APUS: Fast and Scalable PAXOS on RDMA

1 Introduction

Existing PAXOS protocols suffer from high, scale-limited consensus latency. OS kernel, a major source of this problem, can be bypassed with advanced network features such as Remote Direct Memory Access (RDMA) within the same datacenter.

In this report, we present APUS, an RDMA-based PAXOS protocol and its runtime system that can efficiently provide fault tolerance to unmodified server programs. 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 such as atomic delivery of messages (§4.2).

2 Background on RDMA

RDMA is a kernel-bypassing technique that offers ultra low latency and high throughput. As the prices decrease, RDMA architectures (e.g., Infiniband [1] and RoCE [4]) have become common within a datacenter.

RDMA has three operation types, from fast to slow: one-sided read/write operations, two sided send/recv operations. An 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 [23] shows that one-sided operations are up to 2X faster than two-sided operations [19], so APUS uses one-sided operations (or "WRITE" in this report). 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 [18] shows that WRITEs on RC and UC QPs 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` [27]. 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: 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* [18]: 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

APUS deployment is similar to a typical State Machine Replication (SMR) system: it runs a server program on replicas within a datacenter. Replicas connect with each other using RDMA QPs. Client programs are 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 1 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*).

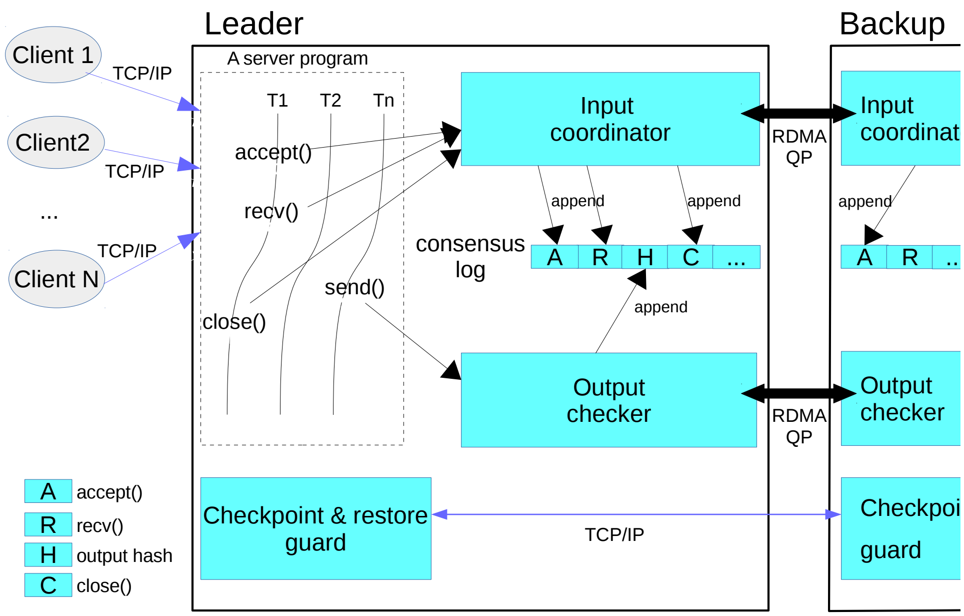

Figure 1: APUS *Architecture (key components are in blue).*

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).

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 [22], 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.

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.

