APUS: Fast and Scalable PAXOS on RDMA

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

Department of Computer Science, University of Hong Kong

Abstract

State machine replication (SMR) uses PAXOS to enforce the same inputs for a program (e.g., Redis) replicated on a number of hosts, tolerating various types of failures. Unfortunately, traditional PAXOS protocols incur prohibitive performance overhead on server programs due to their high consensus latency on TCP/IP. Worse, the consensus latency of extant PAXOS protocols increases drastically when more concurrent client connections or hosts are added. This paper presents APUS, the first RDMA-based PAXOS protocol that aims to be fast and scalable to client connections and hosts. APUS intercepts inbound socket calls of an unmodified server program, assigns a total order for all input requests, and uses fast RDMA primitives to replicate these requests concurrently.

We evaluated APUS on nine widely-used server programs (e.g., Redis and MySQL). APUS incurred a mean overhead of 4.3% in response time and 4.2% in throughput. We integrated APUS with an SMR system Calvin. Our Calvin-APUS integration was 8.2X faster than the extant Calvin-ZooKeeper integration. The consensus latency of APUS outperformed an RDMA-based consensus protocol by 4.9X. APUS source code and raw results are released on github.com/hku-systems/apus.

1 Introduction

State machine replication (SMR) runs the same program on replicas of hosts and invokes a distributed consensus protocol (typically, PAXOS [56]) to enforce the same total order of inputs among replicas. Since the consensus on an input can be reached as long as a quorum (typically, majority) of replicas agree, SMR tolerates various errors, including hardware failures of minor replicas. SMR is deployed on clouds to make the metadata (e.g., leadership) of a distributed system highly available.

The strong fault-tolerance of SMR makes it an ideal high-availability service for general server programs. Recent SMR systems [51, 42, 32] use PAXOS to enforce the same inputs for a server program, and they use advanced techniques (e.g., deterministic inter-thread synchronization [32, 81]) to make the program transit the same execution states across replicas. These SMR systems tolerate hardware failures for server programs.

Unfortunately, despite much effort, state-of-the-art still lacks a fast, scalable PAXOS protocol for general server programs. A main reason is that traditional PAXOS protocols [78, 69, 32] go through software network layers in OS kernels [75], which incurs high consensus latency. For efficiency, PAXOS protocols typically take the Multi-Paxos approach [55]: it 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 round-trip time (RTT) is required between the leader and a backup. Given that a ping RTT in LAN typically takes hundreds of μs, and that the request processing time of key-value store servers (e.g., Redis) is at most hundreds of μs, PAXOS incurs high overhead in the response time of server programs.

Worse, the consensus latency of extant consensus protocols is often *scale-limited*: it increases drastically when the number of concurrent requests or replicas increases. For instance, the consensus latency of ZooKeeper [44] increases by 2.6X when the number of concurrent proposing requests increases from 1 to 20 (on 3 replicas). Scatter [40] shows that the consensus latency of its PAXOS protocol increases by 1.6X when the number of replicas increases from 3 to 9.

Our evaluation found that the scalability problem in traditional consensus protocols mainly stem from OS kernels. We ran 4 popular consensus protocols [6, 24, 32, 78] on 24-core hosts with 40Gbps network (i.e., network bandwidth was not a bottleneck), we then ran 24 concurrent request connections. When the number of replicas increased from 3 to 9, the consensus latency of 3 protocols increased by 105.4% to 168.3%, and 36.5% to 63.7% of the increase was in OS kernels.

As modern server programs tend to support more concurrent client connections, and advanced SMR systems tend to deploy more replicas (e.g., Azure [54] deploys seven or nine replicas) to support both replica failures and upgrades, the limited scalability in extant consensus protocols becomes even more pronounced.

To reduce consensus latency, NOPaxos [58] uses a dedicated network switch in a datacenter to totally order packets, and it safely skips consensus if packets arrived at the switch and replicas are in the same order. NOPaxos is not designed for server programs with many concurrent connections and replicas, because prior work [36, 57, 14] shows that packet reordering rate increases when more network flows and hosts are added.

Recent hardware-accelerated consensus protocols [46, 35, 45, 77] are also effective on reducing consensus latency, but they are either unsuitable for general server programs or are not designed to be scalable on concurrent client connections. For instance, DARE [76], a novel consensus protocol, achieves the lowest consensus latency on a small number of client connections, but both its evaluation and ours show that its consensus latency increases quickly when more connections are added.

We argue that the problem of high, scale-limited consensus latency is not fundamental in PAXOS. OS kernels, a major source of this problem, can be bypassed with advanced network features such as Remote Direct Memory Access (RDMA) within the same datacenter. Moreover, FaSST [49] shows that RDMA can obtain scalable latency on many concurrent transactions.

We present APUS,[1] the first RDMA-based PAXOS protocol and runtime system. APUS intercepts an unmodified server program's inbound socket calls (e.g., `recv()`), assigns a total order for all received requests in all connections, and uses fast RDMA primitives to invoke consensus on these requests concurrently. To ensure the same robustness as regular PAXOS, APUS's runtime system efficiently tackles several reliability challenges, including atomic delivery of messages (§4.2), persistent input logging (§5.1), and failure recovery (§5.2).

A fast and scalable PAXOS protocol, APUS has many practical applications, and we elaborate two below. First, it can be integrated into existing SMR systems (e.g., Calvin [81]), making the response time of a server program running in these systems almost as fast as the program's unreplicated execution.

Second, it can support many server programs that are already well-tested or deterministic, including single-threaded ones such as `Redis` [79] and multi-processed ones such as `Nginx` [70] and `MediaTomb` [12]. Even if a program is pre-mature and undergoing debugging, enforcing the same order of inputs by APUS can still help debugging tools (e.g., PRES [74]) easily reproduce bugs. §3.2 further illustrates APUS's broad applications.

