PAXOS Made More Scalable with RDMA

Paper #52

Abstract

PAXOS enforces a strongly consistent order of inputs for the same program replicated on a group of machines (or replicas), tolerating various failures. Therefore, PAXOS is served in many systems (e.g., ordering services). Unfortunately, despite much effort, the group size of traditional PAXOS protocols can hardly go up to a dozen because their consensus messages go throught TCP/IP layers, causing the consensus latency to increase almost linearly to the group size. This paper presents FALCON, a new RDMA-based PAXOS protocol. FALCON achieves scalability by: (1) carefully separating RDMA workloads among replicas; and (2) making all replicas receive consensus messages purely on local memory, just like making threads receive other threads' data efficiently on bare memory.

FALCON is the first to achieve low PAXOS consensus latency on over 100 replicas. Evaluation shows that FALCON's consensus latency outperforms four traditional PAXOS protocols by 32.3x ∼85.8x. When the group size increased from 3 to 105, FALCON's consensus latency increases merely from 8.1 μs to 31.6 μs, a sub-linear increase. FALCON is faster than a recent RDMA-based PAXOS protocol by up to 2.2x. FALCON runs with 9 widely used, unmodified server programs (e.g., Redis and `MySQL`) with low overhead. FALCON is deployable: all source code and raw evaluation results are available at `github.com/nsdi17-p52/falcon`.

1 Introduction

PAXOS [57, 52, 51, 74]) plays a core role in datacenters and distributed systems, including ordering services [59, 46, 38], leader election [5, 22], and fault-tolerance [47, 40, 27]. A PAXOS service runs the same program on a group of replicas and enforces a strongly consistent order of inputs for this program, as long as a quorum (typically, majority) of replicas still behave correctly.

Due to this strong fault-tolerance, PAXOS is widely served in many systems. For instance, Scatter [38] runs 8∼12 replicas in each PAXOS group to order client requests, and it lets replicas respond requests in parallel. A bigger group size will improve Scatter throughput. Moreover, recent state machine replication (SMR) systems [27, 47, 40] use PAXOS to greatly improve the availability of general server programs.

Unfortunately, despite these advances, the high PAXOS consensus latency makes many systems suffer. For efficiency, PAXOS typically assigns one replica as the leader to invoke consensus requests, and the other replicas as backups to agree on requests. To agree on an input, at least one message round-trip is required between the leader and a backup. A round-trip causes big latency as it goes through TCP/IP layers such as software network stack and OS kernel. This latency could be acceptable for leader elections [22, 5] or heavyweight transactions [27, 47], but undesirable for key-value stores [70, 58].

As replica group size grows, PAXOS consensus latency increases drastically [38] due to the linearly increasing number of consensus messsages. To improve scalability, one approach is introducing parallel techniques such as multithreading [5, **?**] or asynchrous IO [27, 69]. However, the high latency of TCP/IP round-trips still exist, and the synchronizations in these techniques frequently involve expensive OS events such as context switches. We ran four PAXOS-like protocols [5, 27, **?**, 69] on 40Gbps network with only one client sending consensus requests, and we found that: when replica group size increased from 3 to 9, the consensus latency of these protocols increased by 30.3% to 156.8%, and 36.5% to 63.7% of the increase was in OS kernel.

Another scalability approach is maintaining multiple instances of PAXOS, including partitioning program states [38, **?**, **?**], splitting consensus leadership [56, **?**], and hieratical replication [46, 38]. However, the core of these systems, PAXOS, still scales poorly [59, 38, 46].

Fortunately, Remote Direct Access Memory (RDMA) becomes a possible scalability solution as it becomes common in datacenters. RDMA not only bypasses OS kernel, but it also provides dedicated network hardware to achieve fast round-trip. For instance, the fastest RDMA operation allows a process to directly write to a remote replica's process without involing the remote OS kernel or CPU ("one-sided" operation). To ensure an RDMA write successfully resides in the remote memory, local process polls an ACK sent from remote NIC. Such a RDMA round-trip takes only about 3 μs [60].

However, due to the unrichness of RDMA primitives, fully exploiting RDMA speed in software systems is widely considered challenging [60, 45, 33, 68]. For instance, DARE [68] presents a sole-leader, RDMA-based PAXOS protocol, which lets the leader does all RDMA workloads and backups do nothing. This protocol was

fast with 3∼5 replicas. However, our evaluation shows that, as the group grows, the leader met scalability bottlenecks (e.g., polling ACKs), and its consensus latency increased by 10.6x as the group grows by 35x (§8).

Our key observation is that we should carefully separate RDMA workloads among the leader and backups, especially in such a scalability-sensitive RDMA context. For instance, we should let both leader and backups do RDMA writes directly on desination replicas' memory, and let all replicas poll their own local memory to receive messages. Although doing so will consume more CPU resources than a sole-leader approach, it brings two benefits. First, the leader has less workloads.

Second, both leader and backups participate in consensus; they can get rid of the expensive RDMA ACK polling and just receive consensus messages on their bare, local memory. An analogy is threads receiving other threads' data via bare memory, a fast and scalable multithreading pattern.

This observation may raise reliability issues because now the leader has no clue on whether the remote writes succeed. Fortunately, PAXOS's protocol already tolerates various reliability issues, including message losses caused by hardware or program failures. Now one just needs a runtime system to efficiently ensure the atomicity of remote writes among replicas (§4.1).

We present FALCON,[1] a new RDMA-based PAXOS protocol and its runtime system. In FALCON, all replicas directly write to desination replicas' memory and poll messages from local memory to receive messages, and our runtime system handles other technical challenges such as messsage atomiciy (§4.1), efficient input logging (§6.1), and failure recovery (§6.2).

