APUS: Fast and Scalable State Machine Replication on RDMA

Submission #256

***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 3.3x. 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 a program on several computer replicas and enforces same inputs for this program as long as a quorum majority of replicas still behave normally, tolerating various faults such as minor replicas failures and packet losses. To achieve this powerful fault-tolerance, SMR typically uses a distributed consensus protocol called PAXOS [45], [40], [39], [61]. Recent SMR systems [37], [31], [23] have greatly improved the availability of server programs that serve online requests and store important data.

Unfortunately, existing PAXOS protocols incur prohibitive 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. A round-trip causes big latency as it goes through TCP/IP layers such as OS kernel. This latency may be acceptable for leader election [18], [6] or heavyweight transactions [23], [37], but undesirable for key-value servers (e.g., `Redis` and `Memcached`).

As the number of concurrent requests or replicas increases, PAXOS consensus latency often increases linearly [29] due to the linearly increasing number of consensus messages. This limited scalability prevents SMR from deploying with general server programs for two reasons. First, the performance of server porgrams are often scalable to number of CPU cores on serving concurent requests. Second, some advanced SMR systems (e.g., Azure [38]) often deploy seven or nine replicas to handle replica failures and updates.

To improve scalability, one approach is introducing parallel techniques such as multithreading [6], [17] or asynchronous IO [23], [57]. 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 PAXOS-like protocols [6], [17], [23], [57] on 24 cocurrent consensus requests on a computer with 24 CPU cores and 40Gbps network. We found that, when replica group size increased from 3 to 9, the consensus latency of three protocols increased by 30.3% to 156.8%, 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 [60], partitioning program states [29], [16], [47], splitting consensus leadership [44], [17], and hierarchical replication architecture [36], [29]. These systems have improved throughput. However, the core of these systems, PAXOS, still has an unscalable consensus latency. Previous work [29], [47], [36] explicitly desires to have a PAXOS protocol with a more scalable consensus latency.

Fortunately, Remote Direct Memory Access (RDMA) becomes a possible scalability 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 a remote replica's process without involving the remote OS kernel or CPU ("one-sided" operation). To ensure an RDMA write successfully resides in the remote memory, local process polls an RDMA ACK from remote NIC. Such a RDMA round-trip takes only about 3 μs [48].

However, due to the unrich RDMA operation types, fully exploiting RDMA speed in software systems is widely considered challenging by the community [48],

[35], [27], [55]. For instance, DARE [55] presents a two-round, RDMA-based PAXOS protocol in a sole-leader manner: leader does all RDMA workloads and backups do nothing. DARE was fast with 3 or 5 replicas. However, our evaluation shows that, as replica group grows, the leader met scalability bottlenecks (e.g., polling ACKs). DARE's consensus latency increased by 11.7x as the group grows by 35x (§VIII-B).

Our key observation is that we should carefully separate 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 [55], it has three major benefits. First, the leader has less workloads. Second, both leader and backups participate in consensus, which makes it possible to reach consensus with only one round [45]. Third, all replicas can get rid of the expensive RDMA ACK polling and just receive consensus messages on their bare, local memory. An analogy is threads receiving other threads' data via bare memory, a fast and scalable computation pattern.

This observation may raise reliability concerns because now the leader has no clue on whether remote writes succeed. Fortunately, PAXOS's protocol already tolerates various reliability issues, including message losses caused by hardware or program failures.

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, and our runtime system handles other technical challenges such as message atomicity (§IV-A), efficient input logging (§VI-A), and failure recovery (§VI-B).

Similar to general SMR systems [31], [23], APUS's design supports general, unmodified server programs. Within APUS, a program runs as is. APUS automatically deploys this program on replicas, intercepts inputs from a server program's inbound socket calls (e.g., `recv()`), and invokes its PAXOS protocol 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 protocol. To recover or add replicas, APUS leverages CRIU [22] to automatically, efficiently checkpoint a program without perturbing consensus in normal case.

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

We compared APUS with five popular, open source PAXOS-like implementations, including four traditional ones (libPaxos [57], ZooKeeper [6], CRANE [23] and S-Paxos [17]), and an RDMA-based one (DARE [55]). We evaluated APUS on 9 widely used or studied server programs, including 4 key-value stores (`Redis` [58], `Memcached` [46], `SSDB` [59], and `MongoDB` [49]), a SQL server `MySQL` [13], an anti-virus server `ClamAV` [20], a multimedia server `MediaTomb` [12], an LDAP server OpenLDAP [53], and a transactional database `Calvin` [60]. Evaluation shows that

1. APUS achieves one order of magnitude higher scalability and one order of magnitude faster latency than the literature (Figure 1). APUS's consensus latency outperforms 4 popular PAXOS protocols by 32.3x to 85.8x on 3 to 9 replicas. APUS is faster than DARE by up to 3.3x. When changing the replica group size from 3 to 105 (a 35x increase), APUS's consensus latency increases merely from 8.2 μs to 31.6 μs (a 3.8x, sub-linear increase).
2. APUS is general and easy to use. For all 9 evaluated programs, APUS ran them without any modification except `Calvin`.
3. APUS incurs low overhead on all 9 evaluated server programs. With 9 replicas, compared to servers' own unreplicated executions, APUS incurred only 4.2% overhead on throughput and 4.3% on response time in average.
4. APUS is robust. On leader failures, its leader election latency was reasonable and scalable.