4 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, a 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 2 depicts the format of a log entry in APUS's consensus log. Most fields are the same as those in a typical

```
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 2: APUS*'s log entry for each socket call.*

PAXOS protocol [22] except three: the `reply` array, `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 3 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 3) 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 [22]) for this entry in the consensus log, allocating a distinct entry in the log, and storing the entry to a local storage. 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 0.2μs in our evaluation. t_{PUSH} is serial for concurrently arriving requests on each QP, but the WRITEs (all **L3** arrows in Figure 2) 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 [22] to see whether 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.

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 3), the backup thread does a regular PAXOS check [22] 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 [22].

To ensure PAXOS safety, the backup thread agrees on log entries in order without allowing any gap [22]. 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.

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 [15, 18] 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

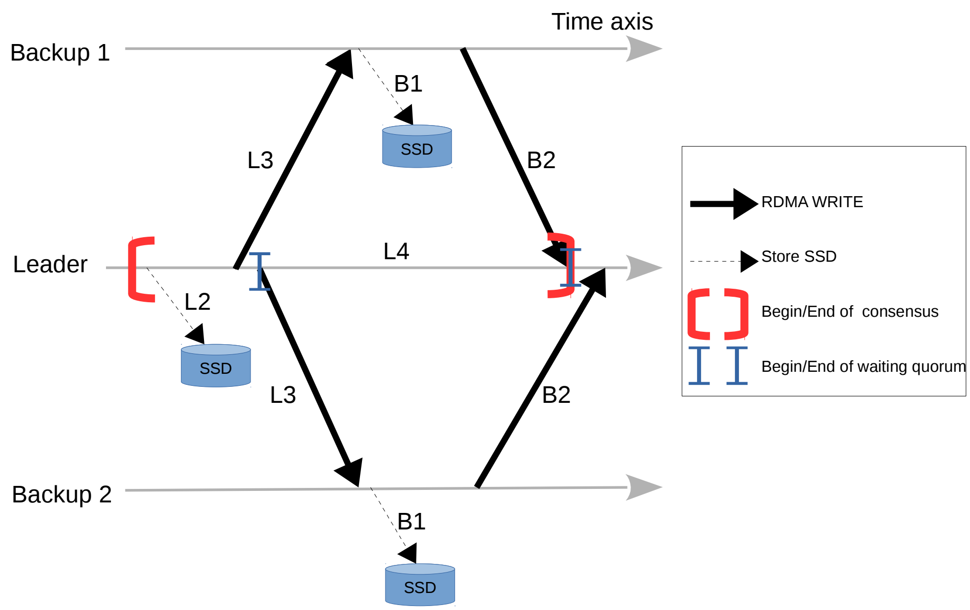

Figure 3: APUS *consensus algorithm in normal case.*

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 [36] 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 [22]. This synchronization-free approach ensures that a 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 [22] 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 [27] 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.

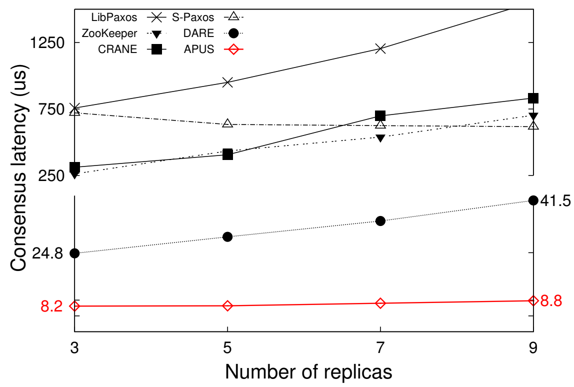

Figure 4: ***Comparing* APUS *to five existing consensus protocols.* All six protocols ran a client with 24 concurrent connections. The Y axis is broken to fit in all protocols.**

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 [22] 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 [26]), and it proposes a new view with a standard two-round PAXOS consensus [20] 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 [22], but APUS makes the election efficient and simple in an RDMA domain.

5 Initial Results

We implemented the normal case protocol and collected initial results.

Evaluation was done on nine RDMA-enabled Dell R430 and five Supermirco SuperServer 1019P hosts. Each host has Linux 3.16.0 and 2.6 GHz Intel Xeon CPU. The Dell R430 hosts are equipped with 24 hyperthreading cores, 64 GB memory, and 1 TB SSD. The SuperServer 1019P hosts have 28 hyperthreading cores, 32 GB memory, and 375GB SSD. All NICs are Mellanox ConnectX-3 (40Gbps) connected with RoCE [4].

We compared APUS with five open source consensus protocols, including four traditional ones (libPaxos [29], `ZooKeeper` [8], CRANE [12] and S-Paxos [11]) and an RDMA-based one (`DARE` [27]).

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` [35], a popular SMR system for databases.

5.1 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 [8, 12, 17, 27].

Figure 4 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.

5.2 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 5 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 [27]. 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. On variant connections, APUS's average consensus latency was faster than DARE by 4.9X.

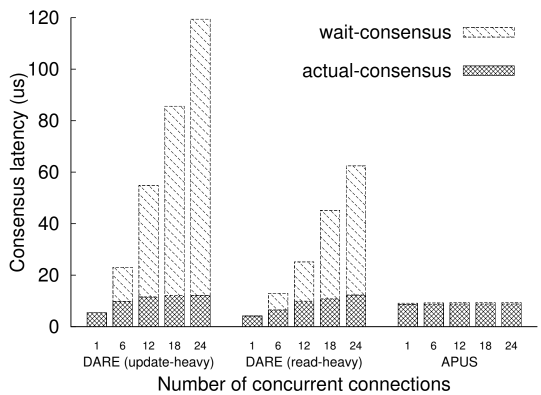