Similar to general SMR systems [40, 27], FALCON's design supports general, unmodified server programs. Within FALCON, a program runs as is. FALCON automatically deploys this program on replicas of machines, intercepts inputs from a server program's inbound socket calls (e.g., `recv()`), and invokes its PAXOS protocol to enforce same inputs across replicas.

We implemented FALCON in Linux. FALCON intercepts POSIX inbound socket calls (e.g., `accept()` and `recv()`) to coordinate inputs using the Infiniband RDMA architecture [1]. To improve the assurance that replicas run in sync, on top of FALCON's PAXOS protocol, it also provides an efficient network output checking protocol, a practical feature that may promote FALCON's deployments. To recover or add new replicas, FALCON leverages CRIU [26] to periodically a program on one backup, so it does not affect other replicas to reach consensus efficiently.

We compared FALCON with five popular, open source PAXOS-like implementations, including four traditional ones (libPaxos [69], ZooKeeper [5], CRANE [27] and S-Paxos [**?**]) and a RDMA-based one (DARE [68]). We also evaluated FALCON on 9 widely used or studied server programs, including 4 key-value stores (Redis [70], `Memcached` [58], `SSDB` [72], and `MongoDB` [61]), one SQL server `MySQL` [12], one antivirus server `ClamAV` [24], one multimedia storage server `MediaTomb` [11], one LDAP server OpenLDAP [66], and one transactional database `Calvin` [73]. Our evaluation shows that

1. FALCON achieves one order of magnitude higher scalability and one order of magnitude faster latency than the literature (Figure 1). FALCON's consensus latency outperforms 4 popular PAXOS implimentations by 32.3x to 85.8x on 3 to 9 replicas. FALCON is faster than DARE by about 2.2x. When changing the replica group size from 3 to 105 (a 35x increase), FALCON's consensus latency increases merely from 8.1 μs to 31.6 μs (a 2.9x, sub-linear increase).
2. FALCON is general and easy to use. For all 9 evaluated programs, FALCON ran them without any modification except `Calvin`.
3. FALCON incurs low overhead on all 9 evaluated server programs. With 9 replicas, compared to servers' own unreplicated executions, FALCON incurred merely 4.16% overhead on throughput and 4.28% on response time in average.
4. FALCON is robust. On leader failures, FALCON's leader election latency was reasonable and scalable.

Our major contribution is a new RDMA-based PAXOS protocol that achieves low consensus latency on over 100 replicas. FALCON has the potential to largely improve both the scale and performance of existing PAXOS services [38, 46, 27, 40]. For instance, Scatter [38] previously deploys 8∼12 replicas in each PAXOS group, and now it can deploy hundreds of replicas in each group and achieves much higher throughput. Moreover, a general and deployable service, FALCON may largely promote PAXOS deployments and improve the consistency and fault-tolerance of various systems in datacenters.

FALCON is deployable: all source code, benchmarks, and raw evaluation results are available at `github.com/nsdi17-p52/falcon`. The remaining of this paper is organized as follows. §2 introduces background on PAXOS and RDMA. §3 gives an overview of FALCON. §4 presents FALCON's consensus protocol with its runtime system. §5 describes the output checking protocol. §6 presents implementation details. §7 compares DARE with FALCON and discusses FALCON's current limitations. §8 presents evaluation results, §9 discusses related work, and §10 concludes.

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

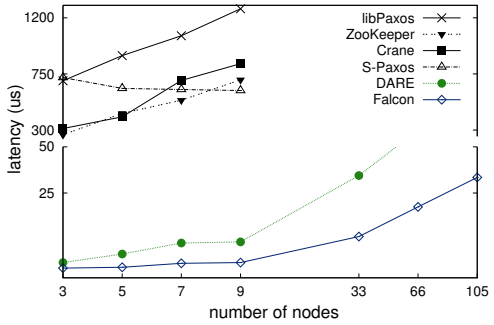

Figure 1: *Consensu latency of six* PAXOS*-like protocols.* All protocols ran with 20 concurrent proposing clients.

2 Background

This section introduces PAXOS (§2.1) and RDMA (§2.2).

2.1 PAXOS

PAXOS [74, 52, 51, 23, 53, 57] runs the same program and its data on a group of replicas and enforces a strongly consistent sequence of inputs across replicas. Because a consensus can be achieved as long as a majority of replicas agree, PAXOS is well known for tolerating various faults, including minor replica failures and packet losses due to hardware or program errors. If the leader replica fails, PAXOS elects a new leader from the backups.

To cope with replica failovers, PAXOS replicas must log inputs in local stable storage. When a new input comes, the PAXOS leader writes this input in local stable storage. The leader then starts a new consensus round among replicas. A backup also writes the received consensus request in local storage if it agrees on this request. The latency of logging inputs is scalable because each replica does logging locally.

The consensus latency of traditional PAXOS protocols is notoriously high and unscalable. As datacenters incorporate increasingly faster networking hardware and more CPU cores, traditional PAXOS protocols [69, **?**, **?**, 40, 5] are having fewer performance bottlenecks on network bandwith and CPU resources. However, these protocols still run on TCP/IP and have to go through various software layers such as network stack and OS kernel. Arrakis [**?**] reported that a `ping` program spent over 70% latency in these two layers.

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

2.2 RDMA

RDMA architecture such as Infiniband [1] or RoCE [2] recently becomes common in datacenters due to its ultra low latency, high throughput, and its decreasing prices. The ultra low latency of RDMA not only comes from its kernel bypassing feature, but also its dedicated network stack implemented in hardware. Therefore, RDMA is considered the fastest kernel bypassing technique [45, 60, 68]; it is several times faster than software-only kernel bypassing techniques (e.g., DPDK [**?**]).