Our major contribution is a new RDMA-based PAXOS protocol that achieves low consensus latency on over 100 replicas. APUS has the potential to largely improve both the scale and performance of many replication systems [29], [36], [23], [31], [16], [17]. For instance, Scatter [29] previously deploys 8$\sim$12 replicas in each PAXOS group, and now it can deploy hundreds of replicas in each group and achieves much better performance. Overall, a fast, scalable, and general service, APUS may significantly promote the deployments of PAXOS and improve both the consistency and fault-tolerance of various systems in datacenters.

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

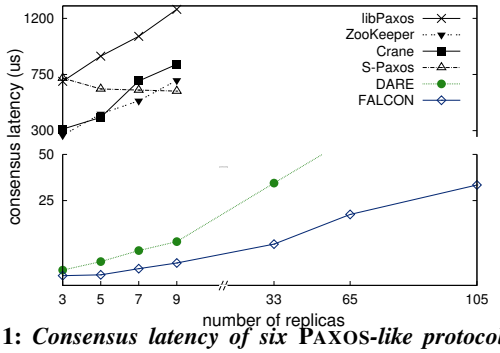

Fig. 1: *Consensus latency of six* PAXOS*-like protocols.* All protocols ran with 20 concurrent proposing clients. Both X and Y axes are broken to fit in all these protocols.

II. BACKGROUND

This section introduces PAXOS (§II-A) and RDMA (§II-B).

A. PAXOS

PAXOS [61], [40], [39], [19], [41], [45] runs the same program and its data on a group of replicas and enforces a strongly consistent sequence 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 due to hardware or program errors. If the leader replica fails, PAXOS elects a new leader from the backups.

To handle replica fail-overs, PAXOS replicas 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 traditional PAXOS protocols is notoriously high and unscalable. As datacenters incorporate increasingly faster networking hardware and more CPU cores, traditional PAXOS protocols [57], [17], [23], [31], [6] are having fewer performance bottlenecks on network bandwidth and CPU resources. However, these protocols still run on TCP/IP and have to go through various software layers such as network stack and OS kernel. Arrakis [54] reported that a `ping` program spent over 70% latency in these two layers.

To further quantify how software layers affect PAXOS consensus latency, we ran four PAXOS-like protocols on 40Gbps network with only one client sending requests. We found that, when changing the replica group size from three to nine, the consensus latency of three protocols increased by 30.3% to 156.8%, and 36.5% to 63.7% of this increase was in OS kernel. Therefore, existing systems (e.g., Scatter) deploy less than one dozen replicas in each PAXOS group.

B. RDMA

RDMA architectures (e.g., Infiniband [1] and RoCE [3]) become common in datacenters 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 [35], [48], [55]; it is several times faster than software-only kernel bypassing techniques (e.g., DPDK [2] and Arrakis [54]).

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 primitives. 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. One-sided operations is about 2x faster than two-sided operations because a two-sided operation contain two one-sided operations [48]. APUS uses one-sided operations, and the rest of this paper denotes such a operation as "WRITE".

RDMA communications between a local NIC and a remote NIC requires setting up a Queue Pair (QP), including a send queue and a receive queue. Each QP associates with 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 [35] reported that WRITEs on RC and UC OPs incur almost the same latency, so APUS uses RC QPs.

To ensure a remote replica is alive and a WRITE succeeds, a common RDMA practice is that after a WRITE is pushed to a QP, the local replica polls an ACK from the associated CQ before it continues (the so called *signaling*). Polling ACK is time consuming as it involves synchronization between the NICs on both sides of a CQ. We collected the time taken in polling ACKs in a recent RDMA-based PAXOS protocol DARE [55], and we found that, although DARE has been carefully optimized (its leader maintains one global CQ to receive backups' ACKs in batches), polling ACKs still became a scalability bottleneck: when the CQ was empty, it took 0.039∼0.12 μs; when the CQ has one or more ACKs, it took 0.051∼0.42 μs. As the number of ACKs is linear to replica group size, polling ACKs is a major scalability bottleneck (§VIII).

Fortunately, depending on protocol logic, one can do *selective signaling* [35]: 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 in its CQs.