Figure 5: APUS *and DARE consensus latency (divided into two parts) on variant connections.* "Wait-consensus" is the time an input request spent on waiting consensus to start. "Actual-consensus" is the time spent on running consensus.

5.3 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 and didn't observe a performance impact. Only two programs (MySQL and OpenLDAP) have different output hashes caused by physical times (an approach [22] can be leveraged to enforce same physical times across replicas).

Figure 6 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 6, 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 [17] also showed that Memcached itself scaled poorly. Overall, APUS scaled as well as unreplicated executions on concurrent requests.

6 Integrating APUS into virtual machine

Primary-backup (e.g., REMUS [13]), a dominant Virtual Machine (VM) fault-tolerance approach, works in a physical time slot manner. In each slot, it runs a service in the primary VM to process client requests, tracks updated VM states (e.g., dirty memory pages), and buffers network outputs. At the end of a slot, a synchronization operation is invoked to transfer dirty pages from the primary to backup. Once the transfer succeeds, network outputs are sent to clients. By doing so, primary-backup approach ensures *external consistency* [13]: primary and backup have the same states and a primary failure will not be observed by clients.

Unfortunately, despite much effort [13, 6, 32, 21, 16], achieving fast and multi-core scalable VM fault-tolerance remains an open problem [14, 16, 32]. There are two main reasons for this. One is that the primary-backup approach often has to transfer an excessive amount of dirty memory pages, which greatly degrades the performance of a service and occupies prohibitive network bandwidth. The other reason is that primary-backup approach suffers from the

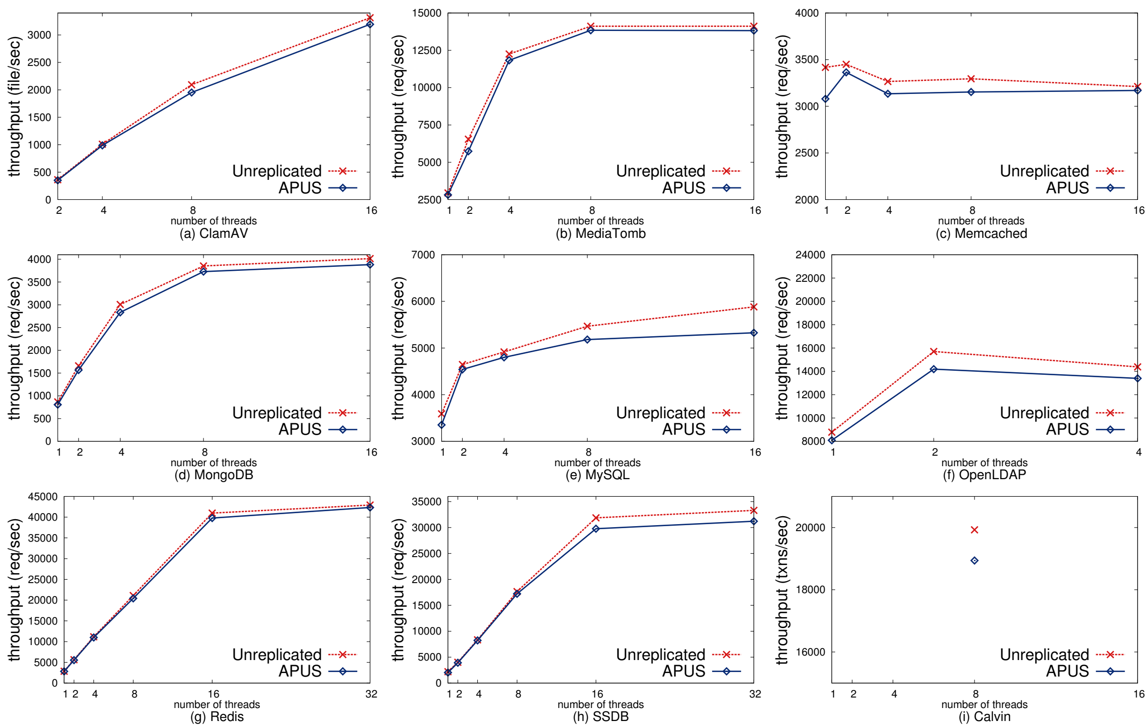

Figure 6: APUS *throughput compared to server programs' unreplicated executions.*

notorious "split-brain" problem [33], in which both the primary and backup think itself as the primary when network partition happens and therefore break external consistency to the clients.

To address the problems mentioned above, we propose Virtualized SMR (VSMR), a new SMR approach that can achieve fast, multi-core scalable VM fault-tolerance. VSMR enforces same total order of network inputs for a VM replicated across hosts. It then periodically invokes a synchronization operation to efficiently compute updated page hashes, to compare them across the replicas, and to transfer only the divergent pages. Because all the replicas run the same code and receive the same network inputs, their memory states should be largely the same in a synchronization operation and accordingly the amount of data to be transferred for memory state synchronization is expected to be small.

In a conceptual level, VSMR replicates an entire guest VM as a state machine and achieves the strengths of both SMR and primary-backup. By transferring only those divergent pages, VSMR automatically and efficiently ensures external consistency. Leveraging the powerful fault-tolerance of PAXOS, VSMR tackles the "split-brain problem" in primary-backup systems.

We implemented PLOVER, a prototype of VSMR system in Linux. PLOVER leverages APUS PAXOS system. PLOVER intercepts inbound network packets in the KVM QEMU hypervisor [30] and replicates them to other VM hypervisors using PAXOS. PLOVER's synchronization operation is built on top of PAXOS for robustness, and it uses RDMA to efficiently compare page hashes across replicas.

PLOVER develops several optimizations for improving the performance, including efficient determination of slot boundary and concurrent computation of dirty page hashes.