We implemented APUS in Linux and compared it with five open source consensus protocols, including four traditional ones (libPaxos [78], ZooKeeper [6], CRANE [32] and S-Paxos [24]), and an RDMA-based one (DARE [76]). We evaluated APUS on nine widely used or studied programs, including 4 key-value stores (`Redis` [79], `Memcached` [65], `SSDB` [80], and `MongoDB` [68]), a SQL server `MySQL` [13], an anti-virus server `ClamAV` [29], a multimedia server `MediaTomb` [12], an LDAP server OpenLDAP [73], and an SMR database `Calvin` [81]. Evaluation shows that

1. APUS is fast and scalable. Figure 1 shows that APUS's consensus latency outperformed four traditional consensus protocols by at least 32.3X. Its consensus latency stayed almost constant to the number of concurrent requests and replicas. Its consensus latency was faster than DARE by 4.9X in average.
2. APUS is easy to work with SMR. The `Calvin`-APUS integration took only 39 lines of code. `Calvin`-APUS's response time was 8.2X faster than the extant `Calvin-ZooKeeper` integration, and it incurred only 10.6% overhead in response time and 4.1% in throughput over `Calvin`'s unreplicated execution.
3. APUS achieves low overhead on real-world server programs. Compared to all nine server programs' unreplicated executions, APUS incurred 4.3% overhead in response time and 4.2% in throughput.
4. It is robust on replicas failures and packet losses.

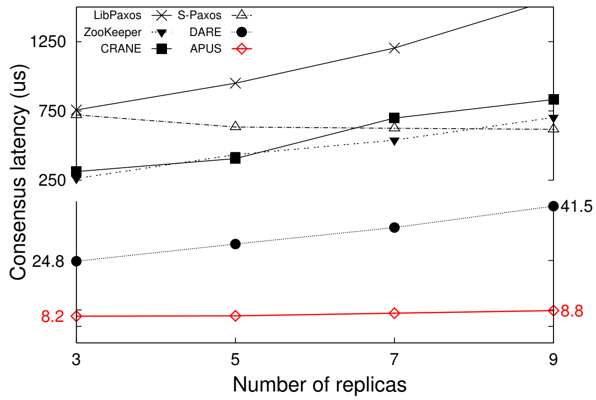

Figure 1: ***Comparing* APUS *to five existing consensus protocols.* All six protocols ran a client with 24 concurrent connections. The Y axis is broken to fit in all protocols.**

Our major contribution is the first PAXOS protocol that achieves low performance overhead on diverse, widely-used server programs. A fast, scalable, and deployable PAXOS protocol, APUS can widely promote the adoption of SMR and improve the fault-tolerance of various systems [54, 50, 32, 42, 23, 24] within a datacenter.

The remaining of this paper is organized as follows. §2 introduces PAXOS and RDMA background. §3 gives an overview of APUS. §4 presents APUS's consensus protocol with its runtime system. §5 presents implementation details. §6 compares APUS with DARE. §7 does evaluation, §8 discusses related work, and §9 concludes.

2 Background

2.1 PAXOS

PAXOS [56, 55] enforces a total order of inputs for a program running across replicas. Because a consensus can be reached as long as a majority of replicas agree, PAXOS is known for tolerating various faults, including hardware failures of minor replicas and packet losses.

SMR systems [42, 32] often use PAXOS to replicate important online services. An typical SMR system con-

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

tains two orthogonal parts: (1) a PAXOS protocol that enforces a total order of inputs for the same program running across replicas; and (2) a technique (e.g., deterministic mutex locks [32, 81]) that makes the program transit same execution states on the same inputs.

The consensus latency of PAXOS protocols is notoriously high and unscalable [6, 40]. As datacenters incorporate faster networking hardware and more CPU cores, traditional consensus protocols [78, 24, 32, 42, 6] are having fewer performance bottlenecks on network bandwidth and CPU resources.

However, software TCP/IP layers in OS kernels remain performance bottlenecks [75]. To quantify this bottleneck, we evaluated four traditional consensus protocols [6, 24, 32, 78] on 24-core hosts with 40Gbps network, and we spawned 24 concurrent consensus connections. When changing the replica group size from 3 to 9, although network and CPUs were not saturated, the consensus latency of 3 protocols drastically increased by 105.4% to 168.3% (Figure 1), and 36.5% to 63.7% of this increase was in OS kernel. When only one consensus connection was spawned, the latency increase on the number of replicas was more gentle (Table 2 in §7.1).

This evaluation shows that both the number of concurrent requests and replicas make consensus latency increase drastically. This problem becomes worse as server programs tend to support more concurrent requests and advanced SMR systems (e.g., Azure [54]) deploy seven to nine replicas to in case replica failures and upgrades.

2.2 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 bypassing the OS kernel, but also its dedicated network stack implemented in hardware. Therefore, RDMA is considered the fastest kernel bypassing technique [48, 67, 76]; it is several times faster than software-only kernel bypassing techniques (e.g., DPDK [2] and Arrakis [75]).

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 runs unmodified socket programs, but it is a few times slower than the other two types. A one-sided RDMA write can directly write from one replica's memory to a remote replica's memory without involving the remote OS kernel or CPU. Prior work [67] shows that one-sided operations are up to 2X faster than two-sided operations [49], so APUS uses one-sided operations (or "WRITE" in this paper). On a WRITE success, the remote NIC (network interface card) sends an RDMA ACK to local NIC.