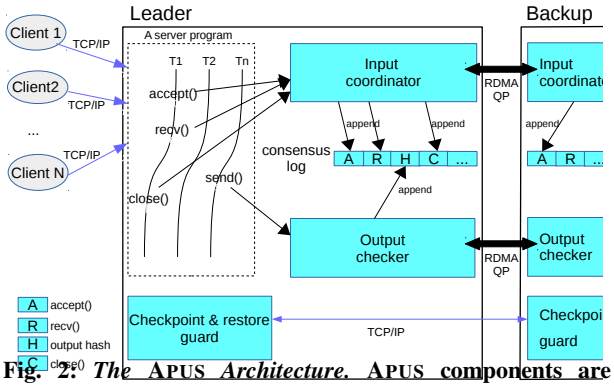

Fig. 2: ***The*** **APUS** ***Architecture.*** **APUS components are shaded (and in blue).**

III. APUS OVERVIEW

APUS follows a typical PAXOS runtime system deployment [29], [37], [31], [23], [55]. 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 leader handles client requests and runs our RDMA-based PAXOS protocol to coordinate inputs among replicas.

Figure 2 shows APUS's architecture. APUS intercepts a server program's 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*).

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. Because output hash generations are fast and an output consensus is invoked occasionally, our evaluation didn't observe performance impact on this 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