Similar to primary-backup for ensuring external consistency, APUS leader must buffer all outbound packets before a synchronization operation succeeds, including client responses and TCP ACKs. Client programs will stop sending new packets when their TCP congestion windows are met, even server programs have finished processing requests and become idle. This leads to unnecessary time slots. In practical workloads with concurrent connections, arrival times of requests are often unpredictable, thus a static synchronization time slot configuration (e.g., 25ms in REMUS and 100ms in MC) can often cause an idle service and unnecessary time slots. To avoid unnecessary time slots, APUS develops an adaptive-slot algorithm by inserting synchronization operations when its leader determines idle status of programs running in guest VM.

We also leveraged multi-core hardware and implemented a multi-threaded dirty page hash computing mechanism.

The mechanism detects the number of CPU cores on local host creates same number of threads to compute hashes of dirty physical pages since the last synchronization operation.

We evaluated PLOVER on 11 widely used programs, including 8 server programs (Redis [31], `SSDB` [34], `MediaTomb` [10], `Nginx` [25], `MySQL` [5], `Tomcat` [2], `PgSql` [28], and `Mongoose` [24]) and 3 dynamic language interpreters (PHP, Python, and JSP). To be close to real-world deployments, we group these programs into 7 practical services, including DjCMS [3], a large, sophisticated content management system (CMS) consisting of `Nginx`, Python, and `MySQL`.

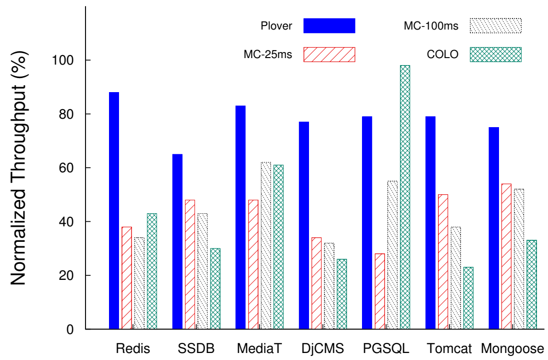

Figure 7: ***Throughputs normalized to unreplicated executions (4 vCPUs per VM)*. 100% means no overhead.**

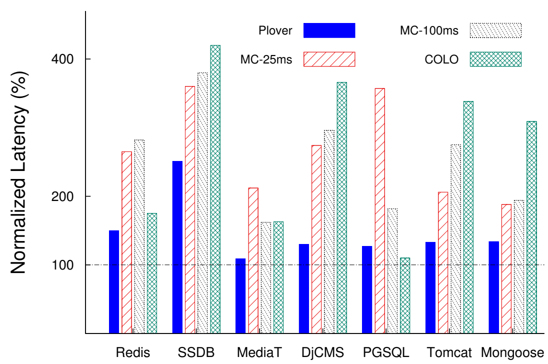

Figure 8: ***Response times normalized to unreplicated executions (4 vCPUs per VM)*. 100% means no overhead.**

For Redis and `SSDB`, each request contains a batch of 1K operations of 50% SET and 50% GET; for the other five services, each request contains one operation. We found sending operations in batches for Redis and `SSDB` made them reach peak throughput. For instance, when each request for Redis contains only one SET or GET operation, Redis's throughput is only 43K operation/s for 32 connections; when each request is a 1K-operation batch, its throughput reaches a peak value of 451K operation/s for 32 connections.

We compared PLOVER with two fault-tolerance systems: QEMU-MicroCheckpoint [6] (for short, MC), a REMUS-based primary-backup system carried in QEMU [30]; and COLO [14], a primary-backup system acquired and deployed by Huawei [9]. MC has an RDMA implementation [7], but it is being actively developed and not runnable on our hosts. We did not use REMUS because it was built before 2008 and did not run on our hosts. COLO runs the same service on both primary and backup, compares per-connection network outputs, and does a synchronization if any connection's output diverges.

Figure 7 shows PLOVER, MC, and COLO's throughput on 7 services. All results are normalized to unreplicated executions ran in KVM because the actual throughputs of different services largely vary. MC includes results for both the 25ms-slot (REMUS's default) and 100ms-slot (MC's default). All experiments ran on 4-vCPU per VM (unless specified) because COLO [14] and REMUS [13] evaluated up to 4 vCPUs per VM. On average, PLOVER's throughput is 2.1X higher than MC and COLO.

PgSql is the only service that PLOVER is slower than COLO. COLO compares per-connection outputs between its primary and backup, and it skips transferring memory if outputs did not diverge. PgSql ran SQL transaction workloads and its outputs were mostly the same. Except for PgSql, PLOVER was several times faster than COLO.