A one-sided RDMA communication between a local and a remote NIC has a Queue Pair (QP), including a send queue and a receive queue. Such a QP is a global data structure between every two replicas, but pushing a message into a local QP takes at most 0.2 μs in our evaluation. Different QPs between different replicas work in parallel (leveraged by APUS in §4.1). 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 [48] shows that WRITEs on RC and UC OPs incur almost the same latency, so APUS uses RC QPs.

Normally, to ensure a WRITE resides in remote memory, the local replica 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 recent RDMA-based consensus protocol DARE [76]. We found that, although it is highly optimized (its leader maintains one global CQ to receive all backups' ACKs in batches), busily polling ACKs slowed DARE down (§7.3): when the CQ was empty, each poll took 0.039$\sim$0.12 μs; when the CQ has one or more ACKs, each poll took 0.051$\sim$0.42 μs.

Fortunately, depending on protocol logic, one can do *selective signaling* [48]: 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.

3 Overview

3.1 APUS Architecture

APUS deployment is similar to a typical SMR's: it runs a program on replicas within 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 enforce the same total order for all requests 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*), a circular in-memory consensus log (the *log*), a guard process that handles checkpointing and recovering a server's process and file system state (the *guard*), and an optional output checking tool (the *checker*).

The coordinator is involved when a thread of a program running on the APUS leader calls an inbound socket call (e.g., `recv()`). The thread executes the Libc call, gets the received data, appends a log entry on the leader's local consensus log, and replicates this entry to backups' consensus logs using our PAXOS protocol (§4).

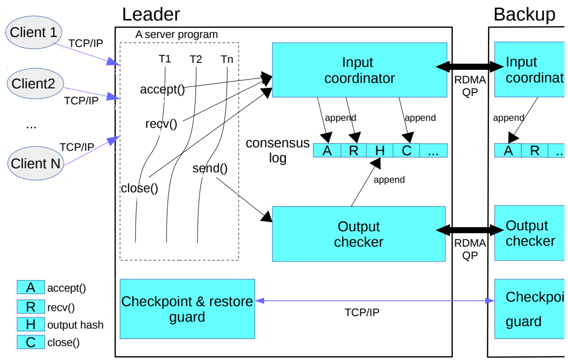

Figure 2: APUS *Architecture (key components are in blue).*

In this protocol, all threads in the server program running on the leader replica can concurrently invoke consensus on their log entries (requests), but APUS enforces a total order for all entries in the leader's local consensus log. As a consensus request, each thread does an RDMA WRITE to replicate its log entry to the corresponding log entry position on all APUS backups. Each APUS backup polls from the latest unagreed entry on its local consensus log; if it agrees with the proposed log entry, it does an RDMA WRITE to write a consensus reply on the leader's corresponding entry.

To ensure PAXOS safety [63], all APUS backups agree on the entries proposed from the leader in a total order without allowing any entry gap. When a majority of replicas (including the leader) has written a consensus reply on the leader's local entry, this entry has reached a consensus. By doing so, APUS consistently enforces the same consensus log for both the leader and backups. §4.5 presents a proof sketch on the correctness of the protocol, and §4.6 analyzes why it is fast and scalable.

The output checker is periodically invoked as a program replicated in APUS 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. For simplicity, the output checker uses APUS's input consensus protocol (§4) to compare hashes across replicas.

Our evaluation found that the output checker had negligible performance impact and all output divergence were due to physical times (§7.4). This suggests that many server programs are well-tested, and the output checker can be turned on only in program debug phase. If APUS is integrated into an SMR system, the output checker is not needed because SMR already has techniques to enforce the same program executions.

A guard runs on each APUS replica to cope with replica management, including checkpointing program states and adding/recovering replicas (§5.2).

3.2 Motivating Applications of APUS

Building fast SMR systems. Extant SMR systems (e.g., CRANE [32], Rex [42], and `Calvin` [81]) use TCP/IP-based consensus protocols, thus they incur high overhead in server programs' response time. APUS can greatly alleviate this overhead. Evaluation (§7.2) shows that the response time of our `Calvin`-APUS integration on realistic SQL workloads was 8.2X than its extant `Calvin`-ZooKeeper integration. Compared to `Calvin`'s unreplicated execution, APUS incurred only 10.6% overhead in response time and 4.1% in throughput.

Improving the availability of server programs. Many real-world server programs handle online requests and store important data, so they naturally demand high availability against hardware failures. Many programs are suitable to run with APUS because they are already well-tested or deterministic (e.g., single-threaded ones such as `Redis` and multi-processed ones such as `Nginx` and `MediaTomb`). Other orthogonal techniques such as deterministic multithreading [59, 18, 81, 33] can be combined with APUS to make a replicated server program behave the same on the same inputs. Our evaluation (§7.4) shows that, compared to all nine evaluated programs' unreplicated executions at peak performance, APUS incurs 4.2% overhead in throughput and 4.3% in response time.

Improving debugging efficiency. Even if a server program is under development and may contain nondeterministic concurrency bugs, APUS can still benefit extant debug tools [74, 15, 52] because these tools often require extra mechanisms to frequently replay the same total order of inputs. APUS logs program inputs persistently, and it can efficiently replay these inputs in the same order when integrated into debug tools (e.g., PRES [74]).

4 The RDMA-based PAXOS Protocol

4.1 Normal Case

APUS's consensus protocol has three main elements. First, a PAXOS consensus log. Second, threads of a server program running on the leader host (or *leader threads*). APUS hooks the inbound socket calls (e.g., `recv()`) of these leader threads and invoke consensus requests on these calls. We denote the data received from each of these calls as a consensus request (i.e., an entry in the consensus log). Third, an APUS internal thread running on every backup (or *backup threads*), which agrees on consensus requests. The APUS leader enables the first and second elements, and backups enable the first and third elements.

