Due to this low overhead, APUS did not need to add read-only optimization [48]. As space is limited, APUS response time overhead figure is here [16].

D. Checkpoint and Recovery

We ran same performance benchmark as in §VIII-C and measured programs' checkpoint timecost. Each program checkpoint operation (§VI-B) cost 0.12s to 11.6s depending on the amount of modified memory and files since a program's last checkpoint. `ClamAV` incurred the largest checkpoint time (11.6s) because it loaded and scanned files in the `/lib` directory. Checkpoints did not affect APUS performance in normal case because they were done on only one backup. Leader and other backups still formed majority and reached consensus rapidly.

To evaluate APUS's PAXOS robustness, we ran APUS with `Redis`. We manually killed one backup and then modified another backup's code to drop all its consensus reply messages. We did not observe a performance change, as other seven replicas still reach consensus. We then manually killed the APUS leader and measured the latency of our leader election approach (§IV-C), shown in Table IV. Because PAXOS leader election is rarely invoked, we considered this election latency reasonable.

# Replicas	3	5	7	9
Election latency (μs)	10.7	12.0	12.8	13.5

TABLE IV: APUS's ***latency on on leader election.***

IX. RELATED WORK

Software-based consensus. Various PAXOS algorithms [61], [52], [51], [79], [67] and implementations [27], [61], [25], [31] exist. PAXOS is notoriously difficult to be fast and scalable [64], [47], [38], so server programs carry a weaker asynchronous replication approach (e.g., `Redis` [76]). Consensus is essential in datacenters [83], [41], [5] and worldwide distributed systems [29], [58], so much work is done to improve specific aspects, including commutativity [67], [59], understandability [68], [52], and verification [82], [39]. PAXOS is extended to tolerate byzantine faults [24], [62], [18], [26], [49], [60], [17], [15] and hardware faults [19].

Three SMR systems, Eve [48], Rex [40], and CRANE [31], use traditional PAXOS protocols to improve the availability of server programs with modest overhead. None of these systems has evaluated their response time overhead on key-value servers, which are extremely sensitive on latency. APUS is the first SMR system that achieves low overhead on both response time and throughput on real-world key-value servers.

Hardware- or Network- assisted consensus. Recent systems [43], [34], [42], [74], [54] leverage augmented network hardware or topology to improve PAXOS consensus latency. Three systems [43], [34], [42] implement consensus protocols in hardware devices (e.g., switches). "Consensus in a Box" [43] implemented ZooKeeper's protocol in FPGA. These systems reported similar performance as DARE and they are suitable to maintain compact metadata (e.g., leader election). Prior work [54] pointed out that these systems' programmable hardware are not suitable to store large amount of replicated states (e.g., server programs' continuously arriving inputs).

Speculative Paxos [74] and NOPaxos [54] use the datacenter topology to order requests, so they can eliminate consensus rounds if packets are not reordered or lost. If packets are lost or reordered, they invoke consensus to rescue. These two systems are not designed for scalability because when the number of concurrent requests or replicas increase, the probability of reordered or lost packets will increase. Moreover, these two systems' consensus modules are TCP/UDP-based and incur high consensus latency, which APUS can help.

RDMA-based systems. RDMA techniques have been implemented in various architectures, including Infiniband [1], RoCE [3], and iWRAP [4]. RDMA are used to speed up high performance computing [37], key-value stores [65], [45], [35], [44], transactional systems [80], [36], [46], and file systems [81]. For instance, FaRM [35] leverages RDMA to build a fast DHT. FaRM works in a primary-backup manner [33], [69]. PAXOS provides better availability than primary-backup. These systems use RDMA to speed up system-specific aspects, so they are complementary to APUS.

X. CONCLUSION

We have presented APUS, a new RDMA-based PAXOS protocol and its runtime system. Evaluation on five consensus protocols and nine widely used programs shows that APUS is fast, scalable, and deployable. It has the potential to greatly promote the deployments of PAXOS and improve the reliability of real-world programs.

REFERENCES

[1] An Introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf.

[2] Data Plane Development Kit (DPDK). http://dpdk.org/.

[3] Mellanox Products: RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.

[4] RDMA iWARP. http://www.chelsio.com/nic/rdma-iwarp/.

[5] Why the data center needs an operating system. http://radar.oreilly.com/2014/12/why-the-data-center-needs-an-operating-system.html.