To analyze COLO, we also looked into SSDB, which had concurrent SET/GET requests. We found that COLO's output divergence was frequent when data dependencies exist among connections (i.e., GET requests frequently got different responses when SET and GET requests on the same key arrived at SSDB concurrently). When any output in any connection had an output divergence, COLO did a synchronization operation. COLO evaluation shows that it greatly slowed down when the number of client connections was large. PLOVER is not sensitive to outputs.

Figure 8 shows the response time of the four systems normalized to unreplicated executions. For five services (excluding Redis and SSDB), APUS's overhead of response time follows the same trend as the overhead of throughput because each client connection in these five services sends requests one by one. For Redis and SSDB, because the requests arrive in batches in order to saturate the two services, all four systems incur high overhead on response time. Specifically, PLOVER incurred the highest latency overhead for SSDB, because it had a low same dirty page rate of 77%.

Figure 9 explains why PLOVER's performance was higher. PLOVER consumes 12.2X less bandwidth than both MC and COLO on average. This reduction makes PLOVER the first VM fault-tolerance system that supports consolidating multiple VMs on a host.

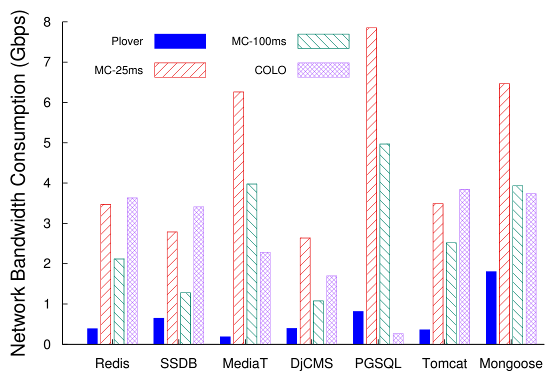

Figure 9: PLOVER *network bandwidth consumption compared with MC and* COLO *(all used four vCPUs per VM).*

7 Conclusion

APUS is an RDMA-based PAXOS system that can efficiently replicate general server programs. Evaluation on five consensus protocols and nine widely used programs shows that APUS is fast, scalable, and deployable. Besides, we have integrated APUS into virtual machine to greatly improve the reliability of VM. In this integration, we proposed VSMR, a novel SMR approach that makes VM fault-tolerance much faster and more scalable on multi-core. We have described PLOVER, a VSMR system implementation which leverages APUS SMR system and evaluation of PLOVER on a wide range of real-world server programs and services. PLOVER runs several times faster than two popular primary-backup systems and it saves much bandwidth.

Below, we summarize the novelty, practicability, and patent protectability of APUS and PLOVER respectively.

Novelty.

- APUS's runtime system uses *fast* RDMA primitives to invoke consensus processes on requests *concurrently* (§4.3). Besides, it addresses several practical issues including *atomic* delivery of RDMA messages (§4.2) and *transparent* replication. All these features make APUS a novel RDMA-based PAXOS protocol.

- *VSMR* is a novel SMR approach that makes VM fault-tolerance much *faster* and *more scalable* on multi-core. Leveraging the powerful fault-tolerance of PAXOS, VSMR tackles the "split-brain problem" in primary-backup systems. PLOVER is a VSMR system implementation which leverages APUS PAXOS system. PLOVER develops several optimizations for improving the performance, including including efficient determination of slot boundary and concurrent computation of dirty page hashes, which also make PLOVER a novel VM fault tolerance system.

Practicability.

- Our initial results show that APUS can provide fault tolerance to server programs with low performance overhead without requiring server developers' intervention (§5). Compared with nine server programs' unreplicated executions, APUS incurred 4.3% overhead in response time and 4.2% in throughput. Its consensus latency outperformed four traditional consensus protocols by at least 32.3X and faster than a recent RDMA-based consensus protocol DARE by 4.9X in average. This consensus latency stayed almost constant to the number of replicas and concurrent requests.