Figure 3 depicts the format of a log entry in APUS's consensus log. Most fields are the same as those in a typical PAXOS protocol [63] except three: the `reply` array,

```
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;
```

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

`conn_vs`, and `call_type`. The `reply` array is a piece of memory on the leader side, preserved for backups to do RDMA WRITEs for their consensus replies. The `conn_vs` is for identifying which TCP connection this socket call belongs to (see §4.3). The `call_type` identifies different types of socket calls (e.g., the `accept()` type and the `recv()` type) for the entry.

Figure 4 shows APUS's consensus protocol. Suppose a leader thread invokes a consensus request when it calls a socket call `recv()`. This thread's consensus request has four steps. The first step (**L1**, not shown in Figure 4) is executing the actual socket call, because the thread needs to get the received data and returned value, to allocate a distinct log entry, and to replicate the entry in backups' consensus logs.

The second step (**L2**) is local preparation, including assigning a viewstamp (a totally-ordered PAXOS consensus request ID [63]) for this entry in the consensus log, allocating a distinct entry in the log, and storing the entry to a local storage (§5.1). We denote the time taken on storing an entry as t_{SSD}.

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 distinct entry and remote backups' corresponding entries. An **L3** WRITE returns quickly after pushing the entry to its local QP connecting the leader and each backup. We denote the time taken for this push as t_{PUSH}, which took at most 0.2 μs in our evaluation. t_{PUSH} is serial for concurrently arriving requests on each QP, but the WRITEs (all **L3** arrows in Figure 3) to different QPs run in parallel.

The fourth step (**L4**) is that the leader thread polls on its `reply` field in its local log entry to wait for backups' consensus replies. It breaks the poll if a number of heartbeats fail (§4.4). If a majority of replicas agrees on the entry, an input consensus is reached, the leader thread leaves this `recv()` call and proceeds with its program logic.

On each backup, a backup thread polls from the latest unagreed log entry. It breaks the poll if a number of heartbeats fail (§4.4). If no heartbeat fails, the backup thread then agrees on entries in the same total order as those on the leader's consensus log, using three steps. First (**B1**), it does a regular PAXOS view ID check [63] to see whether

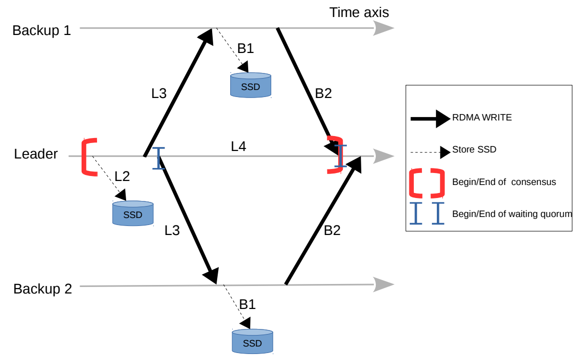

Figure 4: APUS *consensus algorithm in normal case.*

the leader's view ID matches its own one, it then stores the log entry in its local SSD. To scale to concurrently arriving requests, the backup thread scans multiple entries it agrees with at once. It then stores them in APUS's parallel storage (§5.1).

Second (**B2**), on each entry the backup agrees, the backup thread does an RDMA WRITE to send back a consensus reply to the `reply` array element in the leader's corresponding entry. Third (**B3**, not shown in Figure 4), the backup thread does a regular PAXOS check [63] on `last_committed` and to know the latest entry that has reached consensus. It then "executes" the committed entries by forwarding the data in these entries to the server program on its local replica. Carrying latest committed entries in next consensus requests is a common, efficient PAXOS implementation method [63].

To ensure PAXOS safety, the backup thread agrees on log entries in order without allowing any gap [63]. If the backup suspects it misses some log entries (e.g., because of packet loss), it invokes a learning request to the leader asking for the missing entries. We found one backup thread per backup suffices to achieve low overhead on concurrent connections (§7.3).

4.2 Atomic Message Delivery

On a backup side, one tricky challenge is that atomicity must be ensured on the leader's RDMA WRITEs on all entries and backups' polls. For instance, while a leader thread is doing a WRITE on `vs` to a remote backup, the backup's thread may be reading `vs` concurrently, causing a corrupted read value.

To address this challenge, one prior approach [37, 48] 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 [83] has shown that RDMA atomic primitives are much slower than normal RDMA WRITEs and local memory reads.

APUS tackles this challenge by using the leader to add a canary value after the `data` array. A backup thread always first checks the canary value according to `data_size` and then starts a standard PAXOS consensus reply decision [63]. This synchronization-free approach ensures that a APUS backup thread always reads a complete entry efficiently.

4.3 Handling Concurrent Connections

Unlike traditional PAXOS protocols which mainly handle single-threaded programs due to the deterministic execution assumption in SMR, APUS aims to support both single-threaded as well as multi-threaded or -processed programs running on multi-core machines. Therefore, a strongly consistent mechanism is needed to map each connection on the leader and its corresponding connection on backups. A naive approach is matching a leader connection's socket descriptor to the same one on a backup, but programs on backups may return nondeterministic descriptors due to systems resource contention.

Fortunately, PAXOS already makes viewstamps [63] of requests (log entries) 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.

4.4 Leader Election

Leader election on RDMA raises a main challenge: because backups do not communicate with each other in normal case, a backup proposing itself as the new leader 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 [76] establishes an extra control QP to handle leader election, complicating deployments.

APUS addresses this challenge with a simple, clean design. It runs leader election on the normal-case consensus log and QP. In normal case, the leader does WRITEs to remote logs as heartbeats with a period of T. Each consensus log maintains an `elect[MAX]` array, one array element for each replica. This `elect` array is only used in leader election. Once backups miss heartbeats from the leader for 3*T, they suspect the leader to fail, close the leader's QPs, and start to work on the `elect` array to elect a new leader.

