System Technical Report

Authors

We present APUS, an RDMA-based fault tolerance system that can efficiently replicate unmodified server programs.

1 Architecture

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.

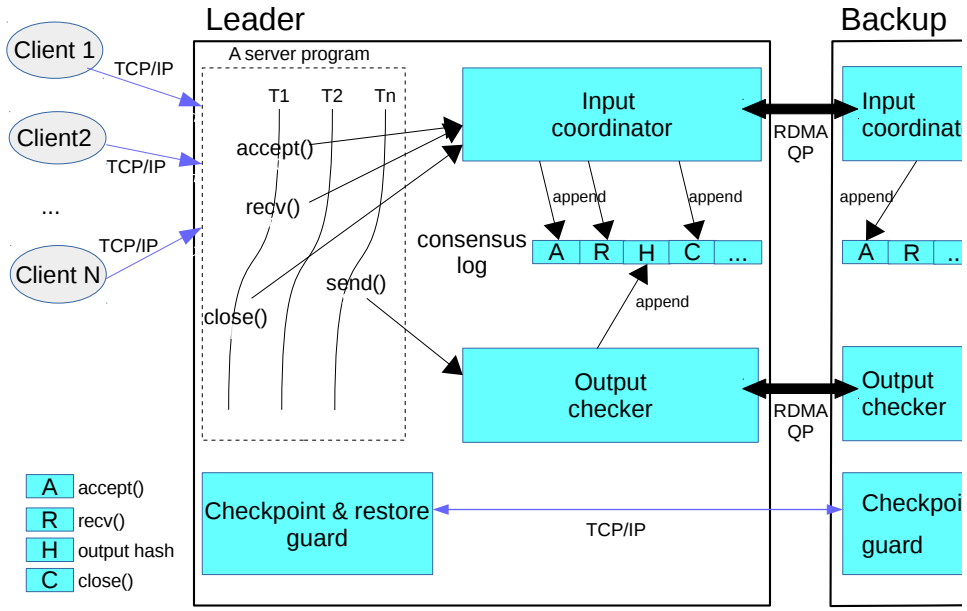

Figure 1: APUS *Architecture (key components are in blue).*

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*).

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 (§2).

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 [4], 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 (§2) to compare hashes across replicas.

2 Protocol

2.1 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.

```
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.*

Figure 2 depicts the format of a log entry in APUS's consensus log. Most fields are the same as those in a typical PAXOS protocol [4] 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 §2.3). 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 [4]) 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 (§2.4). 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.

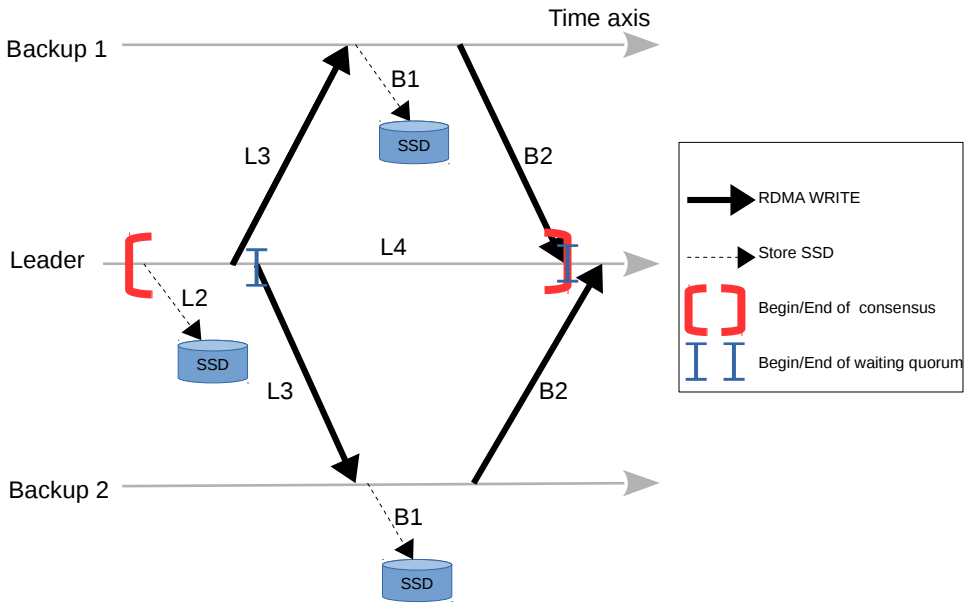

Figure 3: APUS *consensus algorithm in normal case.*

On each backup, a backup thread polls from the latest unagreed log entry. It breaks the poll if a number of heartbeats fail (§2.4). 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 [4] 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 [4] 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 [4].

To ensure PAXOS safety, the backup thread agrees on log entries in order without allowing any gap [4]. 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.

2.2 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 [1, 2] 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 [7] 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 [4]. This synchronization-free approach ensures that a APUS backup thread always reads a complete entry efficiently.

2.3 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 [4] 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.

2.4 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 [6] 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 [4] 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 [5]), and it proposes a new view with a standard two-round PAXOS consensus [3] 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 [4], but APUS makes the election efficient and simple in an RDMA domain.

References

[1] 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.

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

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

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

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

[6] 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.

[7] 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.