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

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 [45] 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

```
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.*

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 `write()`), 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, monotonically increasing viewstamp to locate this entry in the consensus log, building a log entry structure for this call, and writing this entry to a local parallel logging storage on SSD (§VI-A).

The third step (**L3**) is to WRITE a log entry to remote backups in parallel. Unlike a previous RDMA-based consensus algorithm [55] which has to wait for ACKs from remote NICs, our WRITE immediately returns after pushing the entry to its local QP between the leader and each backup, because PAXOS has handled the reliability issues (e.g., packet losses) for our WRITEs. In our evaluation, pushing a log entry to local QP took no more than 0.3 μs. Therefore, the WRITEs to all backups are done in parallel (see **L3** in Figure 3).

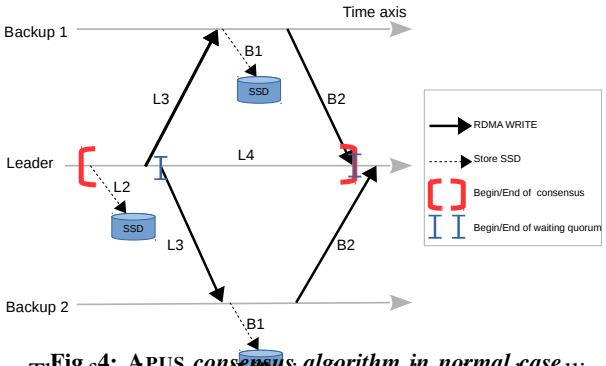

Fig. 4: APUS *consensus algorithm in normal case.*

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 [26], [35] 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 [62] 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 [45]. 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 does three steps to agree on consensus requests, shown in Figure 4. First (**B1**), it does a regular PAXOS view ID checking to see whether the leader is 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 forwarding the socket calls to the local server program. This forwarding faithfully builds and closes concurrent connections between the follower and the local server program according to the socket calls in the consensus log.

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 [45] 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

Compared to traditional PAXOS leader election protocols, RDMA-based leader election poses one main issue caused by RDMA. 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. An existing system [55] establishes an extra control QP with extra remote memory to handle leader election, posing more complexity via the extra communication channels.

APUS addresses this issue with a simple, clean approach. 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 have not received heartbeats from the leader for a period of 3*T, they start to elect a new leader and let their follower threads poll from the `elect` array.

Backups start a standard PAXOS leader election algorithm [45] 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 [39] 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.

D. Reliability Guarantee

To minimize protocol-level bugs, APUS's PAXOS protocol mostly sticks with a popular, practical implementation [45], especially the behaviors of senders and receivers (§IV-A and §IV-C). For instance, both APUS's normal case algorithm and the popular implementation [45] involve two messages and same senders and receivers (although we use WRITEs and carefully make them run in parallel). We made this choice because PAXOS is notoriously difficult to understand [51], [39], [40], [61] or implement [19], [45] verify [64], [30]. Aligning with a practical PAXOS implementation [45] helps us incorporate these readily mature understanding, engineering experience, and the theoretically verified safety rules into our protocol design and implementation.

Although APUS's PAXOS protocol works on a RDMA network, the reliability of this protocol does not rely on the lossless networking in RDMA. APUS's protocol still complies with the standard PAXOS failure-handling model, where a stable storage exists, but hardware may fail, network may be partitioned, packets may be delayed or lost, and server programs may crash.

V. OUTPUT CHECKING PROTOCOL

Most server programs are multithreaded and they may run into nondeterminism (e.g., concurrency errors [43]), which may cause replicas to diverge. APUS provides a fast output checking protocol for a practical purpose: improving APUS users' assurance on whether replicas run in sync. If diverged replicas are detected, users can restore them (§VI-B).

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, it did not incur observable performance lost in our evaluation.

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

This section first presents our parallel input logging mechanism (§VI-A) for storing inputs efficiently, and then our checkpoint/restore mechanism for recovering and adding replicas (§VI-B).

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 [14], as we found this method fastest in evaluation.

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 thread-safe, parallel logging approach [15]: 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 the preceding stores. This implementation efficiently kept our input logging latency within 2.8$\sim$8.7 μs (§VIII-C).

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.

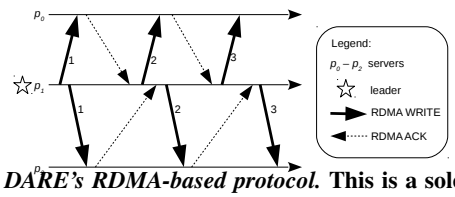

Fig. 5: *DARE's RDMA-based protocol.* This is a sole-leader, two-round protocol with three steps: (1) the leader WRITEs a consensus request to all backups' consensus logs and waits for ACKs to check if they succeed; (2) for the successful backups in (1), leader does WRITEs to update tail pointer of their consensus logs; and (3) on receiving a majority of ACKs in (2), a consensus is reached, leader does WRITEs to notify backups without needing to wait ACKs.