Backups use a standard PAXOS leader election algorithm [63] with three steps. Each backup writes to its own `elect` element indexed by its replica ID on other replicas' `elect`. First, each backup waits for a random time (similar to random election timeouts in Raft [71]), and it proposes a new view with a standard two-round PAXOS consensus [55] by including both its view and the index of its latest log entry. The other backups also propose their views and poll on this `elect` array in order to agree on an earlier proposal or confirm itself as the winner. The backup with a more up-to-date log will win the proposal. 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 `elect` array. 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. For robustness, the above three steps are inherited from a practical PAXOS election algorithm [63], but APUS makes the election efficient and simple in an RDMA domain.

4.5 Correctness

APUS's protocol derives from a practical, viewstamp-based PAXOS protocol [63]. We made this design choice because PAXOS is notoriously difficult to understand [55, 56, 82], implement [28, 63], and verify [85, 41]. Deriving from a practical protocol [63] helps us incorporate these readily mature understanding and theoretically verified safety rules into APUS.

APUS's protocol complies with PAXOS *safety*: all replicas see the same total order of request entries in their local consensus logs. Below, we first prove APUS's safety is guaranteed with or without view change. We then prove APUS's safety on packet losses.

Normal Case. Normal case safety is guaranteed by three protocol steps. First, the leader assigns a total-ordered viewstamp for each incoming request, and each request log entry has a thread-safe, distinct position on the leader's log (in **L2**). Second, only the leader can write to other replicas' consensus logs, which enforces the same consensus log for all replicas; APUS does not allow an outdated leader to write to remote replicas (§4.4). Third, APUS guarantees the integrity of each log entry by its atomicity mechanism (§4.2). In normal case, APUS enforces same total order of log entries across replicas.

View Change. View change safety is ensured by two steps. First, APUS leader always carries a total-ordered viewstamp in its consensus request; an outdated leader can be detected by APUS's heartbeats and it is not allowed to write to remote replicas (§4.4). Second, every new leader's log contains all already-committed entries.

This is achieved by electing a new leader whose log is most up-to-date (§4.4).

Packet Loss. Although APUS's PAXOS protocol works on a RDMA network, the reliability of APUS does not rely on a loss-less RDMA. There are three types of RDMA packets in APUS: heartbeats, consensus requests, and consensus replies. First, loss of heartbeat packets does not affect the safety because it may only trigger view changes. Second, loss of consensus requests may cause entries missing in some backups' consensus logs. However, all backups agree on entries without allowing entry gaps, and they invoke learning processes to fetch missing entries (§4.1). Third, loss of consensus replies may make the leader fail to achieve consensus, but the leader will re-propose. In short, packet loss does not disturb APUS's safety.

APUS also has reasonable *liveness*. For instance, it uses random election timeout (§4.4), a practical technique [71] to minimize the probability of split votes [56].

4.6 Analytical Analysis on Performance

APUS is designed to be scalable to the number of concurrent client connections for general server programs. In contrast, a recent RDMA-based protocol DARE [76] is designed to achieve the lowest latency on a small number of connections for its own key-value store server. Below is an analytical analysis on APUS's consensus latency, and we compare APUS and DARE in §6.1.

Suppose the APUS leader has N client connections, and N requests arrive at the same time. APUS invokes consensus on all requests in the same way without distinguishing them as "read only" or "write". Suppose there are only three replicas.

According to the leader's four steps **L1**$\sim$**L4**, to reach consensus for all these N requests, the time taken on the leader's i_{th} request includes five parts: (1) an SSD storage time t_{SSD} in **L2** (each leader thread does a SSD store in parallel) ; (2) because an RDMA QP is a global data structure between every two replicas, pushing a message to a QP is serialized, which costs $i \times t_{PUSH}$ for i_{th} request; (3) a $\frac{1}{2}t_{RTT}$ in **L3**; (4) an SSD storage time t_{SSD} in **B1** for each backup (done by backups in parallel); and (5) a $\frac{1}{2}t_{RTT}$ in **B2**. On APUS's leader, the average consensus latency for all N requests sums up as the equation below:

$$\begin{aligned} APUS &= \left(\sum_{i=1}^{N}(2t_{SSD} + i \times t_{PUSH} + t_{RTT})\right)/N \\ &= 2t_{SSD} + \frac{(N+1)}{2}t_{PUSH} + t_{RTT} \end{aligned} \tag{1}$$

This equation shows that APUS's consensus latency is scalable to N because t_{PUSH} is often below 0.2 μs (§2.2).

5 Implementation Details

5.1 Parallel Input Logging Storage

To handle replica failovers, a PAXOS protocol must provide a persistent input logging storage. APUS uses the PAXOS viewstamp of each input log entry (§4.1) as key and the input data as value, and it persistently stores this key-value pair in Berkeley DB (BDB) by enabling thread-safety the BTree access method [21]. Our development found BTree fastest in BDB.

To be fast and scalable, APUS's input storage has two key features. First, if more inputs are inserted, the BTree height will grow, which will cause the key-value insertion latency to largely increase. To keep BTree height small, we implemented a parallel logging approach [22]: 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 (§7.4). Second, our storage has internal threads, which receive multiple log entries from a APUS backup thread (§4.1) and concurrently log them.

5.2 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 [31], 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.