- On average, PLOVER's throughput is 2.1X higher than two popular VM fault-tolerance systems, MC and COLO, on 4-vCPU VMs, 4.3X higher on 16-vCPU VMs. Compared to unreplicated executions, APUS's overhead on response time is modest. APUS has reasonable CPU usage. Meanwhile, PLOVER consumes 12.2X less network bandwidth than both MC and COLO on average. It enables consolidating multiple fault-tolerant VMs on one host.

Patent Protectability. PLOVER's throughput is 2.1X higher than two widely-used VM fault-tolerance systems (MC and COLO) on 4-vCPU VMs, 4.3X higher on 16-vCPU VMs. At the same time, PLOVER consumes 12.2X less network bandwidth than both MC and COLO on average. Therefore, if a VM fault tolerance technique is as fast and scalable as PLOVER and consumes as little bandwidth as PLOVER, it is nearly impossible for it to achieve this without leveraging PLOVER's mechanism.

References

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

[2] Apache tomcat. `http://tomcat.apache.org/`.

[3] django cms - enterprise content management with django. `https://www.django-cms.org/en/`.

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

[5] MySQL. `http://www.mysql.com/`.

[6] QEMU MicroCheckpoint. `https://wiki.qemu.org/Features/MicroCheckpointing`.

[7] RDMA migration and rdma fault tolerance for QEMU. `http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf`.

[8] ZooKeeper. `https://zookeeper.apache.org/`.

[9] Huawei FusionSphere. `https://www.youtube.com/watch?v=yvsVuLAOhCo`, 2014.

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

[11] 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.

[12] 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.

[13] 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.

[14] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 3. ACM, 2013.

[15] 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.

[16] B. Gerofi and Y. Ishikawa. Rdma based replication of multiprocessor virtual machines over high-performance interconnects. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 35–44. IEEE, 2011.

[17] 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.

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

[19] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. Nov. 2016.

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

[21] M. Lu and T.-c. Chiueh. Speculative memory state transfer for active-active fault tolerance. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 268–275. IEEE Computer Society, 2012.

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

[23] 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.

[24] `https://code.google.com/p/mongoose/`.

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

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

[27] 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.

[28] Postgresql. `https://www.postgresql.org`, 2012.

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

[30] `http://www.qemu.org`.

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

[32] V. A. Sartakov and R. Kapitza. Multi-site synchronous vm replication for persistent systems with asymmetric read/write latencies. In *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*, pages 195–204. IEEE, 2017.

[33] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[34] `ssdb.io/`.

[35] 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.

[36] 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.