RDMA has three types of communication primitives, from fastest to slowest: one-sided read/write operations, two sided send/recv operations, and IPoIB (IP over Infiniband). One-sided operations is about 2X faster than two-sided operations because two-sided operations actually consist of two one-sided operations [60]. A one-sided RDMA read/write operation can directly write from one replica's memory to a remote replica's memory, completely bypassing OS kernel and CPU of the remote replica. For brevity, the rest of this paper denotes a one-sided RDMA write operation as a "WRITE".

RDMA communications between a local network interface card (NIC) and remote NIC requires setting up a Queue Pair (QP), including a send queue and a receive queue. Each QP associates with a Completion Queue (CQ) to store ACKs. A QP belongs to a type of "XY": X can be R (reliable) or U (unreliable), and Y can be C (connected) or U (unconnected). HERD [45] reported that WRITEs on RC and UC OPs incur negligible difference in latency, so FALCON uses RC QPs.

To ensure a remote replica is alive and a WRITE succeeds, a common RDMA practice is that after a WRITE is pushed to a QP, the local replica polls an ACK from the associated CQ before it continues (the so called *signaling*). Polling ACK is time consuming as it involves synchronization between the NICs on both sides of a CQ. We collected the time taken in polling ACKs in a recent RDMA-based PAXOS protocol DARE [68], and we found that, although DARE has been carefully optimized (its leader maintains one global CQ to receive backups' ACKs in batches), polling ACKs still became a scalability bottleneck: when the CQ was empty, it took 61$\sim$79 μs; when the CQ has one or more ACKs randomly arrived from other replicas, it took 260$\sim$410 μs. As the number of ACKs is linear to the replica group size, polling ACKs is a major scalability bottleneck (§8).

Fortunately, depending on the application logic, we can do *selective signaling* [45]: it only checks for an ACK after pushing a number of WRITEs (previous WRITEs may already succeed before this ACK-checking starts). Because FALCON's protocol semantic does not rely on RDMA ACKs, it uses selective signaling to occasionally clean up ACKs.

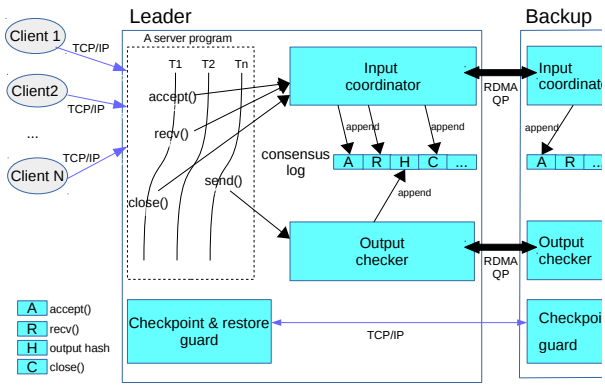

Figure 2: *The* FALCON *Architecture.* FALCON components are shaded (and in blue).

3 FALCON Overview

FALCON follows a typical PAXOS runtime system deployment [38, 47, 40, 27, 68]. It runs replicas of a server program in a datacenter. Replicas connect with each other using RDMA QPs. Client programs located in LAN or WAN. The leader handles client requests and runs our RDMA-based PAXOS protocol to coordinate inputs among replicas.

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

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

The coordinator runs a consensus algorithm (§4), which WRITEs the local entry to backups' remote logs in parallel and polls the local log entry to wait quorum. When a quorum is reached, the leader thread simply finishes intercepting this call and continues with its server execution. As the leader's server threads execute more calls, FALCON enforces the same consensus log and thus the same socket call sequence across replicas.

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

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

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

This output consensus is based on the input consensus algorithm (§4) except that backups carry their hash at the same hash index back to the leader. For this particular output consensus, the leader first waits quorum. It then waits for a few μs in order to collect more remote hashes. It then compares remote hashes it has.

If a hash divergence is detected, the leader optionally invokes the local guard to forward a "rollback" command to the diverged replica's guard. The diverged replica's guard then rolls back and restores the server program to a latest checkpoint before the last successful output check (§5). The replica then restores and re-executes socket calls to catch up. Because output hash generations are fast and an output consensus is invoked occasionally, our evaluation didn't observe performance impact on this checker.

4 The RDMA-based PAXOS Protocol

This section presents FALCON's consensus protocol, including the RDMA-based consensus algorithm in normal case (§4.1), handling concurrent connections (§4.2), leader election (§4.3), and reliability guarantees (§4.4).

4.1 Normal Case Algorithm

Recall that FALCON' input consensus protocol contains three roles. First, the PAXOS consensus log (§3). Second, a leader replica's server program thread (in short, a leader thread) which invokes consensus request. For efficiency, FALCON lets a server program's threads directly handle consensus requests whenever they call inbound socket calls (e.g., `recv()`). Third, a backup replica's FALCON internal follower thread (§3) which agrees on or rejects consensus requests.

Figure 3 shows the format of a log entry in FALCON's consensus log. Most fields are regular as those in a typical PAXOS protocol [57] except three ones: the `reply` array, the client connection ID `conn_vs`, and the type ID of a socket call `call_type`. The `reply` array is for backups to WRITE their consensus replies to the leader. The `conn_vs` is for identifying which connection this socket call belongs to (see 4.2). The `call_type` identifies four types of socket calls in FALCON: the `accept()` type (e.g., `accept()`), the `recv()` type (e.g., `recv()` and `read()`), the `send()` type (e.g., `send()` and `write()`), and the `close()` type (e.g., `close()`).

Figure 4 shows the input consensus algorithm. Suppose a leader thread invokes a consensus request when it calls a socket call with the `recv()` type. A consensus request includes four steps. The first step (**L1**) is executing the actual Libc socket call, because FALCON needs to get the actual return values or received data bytes of this call and then replicates them in remote replicas' logs.