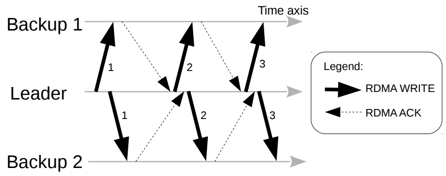

Figure 5: ***DARE's RDMA-based protocol.*** **It is a sole-leader, two-round protocol with three steps: (1) the leader WRITEs a consensus request to all backups' consensus logs and waits for ACKs to check if they succeed; (2) for the successful backups in (1), the leader does WRITEs to update tail pointer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

5.3 Network Output Checking Tool

Server programs often send replies with non-blocking IO. to align outputs across replicas, APUS uses 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. To compare a hash across replicas, the output checker uses APUS' input consensus protocol (§4.1). Because this protocol is invoked rarely, we did not observe its performance impact. The output checker is mainly for server programs' development purpose (§3.1).

6 Discussions

6.1 Comparing APUS with DARE

DARE [76] deviates from PAXOS due to its centralized, sole-leader protocol: in normal case, the leader does all consensus work via RDMA, and the other replicas are silent and do not consume CPU. Figure 5 shows DARE's protocol 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 backups are silent in both rounds, and only their RDMA NICs send back RDMA ACKs to the leader's NIC. Because the second round updates a global variable on every backup, which *serializes* all consensus requests, DARE is not designed to be scalable to concurrent connections.

DARE is mainly designed to achieve the lowest consensus latency on a small number of concurrent key-value connections. To this end, it has two clever features. First, on an input consensus, DARE needs to store the input only once on the leader, because its backups are silent. In current DARE implementation, leader does not store inputs and works purely in-memory. Second, it batches SET and GET requests separately. For GET requests, leader does only one-round RDMA READs to check view IDs from backups. Both DARE's evaluation and ours (§7.3) show that, when there were at most six concurrent connections, DARE achieved the lowest consensus latency in extant evaluation [46, 35, 45, 77, 58].

Despite the two features, the serialization problem in DARE still affects its scalability, especially when many SET and GET requests arrive concurrently. DARE's evaluation [76] confirmed this problem: on three replicas and nine concurrent connections, DARE's throughput on the 50% SET and 50% GET randomly arrival workload was 43.5% lower than that on the 100% SET workload. Our evaluation (§7.3) reproduces this problem when increasing the number of concurrent connections from 1 to 24: DARE's consensus latency increased approximately linearly to the number of connections; APUS's consensus latency was faster than DARE by 4.9X in average.

Overall, APUS differs from DARE in three aspects. First, APUS is a PAXOS protocol for general server programs; DARE is a novel, sole-leader consensus protocol for its own key-value store. Second, APUS is designed to be scalable on many concurrent client connections; DARE is mainly designed to achieve lowest consensus latency on a smaller number of connections. Third, APUS is a persistent protocol; DARE currently works purely in-memory. These differences show that APUS is more suitable for general server programs, and DARE more suitable for maintaining metadata.

6.2 APUS Limitations

APUS currently does not replicate physical times such as `time()` because these physical results are often explicit and easy to examine from network outputs (e.g., a timestamp in the header of a reply). Existing PAXOS approaches [51, 63] can be leveraged to intercept these functions and make programs produce the same results among replicas.

To replicate general client requests [76, 32], APUS totally orders all types of requests and it has not incorporated read-only optimization [51], because its performance overhead is already low (§7.4).

7 Evaluation

Evaluation was done on nine RDMA-enabled, Dell R430 hosts. Each host has Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 cores, 64GB memory, and 1TB SSD. NICs are Mellanox ConnectX-3 (40Gbps) connected with Infiniband [1]. All programs' unreplicated executions run on IPoIB (§2.2). Workloads run on idle replicas.

We compared APUS with five open source consensus protocols, including four traditional ones (libPaxos [78], ZooKeeper [6], CRANE [32] and S-Paxos [24]) and an

RDMA-based one (DARE [76]). 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 1: *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 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` [81], a widely studied SMR system for databases. We picked `Calvin` because: (1) it replicates inputs with a highly-engineered consensus protocol ZooKeeper [6], a good comparison target for APUS; and (2) it implements deterministic synchronization, which can make a program run deterministically. Table 1 shows workloads. The rest of this section focuses on five questions:

§7.1: Is APUS much faster than traditional consensus?
§7.2: How easy is APUS to integrate into SMR systems?
§7.3: How faster is APUS compared to DARE?
§7.4: What is the performance overhead of running APUS with server programs? How well does it scale?
§7.5: How well does APUS handle replica failures?

7.1 Comparing w/ Traditional Consensus

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 the number of concurrent connections and replicas are common high values [6, 32, 42, 76].

All four traditional protocols were run on IPoIB (§2.2). 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.

To find scalability bottlenecks in traditional protocols, we used only one client connection and broke down their consensus latency on leader (Table 2). From 3 to 9 replicas, the consensus latency (the "Latency" column) of these protocols increased more gently than that on 24 concurrent connections. For instance, when the number of replicas increased from three to nine, ZooKeeper latency increased by 30.3% with one connection; this latency increased by 168.3% with 24 connections (Figure 1). This indicates that concurrent consensus requests are the major scalability bottleneck for these protocols.

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 2: *Performance breakdown of traditional protocols on leader with only one connection.* 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 stacks, and JVM) between the "First" and "Major" reply. Times are in μs.

Specifically, three protocols had scalable latency on the arrival of their first consensus reply (the "First" column), which implies that network is not saturated. libPaxos is an exception because its two-round protocol consumed much bandwidth. However, on the 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.

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 on three replicas).