Specifically, APUS leverages CRIU, a popular, open source process checkpoint 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 to do the checkpoint.

VII. Discussions

This section compares APUS and DARE's protocols (§VII-A), and discusses APUS's limitations (§VII-B).

A. Comparing APUS and DARE

We highly appreciate DARE [55], the first RDMA-based, PAXOS-like protocol. It is most relevant to APUS. To tolerate single point of program failures, DARE is designed to run with a small (three to five) replica group. Under this design choice, DARE's protocol is sole-leader (leader does all the consensus work; backups do nothing): the leader only needs to do two-round WRITEs and check the ACKs of these WRITEs for a consensus. Both rounds are essential because DARE's leader needs the second round to indicate the latest (tail) undergoing consensus request on remote replicas to in case they become a new leader. Figure 5 shows DARE's protocol. Figure 4 shows APUS's protocol.

To further improve performance, DARE makes two technical choices. First, to avoid delays caused by polling ACKs, DARE uses a global RDMA CQ (Completion Queue) for all replicas, making it possible to collect multiple ACKs each poll operation. Second, this protocol does not incorporate a persistent storage or checkpoint/restore design. Therefore, DARE lacks durability (§II), an important guarantee in traditional PAXOS protocols and in APUS.

However, DARE is not designed to scale to a large replica group because its leader does all the consensus work. Our evaluation shows that both DARE's ACK pollings and its two-round consensus incurred scalability bottlenecks and had an approximately linear consensus

latency: DARE's consensus latency increased by 11.7x as replica group size increased by 35x (§VIII).

Overall, APUS differs from DARE in three aspects. First, APUS's protocol has only one RDMA round (Figure 4), while DARE has two rounds. To achieve one-round consensus, APUS's backups poll from its local memory to receive consensus requests, so APUS consumes more CPU than DARE on backups. Second, APUS has shown to scale well (sub-linearly) on 100+ nodes (§VIII-B); DARE [55] 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 up to 3.3x. §VIII-B compares APUS and DARE performance in detail.

B. APUS *Limitations*

APUS currently does not hook random functions such as `gettimeofday()` and `rand()` because these random results are often explicit and easy to examine from network outputs (e.g., a timestamp in the header of a reply). Existing approaches [37], [45] in PAXOS protocols can also be leveraged to intercept these functions and make general programs produce same results among replicas.

Like recent systems [55], [23], APUS totally orders all types of requests and it has not incorporated read-only optimization [37], because its performance overhead compared to the evaluated programs' unreplicated executions is already reasonable (§VIII-C). However, APUS can be extended to support read-only optimization if two conditions are met: (1) whether the semantic an operation is read-only is clear in a server program; and (2) the number of output bytes for this operation is fixed. GET requests in key-value stores often meet these two conditions.

We use GET requests to present a design. APUS intercepts a client program's outbound socket calls (e.g., `send()`), compares the first three bytes in each call with "GET". If they match, APUS appends two extra APUS metadata fields `read_only` and `length` in this outbound call to the server. APUS then intercepts a server's `recv()` calls and strips these two fields. If the first field is true, APUS directly processes this operation in a local replica and strips the next `length` bytes from the output checker within the same connection. In sum, APUS processes these operations locally without making outputs across replicas diverge.

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 via Infiniband [1]. The `ping` latency between every two replicas is 84 μs (the IPoIB round-trip latency).

Our evaluation machines also include one Dell R320 server for client programs. It has Linux 3.16.0, 2.2GHz Intel Xeon 12 hyper-threading cores, 32GB memory, and 160GB SSD. To mitigate latency of client requests, this client machine is located at the same LAN as the RDMA replicas with a 1Gbps NIC. The average `ping` latency between this machine and a RDMA replica is 250 μs. Note that which machines to hold clients do not affect any PAXOS protocol's consensus latency (it is only affected by the RDMA network among replicas).

We compared APUS with five popular, open source PAXOS-like implementations, including four traditional ones (libPaxos [57], ZooKeeper [6], CRANE [23] and S-Paxos [17]), and an RDMA-based one (DARE [55]). S-Paxos is designed to achieve scalable throughput when more replicas are added.

We evaluated APUS on 9 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). These servers all update or store important data and files, thus the strong PAXOS fault-tolerance is especially attractive to these programs.