The second step (**L2**) is local preparation, including assigning a global, monotonically increasing viewstamp to locate this entry in the consensus log, building a log entry structure for this call, and writing this entry to its local SSD.

The third step (**L3**) is to WRITE a log entry to remote backups in parallel. Unlike a previous RDMA-based consensus algorithm [68] which has to wait for ACKs from remote NICs, our WRITE immediately returns after pushing the entry to its local QP between the leader and each backup, because PAXOS has handled the reliability issues (e.g., packet losses) for our WRITEs. In our evaluation, pushing a log entry to local QP took no more than 0.3 μs. Therefore, the WRITEs to all backups are done in parallel (see **L3** in Figure 3).

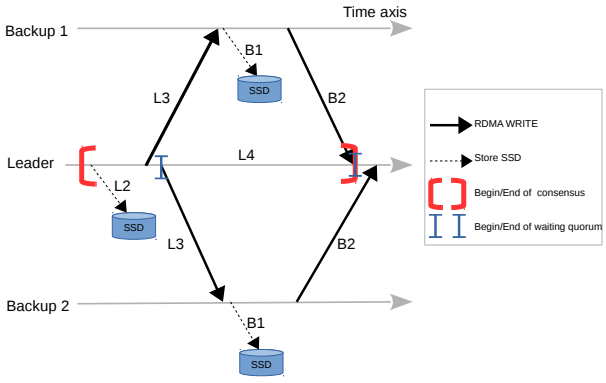

Figure 4: FALCON *consensus algorithm in normal case.*

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

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

To address this issue, one existing approach [32, 45] leverages the left-to-right ordering of RDMA WRITEs and puts a special non-zero variable at the end of a fix-sized log entry because they mainly handle key-value stores with fixed value length. As long as this variable is non-zero, the RDMA WRITE ordering guarantees that the log entry WRITE is complete. However, because FALCON aims to support general server programs with largely variant received data lengths, this approach cannot be applied in FALCON.

Another approach is using atomic primitives provided by RDMA hardware, but a prior evaluation [76] has shown that RDMA atomic primitives are much slower than normal RDMA WRITEs and local memory reads.

FALCON tackles this issue by adding a canary value after the actual `data` array. Because FALCON uses a QP with the type of RC (reliable connection) (§2), the follower always first checks the canary value according to `data_size` and then starts a standard PAXOS consensus reply decision [57]. Our efficient, synchronization-free approach guarantees that the follower always reads a complete log entry.

A follower thread in a backup replica polls from the latest un-agreed log entry and does three steps to agree on consensus requests, shown in Figure 4. First (**B1**), it does a regular PAXOS view ID checking to see whether the leader is up-to-date, it then stores the log entry in its local SSD. Second (**B2**), it does a WRITE to send back a consensus reply to the leader's `reply` array element according to its own node ID. Backups perform these two steps in parallel (see Figure 3).

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

ing the socket calls to the local server program. This forwarding faithfully builds and closes concurrent connections between the follower and the local server program according to the socket calls in the consensus log.

In an implementation level, FALCON stores log entries in local SSD using Berkley DB [19]. Although FALCON's algorithm does not wait every RDMA ACK in order to achieve high scalability, we use selective signaling (§2) to occasionally check and clear ACKs in the CQ associated with the QP, an essential ACK clearing step in RDMA implementations.

4.2 Handling Concurrent Connections

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

Fortunately, PAXOS already makes viewstamps [57] of socket calls strongly consistent across replicas. For TCP connections, FALCON adds the `conn_vs` field, the viewstamp of the the first socket call in each connection (i.e., `accept()`) as the connection ID for log entries. Then, FALCON maintains a hash map on each local replica to map this connection ID to local socket descriptors.

4.3 Leader Election

Compared to traditional PAXOS leader election protocols, RDMA-based leader election poses one main issue caused by RDMA. Because backups do not communicate frequently with each other in normal case, thus a backup does not know the remote memory locations where the other backups are polling. Writing to a wrong remote memory location may cause the other backups to miss all leader election messages. An existing system [68] establishes an extra control QP with extra remote memory to handle leader election, posing more complexity via the extra communication channels.

FALCON addresses this issue with a simple, clean approach. It runs a leader election with the same consensus log and the same QP. In normal case, the leader does WRITEs to remote logs as heartbeats with a period of T. Each consensus log maintains a control data structure called `elect[MAX]`, one element for each replica. Normal case operations and heartbeats use the other parts of the consensus log but leave this `elect` array alone. Once backups have not received heartbeats from the leader for a period of 3*T, they start to elect a new leader and let their follower threads poll from the `elect` array.

Backups start a standard PAXOS leader election algorithm [57] with three steps. Each replica writes to its own `elect` element at remote replicas. First, backups propose a new view with a standard two-round PAXOS consensus [51] by including both the view and the index of the latest log entry. The other backups also propose their views and poll on this array in order to follow other proposals or confirm itself as the winner. The backup whose log is more up-to-date will win. A log is more up-to-date if its latest entry has either a higher view or the same view but a higher index.

Second, the winner proposes itself as a leader candidate using this array, another two-round PAXOS consensus. Third, after the second step reaches a quorum, the new leader notifies remote replicas itself as the new leader and it starts to WRITE periodic heartbeats.

4.4 Reliability Guarantee

To minimize protocol-level bugs, FALCON's PAXOS protocol mostly sticks with a popular, practical implementation [57], especially the behaviors of senders and receivers (§4.1 and §4.3). For instance, both FALCON's normal case algorithm and the popular implementation [57] involve two messages and same senders and receivers (although we use WRITEs and carefully make them run in parallel). We made this choice because PAXOS is notoriously difficult to understand [65, 51, 52, 74] or implement [23, 57] verify [78, 39]. Aligning with a practical PAXOS implementation [57] helps us incorporate these readily mature understanding, engineering experience, and the theoretically verified safety rules into our protocol design and implementation.