7.2 Integrating APUS into `Calvin`

The `Calvin`-APUS integration took 39 lines of code. `Calvin` currently uses ZooKeeper to batch inputs and then replicate them. To reduce response time, `Calvin`-APUS replicates each request immediately on its arrival. Figure 6 shows that the consensus latency of ZooKeeper was 7.6X higher than `Calvin`'s own request processing time, which indicates that ZooKeeper added a high overhead in `Calvin`'s response time. `Calvin`-APUS's response time was 8.2X faster than `Calvin`-ZooKeeper's because APUS's consensus latency was 45.7X faster than

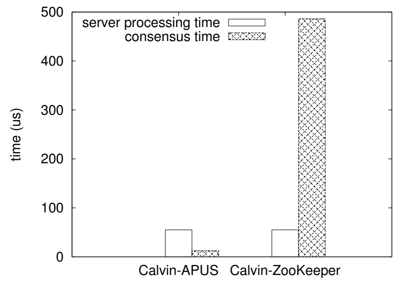

Figure 6: ***Comparing*** `Calvin`***-*****APUS** ***and*** `Calvin`***-ZooKeeper.***

ZooKeeper's. `Calvin`'s unreplicated execution throughput is 19825 requests/s, and `Calvin`-ZooKeeper was 16241 requests/s. `Calvin`-APUS was 19039 requests/s, a 4.1% overhead over `Calvin`'s unreplicated execution.

7.3 Comparing with DARE

Because DARE only supported a key-value server written by the authors, we ran APUS with `Redis`, a popular key-value server for comparison. Figure 7 shows APUS and DARE's consensus latency on variant concurrent connections. Both APUS and DARE ran seven replicas with randomly arriving, update-heavy (50% SET and 50% GET) and read-heavy (10% SET and 90% GET) workloads. DARE performance on two workloads were different because it handles GETs with only one consensus round [76]. APUS handles all requests with the same protocol. When there was only one connection, DARE achieved the lowest consensus latency we have seen in prior work because it is a sole-leader protocol (§6.1). On variant connections, APUS's average consensus latency was faster than DARE by 4.9X for two main 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

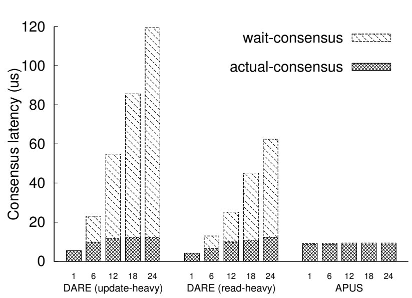

Figure 7: APUS *and DARE consensus latency (divided into two parts) on variant connections.* "Wait-consensus" is the time an input request spent on waiting consensus to start. "Actual-consensus" is the time spent on running consensus.

still slightly faster than DARE's on over six connections, because APUS avoids expensive ACK pollings (§2.2).

Second, DARE's second consensus round updates a global variable for each backup and serializes consensus requests (§6.1). 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 [76] confirmed such a high wait duration: with three replicas and nine concurrent connections, 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 (§4.1).

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 connections, 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, we found DARE better on smaller number of concurrent connections and replicas (e.g., metadata [26, 6]), and APUS better on larger number of connections or replicas (e.g., replicating server programs [42, 32]).

7.4 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 (§5.3) and didn't observe a performance impact. Only two programs (`MySQL` and OpenLDAP) have different output hashes caused by physical times (an approach [63] can be leveraged to enforce same physical times across replicas).

Figure 8 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 requests. Compared to these server programs' unreplicated executions, APUS merely incurred a mean throughput overhead of 4.2% (note that in Figure 8, 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 [42] 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.

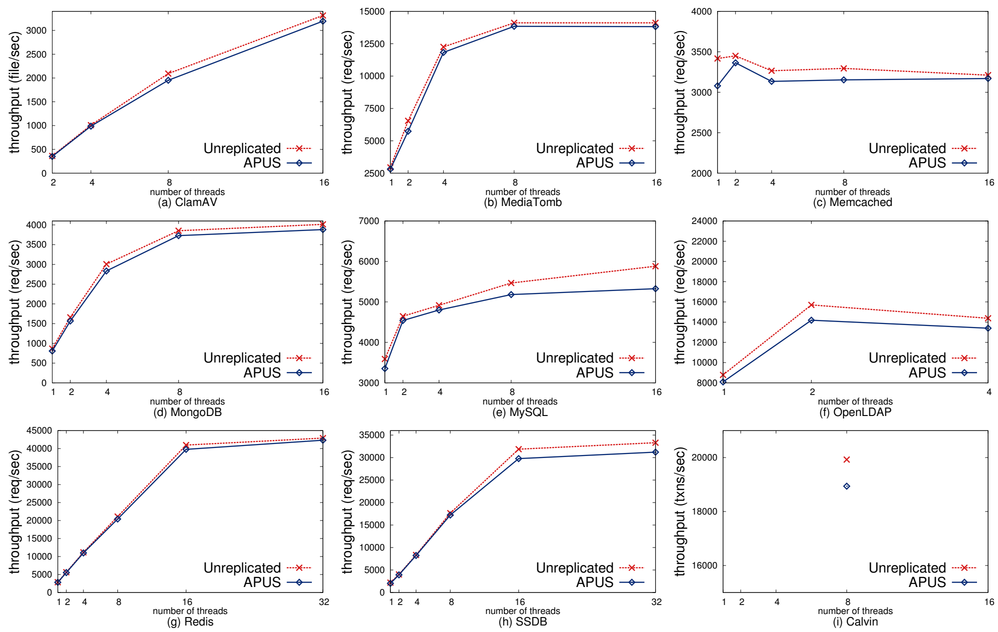

Figure 8: APUS ***throughput compared to server programs' unreplicated executions.***

Table 3 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 3) 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 the 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 3: ***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 3, APUS consensus latency took 8.8∼22.4 μs. This small consensus latency makes it achieve a low overhead of 4.3% on response time in Figure 9. Figure 7 and Table 3 also indicate that APUS had a low overhead of on programs' response time.