Table I introduces the benchmarks and workloads we used. To evaluate APUS's practicality, we used the protocol or program developers' own performance benchmarks or popular third-party benchmarks. For benchmark workload settings, we used the benchmarks' default workloads whenever available. To perform a stress testing on APUS's input consensus protocol, we chose workloads with significant portions of writes, because write operations often contain more input bytes than reads (e.g., a key-value SET operation contains more bytes than a GET).

The rest of this section focuses on these questions:

§VIII-A: What is APUS's consensus latency compared to traditional PAXOS protocols?

§VIII-B: What is APUS's consensus latency compared

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 performance benchmark program. Workloads are all concurrent.**

to DARE?

§VIII-C: What is the performance overhead of running APUS with general 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 PAXOS

As a common evaluation practice in PAXOS systems [55], [42], when comparing APUS with other PAXOS protocols, we ran APUS with a popular key value store `Redis`. For all six PAXOS protocols, we spawned 20 consensus requests, a common high concurrent value in prior evaluation [6], [23], [31]. Our evaluation also showed that most server programs reached peak performance at this concurrent value (§VIII-C).

We compared APUS with four traditional protocols, libPaxos [57], ZooKeeper [6], CRANE [23] and S-Paxos [17]. All these four protocols were run on IPoIB (§II-B). Figure 1 shows the results. Three traditional protocols incurred almost a linear increase of consensus latency 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. Afterall, its latency was always over 600 μs. APUS's consensus latency outperforms these four protocols by 32.3x to 85.8x on 3 to 9 replicas. Therefore, we needn't run these protocols with more replicas.

To understand why traditional protocols are unscalable, we ran only one client with them and inspect the micro events in their protocols, shown in Table II. Three protocols had scalable latency on the arrival of their first consensus reply (the "First" column), which means that network bandwidth is not a scalability bottleneck for them. libPaxos is an exception because its two-round protocol consumed much bandwidth. However, 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 the various systems layers, including OS kernels, network libraries, and language runtimes (e.g., JVM), are the major scalable bottleneck (the "Sys" column). This indicates that RDMA is useful to bypass the systems layers for better scalability.

Note that when running with three replicas, libPaxos and CRANE's proposing leader and acceptors are in different threads, so they two had different "First" and "Major" arrival times. CRANE and S-Paxos's proposing leader itself is just an acceptor, so they two had same "First" and "Major" arrival times (i.e., their "Sys" times were 0).

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: *Scalability bottleneck analysis in traditional* PAXOS *protocols.* The "Proto-#Rep" column means the PAXOS protocol name and replica group size; "Latency" means the consensus latency; "First" means the latency of its first consensus reply; "Major" means the latency of its the majority reply; "Process" means time spent in processing all replies; and "Sys" means time spent in systems (OS kernel, network libraries, and JVM) between the "First" and the "Major" reply. All times are in μs.

B. Comparing with DARE

We compared APUS and DARE [55] on key-value stores with the same key and data length. APUS ran a widely used one, `Redis`; DARE ran a 335-line, RDMA-based key-value store written by their authors. To thoroughly analyze their latency with scalability, we run more replicas than the nine physical machines (i.e., each machine runs several replicas). Since our RDMA NIC bandwidth is 40Gpbs, we didn't find network an obvious bottleneck while running several replicas on one machine: when changing replica group size from 9 to 105, an RDMA round-trip increased from 2.6$\sim$4.4 μs.

Table III shows the consensus latency of APUS and DARE on the same update-heavy workload, which contains half GETs and half SETs. This workload represents real-world applications such as an advertisement log that records recent user activities [55]. APUS and DARE achieved similar latency at three replicas, and APUS was much faster than DARE on more replicas. Their difference was 2.5x$\sim$3.3x on over 33 replicas. When changing replica group size from 3 to 105 (a 35x increase), APUS's consensus latency merely increased by 3.8x, and DARE increased by 11.7x.

APUS scales better than DARE due to two main reasons. First, in a protocol level, APUS's protocol carefully separate the RDMA workloads across leader and backups, and it is a one-round protocol (§IV-A). DARE lets its leader do all the consensus work and backups do nothing, and it is a two-round protocol (§VII-A). Therefore, DARE involves approximately 2x more RDMA communications than APUS.

Second, in an RDMA communication primitive level, APUS lets all replicas receive consensus messages on their bare, local memory. DARE frequently polls RDMA ACKs from a RDMA Completion Queue (CQ) in each consensus round. An ACK polling or insertion operation on the CQ involves synchronization between RDMA NICs among replicas. We found that DARE's ACK

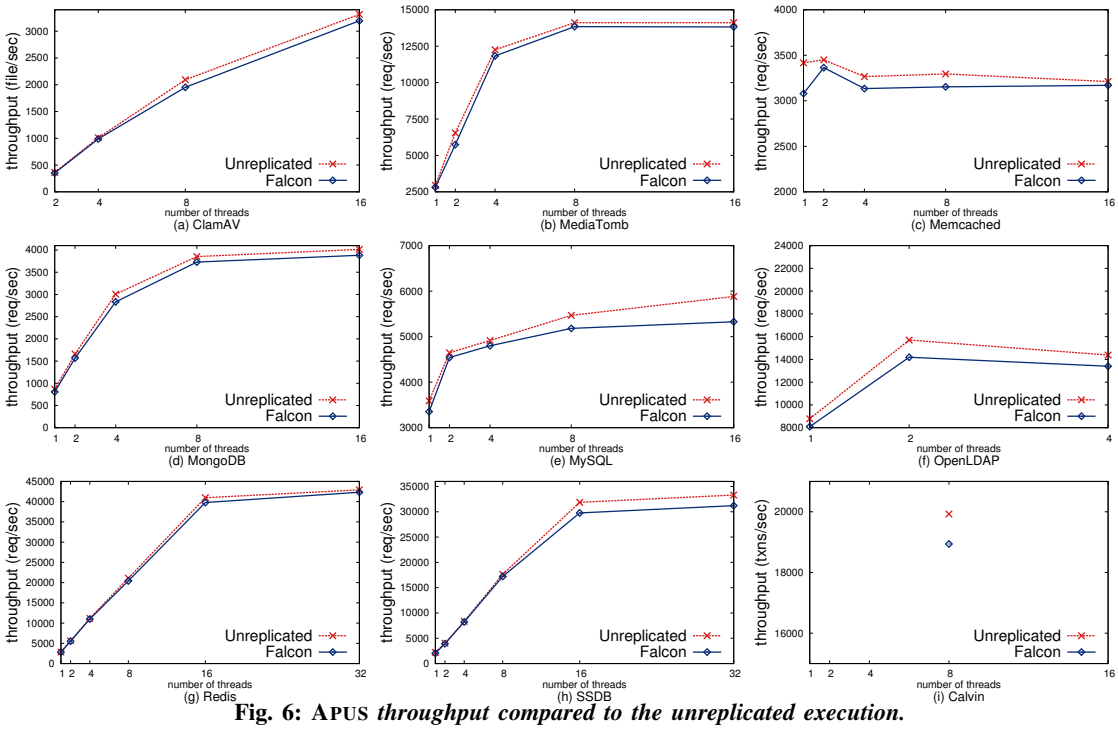

Fig. 6: APUS *throughput compared to the unreplicated execution.*