[6] ZooKeeper. https://zookeeper.apache.org/.

[7] A tool for measuring memcached server performance. https://github.com/twitter/twemperf, 2004.

[8] clamscan - scan files and directories for viruses. http://linux.die.net/man/1/clamscan, 2004.

[9] SysBench: a system performance benchmark. http://sysbench.sourceforge.net, 2004.

[10] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2004.

[11] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html, 2014.

[12] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[13] MySQL Database. http://www.mysql.com/, 2014.

[14] H. Abu-Libdeh, R. Van Renesse, and Y. Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 12. ACM, 2013.

[15] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.

[16] Apus reponse time overhead on nine server programs. http://github.com/icdcs17-p256/extra-figures/latency-overhead.pdf, 2016.

[17] F. Araujo, R. Barbosa, and A. Casimiro. Replication for dependability on virtualized cloud environments. In *Proceedings of the 10th International Workshop on Middleware for Grids, Clouds and e-Science*, page 2. ACM, 2012.

[18] B. Balasubramanian and V. K. Garg. Fault tolerance in distributed systems using fused state machines. *Distrib. Comput.*, 2014.

[19] D. Behrens, D. Kuvaiskii, and C. Fetzer. Hardpaxos: Replication hardened against hardware errors. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, Oct 2014.

[20] http://www.sleepycat.com.

[21] A. Bessani, M. Santos, J. a. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, 2013.

[22] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, 2014.

[23] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, 2012.

[24] Y. Brun, G. Edwards, J. Y. Bang, and N. Medvidovic. Smart redundancy for distributed computation. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, 2011.

[25] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[26] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, Oct. 1999.

[27] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[28] http://www.clamav.net/.

[29] 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 *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Oct. 2012.

[30] Criu. http://criu.org, 2015.

[31] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[32] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nov. 2013.

[33] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[34] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, 2015.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, 2014.

[36] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[37] M. P. I. Forum. Open mpi: Open source high performance computing, Sept. 2009.

[38] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[39] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, Oct. 2011.

[40] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[41] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[42] D. Huynh Tu, B. Pietro, W. Han, L. Ki Shu, W. Hakim, C. Marco, P. Fernando, and S. Robert. Network hardware-accelerated consensus. Technical report, USI Technical Report Series in Informatics, 2016.

[43] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, 2016.

[44] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Pro-*

ceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012), CCGRID '12, 2012.

[45] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. Aug. 2014.

[46] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2016.

[47] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, 2010.

[48] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[49] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[50] S. Krishnan. *Programming Windows Azure: Programming the Microsoft Cloud*. May 2010.

[51] L. Lamport. Paxos made simple. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf.

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

[53] L. Lamport. Fast paxos. Fast Paxos, Aug. 2006.

[54] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Fast replication with nopaxos: Replacing consensus with network ordering. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2016.

[55] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[56] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, Mar. 2008.

[57] M. R. Lyu. *Software fault tolerance*. John Wiley & Sons, Inc., 1995.

[58] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[59] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, 2014.

[60] R. Martins, R. Gandhi, P. Narasimhan, S. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo. Experiences with fault-injection in a byzantine fault-tolerant protocol. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 41–61. Springer, 2013.

[61] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[62] H. Meling, K. Marzullo, and A. Mei. When you don't trust clients: Byzantine proposer fast paxos. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, 2012.

[63] https://memcached.org/.

[64] E. Michael. *Scaling Leader-Based Protocols for State Machine Replication*. PhD thesis, University of Texas at Austin, 2015.

[65] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2013.

[66] Mongodb. http://www.mongodb.org, 2012.

[67] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[68] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[69] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[70] http://www.openldap.org/.

[71] S. Peluso, A. Turcu, R. Palmieri, and B. Ravindran. On exploiting locality for generalized consensus. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 766–767. IEEE, 2015.

[72] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014.

[73] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[74] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[75] M. Primi. LibPaxos. http://libpaxos.sourceforge.net/.

[76] http://redis.io/.

[77] ssdb.io/.

[78] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. May 2014.

[79] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[80] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, SOSP '15, Oct. 2015.

[81] G. G. Wittawat Tantisiriroj. Network file system (nfs) in high performance networks. Technical Report CMU-PDLSVD08-02, Carnegie Mellon University, Jan. 2008.

[82] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, Apr. 2009.

[83] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.