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

5 Output Checking Protocol

Most server programs are multithreaded and they may run into nondeterminism (e.g., concurrency errors [55]), which may cause replicas to diverge. FALCON provides a fast output checking protocol for a practical purpose: improving FALCON deployers' assurance on whether replicas run in sync. If diverged replicas are detected, deployers can restore the them (§6.2).

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

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

To compare a hash across replicas, FALCON's output checking protocol runs the same as the input coordination protocol (§4.1) except that the follower thread on each backup replica carries this hash value in the `reply` written back into the leader's corresponding log entry.

6 Implementation Details

This section first presents our parallel input logging mechanism (§6.1) for storing inputs efficiently, and then our checkpoint/restore mechanism for recovering and adding replicas (§6.2).

6.1 Parallel Input Logging

To handle replica failovers, a standard PAXOS protocol should provide a persistent input logging storage. FALCON uses the PAXOS viewstamp of each input as key and its input data as value. FALCON stores this key-value pair in Berkely DB (BDB) with a BTree access method [19], because found this method fastest in our evaluation.

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

To handle this issue, we implementated a thread-safe, parallel logging approach [**?**]: instead of maintaining a single BDB store, we maintain an array of BDB stores. We use an index to indicate the current active store and insert new inputs. Once the number of insertions reach a threshold, we move the index to the next empty store in the array and recycle the preceding stores. This Implementation efficiently kept our input logging latency within 2.8∼8.7 μs (§8.2).

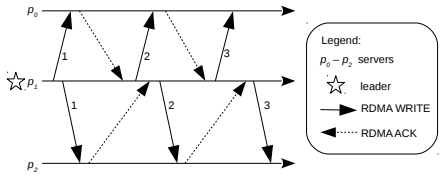

Figure 5: *DARE's RDMA-based protocol.* This is a sole-leader, two-round protocol with three steps: (1) the leader WRITEs a consensus request to all backups' consensus logs and waits for ACKs to check if they succeed; (2) for the successful backups in (1), leader does WRITEs to update tail pointer of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

6.2 Checkpoint and Restore

We proactively design FALCON'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 duration T_{ckpt} (one minute by default).

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

7 Discussions

This section compares FALCON and DARE's protocols (§7.1), and discusses FALCON's limitations (§7.2).

7.1 Comparing FALCON and DARE

We highly appreciate DARE [68], the first RDMA-based, PAXOS-like protocol. It is most relevant to FALCON. To

tolerate single point of program failures, DARE is designed to run with a small (three to five) replica group. Under this design choice, DARE's protocol is sole-leader (leader does all the consensus work; backups do nothing): the leader only needs to do two-round WRITEs and check the ACKs of these WRITEs for a consensus. Both rounds are essential because DARE's leader needs the second round to indicate the latest (tail) undergoing consensus request on remote replicas for a future new leader. Figure 5 shows DARE's protocol.

To further improve performance, DARE makes two technical choices. First, to avoid delays caused by polling ACKs, DARE uses a global RDMA CQ (Completion Queue) for all replicas, making it possible to collect multiple ACKs each poll operation. Second, this protocol does not incorporate a persistent storage or checkpoint/restore design. Therefore, DARE lacks durability (§2), an important guarantee in traditional PAXOS protocols and in FALCON.

However, DARE is not designed to scale to a large replica group because its leader does all the consensus work. Our evaluation shows that both DARE's ACK pollings and its two-round consensus incurred scalability bottlenecks and had a approximately linear consensus latency: DARE's consensus latency increased by 10.6x as replica group size increased by 35x (§8).

Overall, FALCON differs from DARE in three aspects. First, FALCON's protocol has only one RDMA round (Figure **??**), while DARE has two rounds. To achieve one-round consensus, FALCON's backups poll from its local memory to receive consensus requests, so FALCON consumes more CPU than DARE on backlups. Second, FALCON has shown to scale well (sub-linearly) on 100+ nodes (§**??**), while DARE did not discuss their scalability in paper. Third, although FALCON's protocol includes a persistent storage to ensure durability, which DARE lacks, FALCON was faster than DARE by about 2.2x. §**??** analyzes FALCON and DARE performance.

7.2 FALCON Limitations

FALCON currently does not hook random functions such as `gettimeofday()` and `rand()` because these random results are often explicit and easy to examine from network outputs (e.g., a timestamp in the header of a reply). Existing approaches [47, 57] in PAXOS protocols can also be leveraged to intercept these functions and make general programs produce same results among replicas.

Like recent systems [68, 27], FALCON totally orders all types of requests and it has not incorporated read-only optimization [47], because its performance overhead compared to the evaluated programs' unreplicated executions is already reasonable (§8.2). However, FALCON can be extended to support read-only optimization if two conditions are met: (1) whether the semantic an operation is read-only is clear in a server program; and (2) the number of output bytes for this operation is fixed. GET requests in key-value stores often meet these two conditions.

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

8 Evaluation

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

Our evaluation machines also include one Dell R320 server for client programs. It has Linux 3.16.0, 2.2GHz Intel Xeon 12 hyper-threading cores, 32GB memory, and 160GB SSD. To mitigate latency of client requests, this client machine is located at the same LAN as the RDMA replicas with a 1Gbps NIC. The average `ping` latency between this machine and a RDMA replica is 241 μs. Note that which machines to run cilents on do not affect any PAXOS protocol's consensus latency (it is only affected by the RDMA network among replicas).