mechanism is already highly optimized with a global CQ (it can poll more ACKs at one time). However, we found that ACKs arrived randomly, and each polling in DARE took up to 0.042 μs (§II). The number of ACK pollings in DARE is linear to replica group size (§VII-A), causing a linearly increased consensus latency.

APUS does the same consensus round on all types of requests. DARE has two special techniques on GET requests. First, it batches GET requests for consensus, which may improve throughput but aggravate latency. Second, DARE's GET requests only involve one-round consensus (it does RDMA reads to fetch all replicas' PAXOS view IDs [45] and check if a majority of them match the leader's). We also ran DARE with a read-heavy workload (Table III) and its consensus latency was about 50% slower than APUS on over 33 replica groups. Note that APUS is a durable protocol, and DARE is a volatile protocol. Overall, we considered APUS faster and more scalable than DARE.

# Replicas	3	9	33	65	105
APUS (update-heavy)	8.2	8.8	13.0	20.3	31.6
DARE (update-heavy)	8.8	12.0	32.5	64.0	102.5
DARE (read-heavy)	5.8	7.2	16.8	29.2	45.1

TABLE III: *Consensus latency of* APUS *and DARE.* The update-heavy workload consist of 50% SETs. The read-heavy workload consist of 10% SETs.

C. Performance Overhead

To stress APUS, we used a large replica group size of 9 to run all server programs. We spawned up to 32 concurrent connections, and then we measured both response time and throughput. We also measured APUS's bare consensus latency. Each performance data point was taken from a mean value of 10 executions.

APUS ran with 9 evaluated programs without modifying them except `Calvin`. `Calvin` integrates its client program and server program within the same process and uses local memory to let these two programs talk. To make `Calvin`'s client and server communicate with POSIX sockets so that APUS can intercept the server's inputs, we wrote a 23-line patch for `Calvin`.

We turned on and off the output checking (§V) and didn't observe difference in APUS performance. Only three programs (`MediaTomb`, `MySQL`, and OpenLDAP) have different output hashes cause by physical times.

Figure 6 shows APUS's throughput and Figure 7 response time. We varied the number of concurrent client connections for each server program by from one to 32 threads. For `Calvin`, we only collected the 8-thread result because `Calvin` uses this constant thread count in their code to serve client requests.

Overall, compared to these server programs' unreplicated executions, APUS merely incurred a mean throughput overhead of 4.2% (note that in Figure 6, the Y-axises of most programs start from a large number). APUS's mean overhead on response time was 4.3%.

As the number of threads increases, all programs' unreplicated executions got a performance improvement except `Memcached`. A prior evaluation [31] also observed a similar `Memcached` low scalability. APUS scaled almost as well as the unreplicated executions.

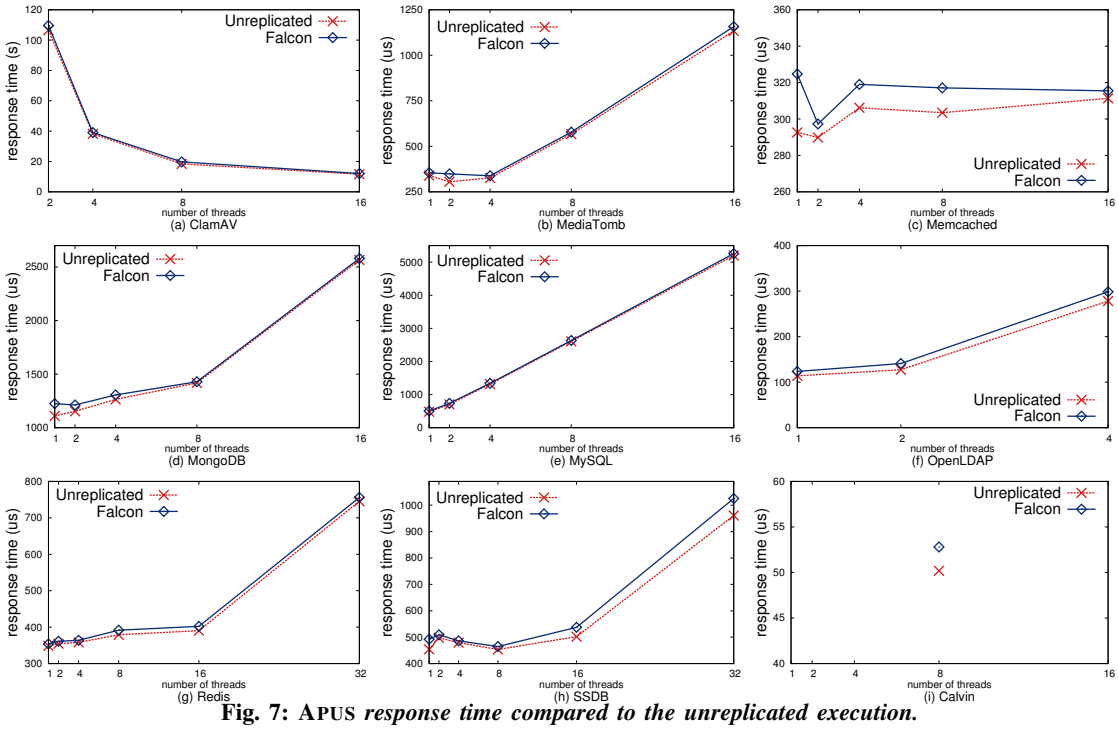

Fig. 7: APUS *response time compared to the unreplicated execution.*

APUS achieves such a low overhead in both throughput and response time mainly because of two reasons. First, for each `recv()` call in a server, APUS's input coordination protocol only contains two one-sided RDMA writes and two efficient local SSD writes (§VI-A) for the leader and each backup.

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	6.3 μs
`SSDB`	10,016	47.0	3.0 μs	6.2 μs
`Calvin`	10,002	128.0	8.7 μs	10.8 μs

TABLE IV: *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 for these calls.

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

By adding the last two columns in Table IV, a APUS input consensus took only 9.1 μs (`Redis`) to 22.4 μs (`MediaTomb`). APUS's small consensus latency makes it achieve reasonable throughputs in Figure 6 and response times Figure 7.

D. Checkpoint and Recovery

We ran the same performance benchmark as in §VIII-C and measured programs' checkpoint timecost. Each APUS periodic checkpoint operation (§VI-B) cost 0.12s to 11.6s on the evaluated server programs, depending on the amount of modified memory and files in the server programs since their own last checkpoint. `ClamAV` incurred the largest checkpoint time (11.6s) because it loaded and scanned files in a `/lib` directory.

Checkpoint operations did not affect APUS's performance in normal case because they were done on only one backup. The leader and other backups still formed majority and reached consensus rapidly.

To evaluate APUS's PAXOS recovery feature, we ran APUS with `Redis` and manually killed one backup, and we did not observe a performance change in the benchmark runs. We then manually killed the APUS leader and measured the latency of our RDMA-based leader election with three rounds (§IV-C). Figure V shows APUS's election latency from three to eleven replicas. Because PAXOS leader election is rarely invoked in practice, although APUS's election latency was slightly higher than its normal case consensus latency, we considered it reasonable.

# Replicas	3	5	7	9	11
Election latency (μs)	10.7	12.0	12.8	13.5	14.0

TABLE V: APUS*'s scalability on leader election.*

IX. RELATED WORK