7.5 Checkpoint and Recovery

We ran same performance benchmark as in §7.4 and measured programs' checkpoint timecost. Each program checkpoint operation (§5.2) costs 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` with three replicas. 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 `Redis` throughput on the leader election approach (§4.4). APUS's default heartbeat period was 100 ms, and its three-round leader election took only 10.7 μs. `Redis` throughput is shown in Figure 10. After a new leader was elected, `Redis` throughput went up slightly because there were only two replicas left.

8 Related Work

Software-based consensus. There exist various PAXOS algorithms [63, 56, 55, 82, 69] and implementations [28, 63, 26, 32]. PAXOS is notoriously difficult to be fast and scalable [66, 50, 40], so server programs carry a weaker asynchronous replication approach (e.g., `Redis` [79]).

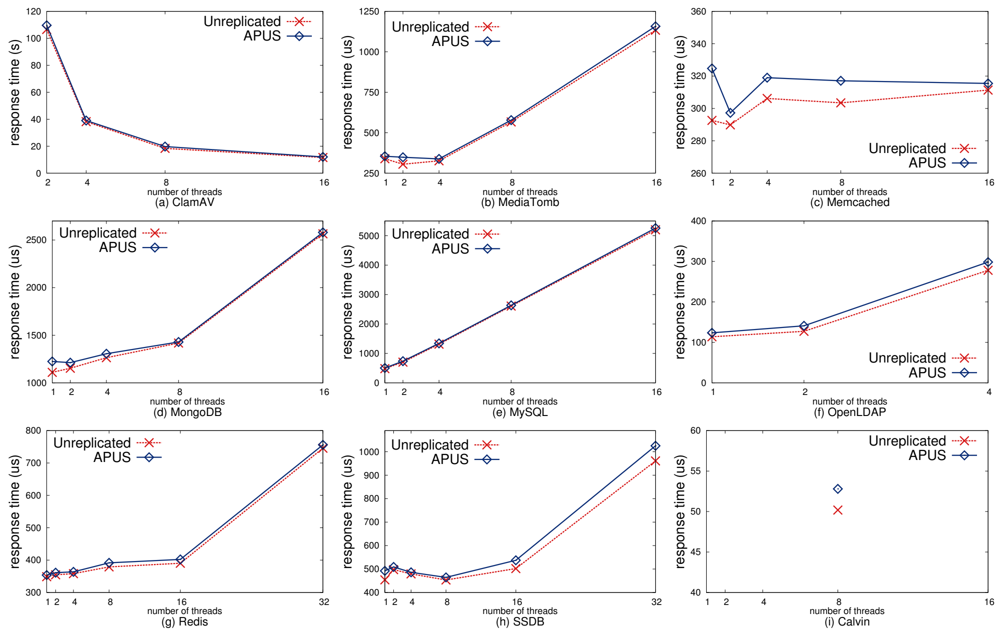

Figure 9: APUS *response time compared to server programs' unreplicated executions.*

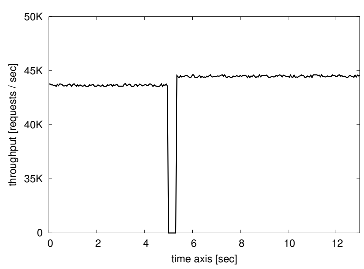

Figure 10: `Redis` *throughput on* APUS *leader election.*

Consensus is essential in datacenters [86, 43, 5] and worldwide distributed systems [30, 60], so much work is done to improve PAXOS's input commutativity [69, 61], understandability [71, 56], and verification [85, 41]. PAXOS is extended to tolerate byzantine faults [25, 64, 19, 27, 53, 62, 17, 16] and hardware faults [20].

Three SMR systems, Eve [51], Rex [42], and CRANE [32], 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 [46, 35, 45, 77, 58] leverage augmented network hardware or topology to improve PAXOS consensus latency. Three systems [46, 35, 45] implement consensus protocols in hardware devices (e.g., switches). "Consensus in a Box" [46] 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 [58] 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 [77] and NOPaxos [58] 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 [36, 57, 14]. 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 [39], key-value stores [67, 48, 37, 47], transactional systems [83, 38, 49], and file systems [84]. For instance, FaRM [37] runs on RDMA and it provides in a primary-backup replication [34, 72]. PAXOS provides better availability than primary-backup. These systems use RDMA to speed up different aspects, so they are complementary to APUS.

9 Conclusion

We have presented APUS, the first 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 SMR and improve the reliability of many 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. https://cs.stanford.edu/~matei/papers/2011/hotcloud_datacenter_os.pdf.

[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] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[15] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[16] 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.

[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] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] `http://www.clamav.net/`.

[30] 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.

[31] Criu. `http://criu.org`, 2015.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] A. A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM*, 2013.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] 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.

[44] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, 2010.

[45] 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.

[46] 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.

[47] 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)*, CC-GRID '12, 2012.

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

[49] 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.

[50] 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.

[51] 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.

[52] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[53] 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.

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

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

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

[57] M. Lee, S. Goldberg, R. R. Kompella, and G. Varghese. Fine-grained latency and loss measurements in the presence of reordering. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 329–340. ACM, 2011.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] `https://memcached.org/`.

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

[67] 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.

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

[69] 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.

[70] Nginx web server. `https://nginx.org/`, 2012.

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

[72] 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.

[73] `http://www.openldap.org/`.

[74] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[75] 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.

[76] 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.

[77] 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.

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

[79] `http://redis.io/`.

[80] `ssdb.io/`.

[81] 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.

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

[83] 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.

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

[85] 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.

[86] 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.