We compared FALCON with five popular, open source PAXOS-like implementations, including four traditional ones (libPaxos [69], ZooKeeper [5], CRANE [27] and S-Paxos [**?**]) and a RDMA-based one (DARE [68]). S-

Program	Benchmark	Workload/input description
`ClamAV`	clamscan [7]	Files in `/lib` from a replica
`MediaTomb`	ApacheBench [10]	Transcoding videos
`Memcached`	mcperf [6]	50% set, 50% get operations
`MongoDB`	YCSB [9]	Insert operations
`MySQL`	Sysbench [8]	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 performance benchmark program. Workloads are all concurrent.

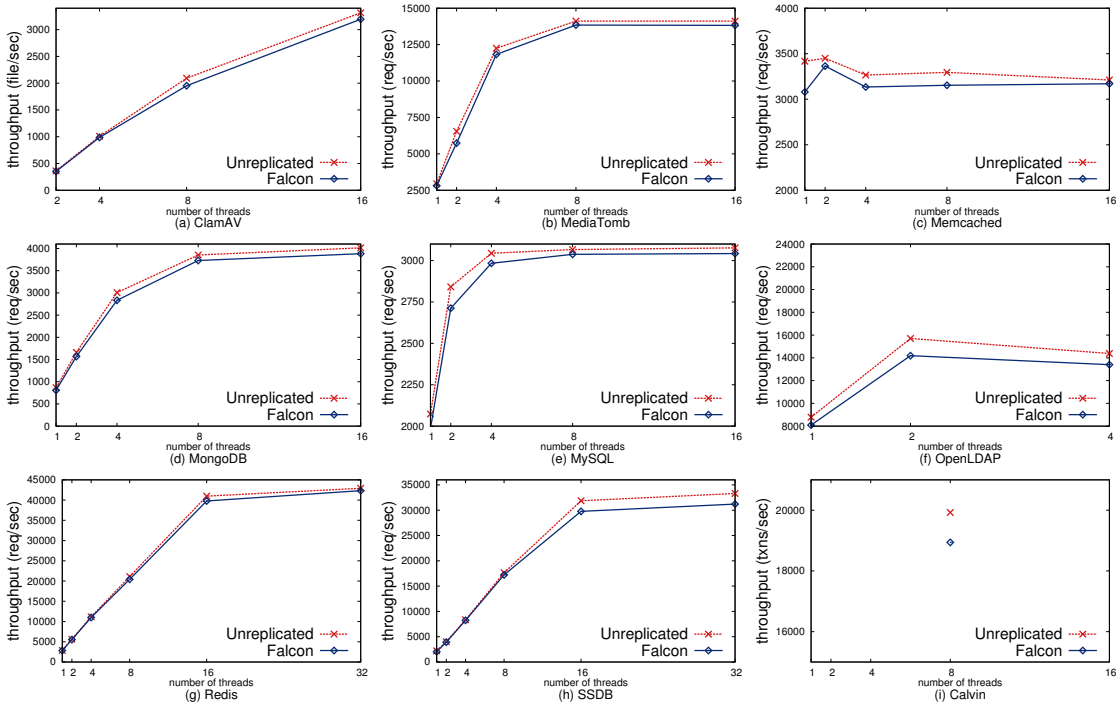

Figure 6: FALCON *throughput compared to the unreplicated execution.*

Paxos is designed to achieve scalable throughput when more replicas are added.

We evaluated FALCON on 9 widely used or studied server programs, including 4 key-value stores Redis, `Memcached`, `SSDB`, `MongoDB`; `MySQL`, a SQL server; `ClamAV`, an anti-virus server that scans files and delete malicious ones; `MediaTomb`, a multimedia storage server that stores and transcodes video and audio files; OpenLDAP, an LDAP server; `Calvin`, a widely studied transactional database system. All these programs are multithreaded except Redis (but it can serve concurrent requests via Libevent). These servers all update or store important data and files, thus the strong PAXOS fault-tolerance is especially attractive to these programs.

Table 1 introduces the benchmarks and workloads we used. To evaluate FALCON's practicality, we used the protocol or program developers' own performance benchmarks or popular third-party benchmarks. For benchmark workload settings, we used the benchmarks' default workloads whenever available. To perform a stress testing on FALCON's input consensus protocol, we chose workloads with significant portions of writes, because write operations often contain more input bytes than reads (e.g., a key-value SET operation contains more bytes than a GET).

The rest of this section focuses on these questions:

§8.1: What is FALCON's consensus latency compared to traditional PAXOS protocols?

§**??**: What is FALCON's consensus latency compared to DARE?

§8.2: What is the performance overhead of running FALCON with general server programs? How does it scale with concurrent requests?

§8.3: How fast is FALCON on handling checkpoints and electing a new leader?

8.1 Comparing with Traditional PAXOS

As a common evaluation practice in PAXOS systems [68, **?**], when comparing FALCON with other PAXOS protocols, we ran FALCON with a popular key value store Redis. For all six PAXOS protocols, we spawned 20 consensus requests, a common high concurrent value in prior evaluation [5, 27, 40]. Our evaluation also showed that most server programs reached peak performance at this concurrent value (8.2).

We compared FALCON with four traditional protocols, libPaxos [69], ZooKeeper [5], CRANE [27] and S-Paxos [**?**]. Figure 1 has already shown the results. Three traditional protocols incurred almost a linear increase of consensus latency except S-Paxos. S-Paxos batch requests from replicas and it only invoke consensus when a fixed batch is full. More replicas will make this batch be full more quickly, so S-Paxos incurred slightly better consensus latency at more nodes. Nevertheless, its consensus latency was over 700 μs. FALCON's consensus latency outperforms these protocols by 32.3x to 85.8x on 3 to 9 replicas. Therefore, we needn't run these traditional protocols on more replicas.