Software-based consensus. There are a rich set of PAXOS algorithms [45], [40], [39], [61], [50] and implementations [19], [45], [18], [23]. PAXOS is notoriously difficult to be fast and scalable [47], [36], [29]. Since consensus protocols play a core role in datacenters [65], [32], [5] and worldwide distributed systems [21], [44], a variety of study have been conducted to improve specific aspects of consensus protocols, including order commutativity [50], understandability [51], [40], and verifiable reliability rules [64], [30].

To make PAXOS's throughput scalable (i.e., more replicas, higher throughput), various systems leverage PAXOS as a core building block to develop advanced replication approaches, including partitioning program states [29], [16], splitting consensus leadership [44], [17], and hierarchical replication [36], [29]. Theses approaches have shown to largely improve throughput. However, the core of these systems, PAXOS, still faces an unscalable consensus latency [47], [29], [36]. By using APUS as a building block, these system may scale even better.

Three state machine replication (SMR) systems, Eve [37], Rex [31], and CRANE [23], build traditional PAXOS protocols to improve the availability of general server programs. Evaluation in these systems shows that SMR services incur modest overhead on server programs' throughput compared to their unreplicated executions. APUS's consensus latency is much faster than their traditional protocols (e.g., see CRANE in Figure 1).

Hardware- or Network- assisted consensus. Recent systems [33], [56], [42], [25] leverage augmented network hardware or topology to improve PAXOS consensus latency. "Consensus in a Box" [33] implemented ZooKeeper's consensus protocol in FPGA and intensively modify its hardware TCP/IP stack for fast latency. It implemented a key-value store in hardware and showed similar consensus latency to APUS on 3 or 5 replicas. It did not discuss or evaluate its scalability. Compared to this protocol, APUS does not need to modify hardware, so it is easier to deploy and support general programs.

Speculative Paxos [56] and NOPaxos [42] leverage the synchrony feature of datacenter topology to order requests, so they can eliminate consensus rounds if packets are not reordered or lost. Moreover, NOPaxos assigns a global sequence ID to packets so that it can proactively detect reordered or lost packets. If packets are lost or reordered, they invoke consensus to rescue. These two systems are not designed for scalability because when replica group and the datacenter network become big, the probability that replicas incur reordered or lost packets will increase. Moreover, these two systems' consensus modules go through TCP/IP layers and incur high round-trip latency. APUS can help them.

RDMA-based systems. RDMA techniques have been implemented in various architectures, including Infiniband [1], RoCE [3], and iWRAP [4]. RDMA have been leveraged in many systems to improve application-specific latency and throughput, including high performance computing [28], key-value stores [48], [35], [26], [34], transactional processing systems [62], [27], and file systems [63]. For instance, FaRM [26] leverages RDMA to build a fast DHT. FaRM uses a data replication approach [52], which works in a primary-backup manner [24]. In general, PAXOS provides better consistency and availability than primary-backup. These RDMA-based systems use RDMA to improve performance in different 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 5 PAXOS protocols and 9 widely used programs shows that APUS is fast, scalable, and robust. It has the potential to improve the consistency and fault-tolerance of many systems in datacenters. APUS is deployable: all source code, benchmarks, and raw evaluation results are available at http://github.com/icdcs17-p256/apus.

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] http://www.sleepycat.com.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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 *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

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

[36] 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.

[37] 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.

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

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

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

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

[42] 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.

[43] 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.

[44] 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.

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

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

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

[48] 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.

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

[50] 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.

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

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

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

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

[59] ssdb.io/.

[60] 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.

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

[62] 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.

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

[64] 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.

[65] 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.