To understand why traditional protocols are unscal-

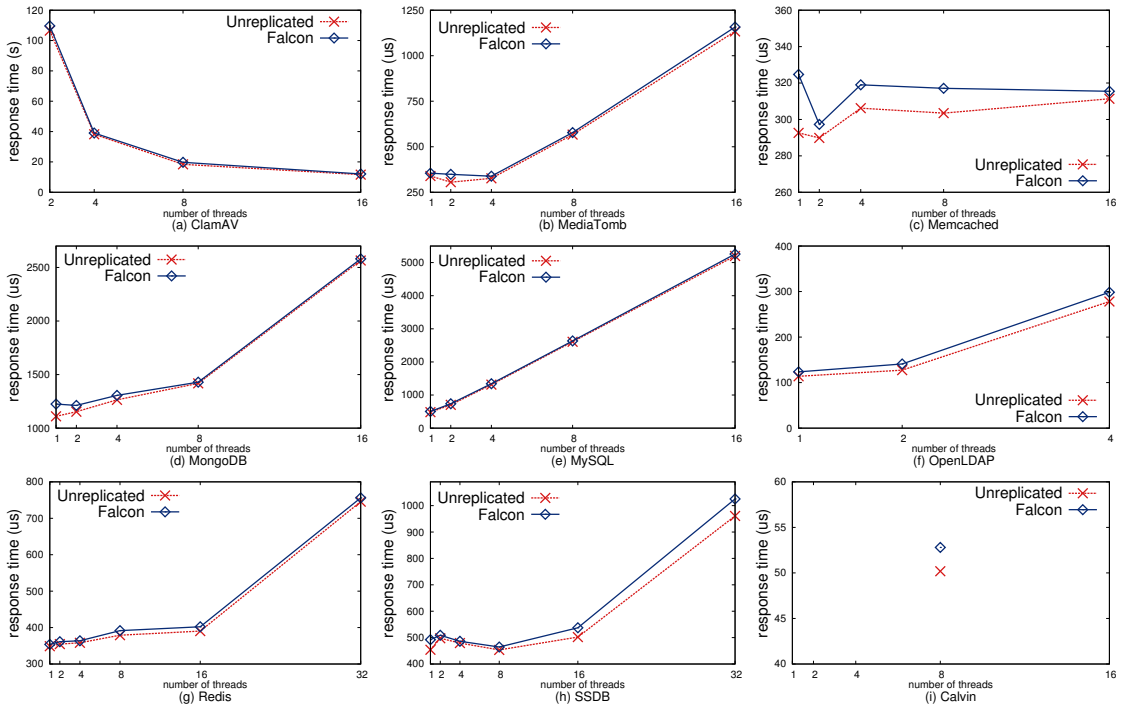

Figure 7: FALCON *response time compared to the unreplicated execution.*

able, we ran only one client with them and inspect the micro events in their protocols, shown in Table 2. Three protocols had scalable latency on the arrival of their first consensus reply (the "First" column), which means that network bandwith is not a scalability bottleneck for them. libPaxos is an exception because its two-round protocol consumed much bandwidth. However, there is a big gap between the arrival of the first consensu reply and the "majority" reply (the "Major" column). Given that the reply CPU processing time was small (the "Process" column), we can see that the various systems layers, including OS kernels, network libraries, and language runtimes (e.g., JVM) are the scalable bottleneck (the "Sys" column). This indicates that RDMA is useful to bypass the systems layers for better scalability.

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

We first compared FALCON and DARE [68] with the key-value stores. FALCON uses a widely used one, Redis; DARE uses 335-line key-value store written by their authors. To thoroughly analyze their latency with scalability, we run more replicas than the nine physical machines (i.e., each machine runs several replicas). Since our RDMA NIC bandwidth is 40Gpbs, we didn't find network a bottleneck when running more replicas on a machine; from 9 to 105 replicas for both protocols, each RDMA round-trip increased merely from 2.6 μs to 4.4 μs.

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	30.3
ZooKeeper-9	129.0	76.0	128.0	3.6	49.4
CRANE-3	78.0	69.0	69.0	13.0	0.1
CRANE-9	148.0	83.0	142.0	30.0	35.0
S-Paxos-3	865.1	846.0	846.0	20.0	0.1
S-Paxos-9	739.1	545.0	731.0	35.0	159.1

Table 2: *Scalability bottleneck analysis in traditional* PAXOS *protocols.* The "Proto-#Rep" column means the PAXOS protocol name and replica group size; "Latency" means the consensus latency; "First" means the latency of its first consensus reply; "Major" means the latency of its the majority reply; "Process" means time spent in processing all replies; and "Sys" means time spent in systems (OS kernel, network libraries, and JVM) between the "First" and the "Major" reply. All times are in μs.

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

Table 3: *Consensus latency of* FALCON *and DARE.* The update-heavy workload consist of 50% SETs. The read-heavy workload consist of 10% SETs.

Table 4 shows the consensus latency of FALCON and DARE on the same update-heavy workload, which contains half GETs and half SETs. This workload represents real-world applications such as an advertisement log that records recent user activities [68]. FALCON and DARE achieves similar latency at three replicas, and FALCON was much faster than DARE on more replicas. Their difference is 2.5x∼3.3x on over 33 replicas. When changing replica group size from 3 to 105 (a 35x increase), FALCON's consensus latency merely increased by 3.8x, while DARE increased by 11.7x.

FALCON scales better than DARE for two main reasons. First, in a protocol level, FALCON's protocol carefully separate the RDMA workloads across leader and backups, and it is a one-round protocol (§4.1). DARE lets its leader do all the consensus work and backups do nothing, and it is a two-round protocol (§7.1). DARE involves approximately 2x more RDMA communications than FALCON.

Second, in a RDMA communication primitive level, FALCON lets all replicas receive consensus messages on their bare, local memory. DARE frequently polls RDMA ACKs from a RDMA Completion Queue (CQ) in each consensus round. An ACK polling or insertion operation on the CQ involves synchronization between RDMA NICs among replicas. We found that although DARE's ACK mechanism is already highly optimized with a global CQ, so that it can poll more ACKs at one time. However, we found that ACKs arrived randomly, and each polling in DARE took up to 410 μs (§2). The more replicas, the more ACK polling operations were needed.

FALCON does the same consensus round on all types of requests. DARE does two special handling on GET requests. First, it batchs GET requests for consensus, which may improve throughput but aggravate latency. Second, DARE's GET requests only involve one-round consensus (does RDMA reads to fetch all replicas's PAXOS view IDs [57] and check if a majority of them match the leader's). We also ran DARE with a read-heavy workload (Table 4) and its consensus latency was about 50% slower than FALCON on large replica groups. Note that FALCON has a durable input storage, and DARE is a volatile protocol. Overall, we considered FALCON faster and more scalable than DARE.

8.2 Performance Overhead

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

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

Figure 6 shows FALCON's throughput and Figure 7 response time. We varied the number of concurrent client connections for each server program by from one to 32 threads. For `Calvin`, we only collected the 8-thread result because `Calvin` uses this constant thread count in their code to serve client requests. Overall, compared to these server programs' unreplicated executions, FALCON merely incurred a mean throughput overhead of 4.16% (note that in Figure 6, the Y-axises of most programs start from a large number). FALCON's mean overhead on response time was merely 4.28%.

As the number of threads increases, all programs' unreplicated executions got a performance improvement except `Memcached`. A prior evaluation [40] also observed a similar `Memcached` low scalability. FALCON scaled almost as well as the unreplicated executions.

FALCON achieves such a low overhead in both throughput and response time mainly because of two reasons. First, for each `recv()` call in a server, FALCON's input coordination protocol only contains two one-sided RDMA writes and two SSD writes between each leader and backup. A parallel SSD write approach [20] may further improve FALCON's SSD performance. Second, FALCON's output checking protocol invokes occasionally (§**??**).

Program	# Calls	Input	SSD time	Quorum time
`ClamAV`	30,000	37.0	7.9 μs	10.9 μs
`MediaTomb`	30,000	140.0	5.0 μs	17.4 μs
`Memcached`	10,016	38.0	4.9 μs	7.0 μs
`MongoDB`	10,376	490.6	7.8 μs	9.2 μs
`MySQL`	10,009	28.8	5.1 μs	7.8 μs
OpenLDAP	10,016	27.3	5.5 μs	6.4 μs
Redis	10,016	40.5	2.8 μs	6.3 μs
`SSDB`	10,016	47.0	3.0 μs	6.2 μs
`Calvin`	10,002	128.0	8.7 μs	10.8 μs

Table 4: ***Leader's input consensus events per 10K requests, 8 threads.*** **The "# Calls" column means the number of socket calls that went through FALCON input consensus; "Input" means average bytes of a server's inputs received in these calls; "SSD time" means the average time spent on storing these calls to stable storage; and "Quorum time" means the average time spent on waiting quorum for these calls.**

To deeply understand FALCON's performance overhead, we collected the number of socket call events and consensus durations on the leader side. Table 4 shows these statistics per 10K requests, 8 or max (if less than

8) threads. According to the consensus algorithm steps in Figure 4, for each socket call, FALCON's leader does an "L2": SSD write (the "SSD time" column in Table 4) and an "L4": quorum waiting phase (the "quorum time" column). L4 implies backups' performance because each backup stores the proposed socket call in local SSD and then WRITEs a consensus reply to the leader.

By adding the last two columns in Table 4, a FALCON input consensus took only 9.1 μs (Redis) to 22.4 μs (`MediaTomb`). This consensus latency mainly depends on the "Input" column: the average number of data bytes received in socket calls (e.g., `MongoDB` has the largest received bytes). FALCON's small consensus latency makes FALCON achieve reasonable throughputs in Figure 6 and response times Figure 7.

8.3 Checkpoint and Recovery

We ran the same performance benchmark as in §8.2 and measure programs' checkpoint timecost. Each FALCON periodic checkpoint operation (§6.2) cost 0.12s to 11.6s on the evaluated server programs, depending on the amount of modified memory and files in the server programs since their own last checkpoint. `ClamAV` incurred the largest checkpoint time (11.6s) because it loaded and scanned files in a `/lib` directory.

Checkpoint operations did not affect FALCON's performance in normal case because they were done on only one backup, and the leader and other backups can still reach consensus rapidly. This indicates PAXOS's fault-tolerance strength: a backup restart or failure does not affect consensus.

To evaluate FALCON's PAXOS recovery feature, we ran FALCON with Redis and manually kill one backup, and we did not observe a performance change in the benchmark runs. We then manually killed the FALCON leader and measured the latency of our RDMA-based leader election with three rounds (§4.3). Figure 5 shows FALCON's election latency from three to eleven replicas. Because PAXOS leader election is rarely invoked in practice, although FALCON's election latency was slightly higher than its normal case consensus latency, we considered it reasonable.

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

Table 5: FALCON*'s scalability on leader election.*

9 Related Work

State machine replication (SMR). SMR is a powerful, but complex fault-tolerance technique. The literature has developeed a rich set of PAXOS algorithms [57, 52, 51, 74, 63] and implementations [23, 57, 22]. PAXOS is notoriously difficult to be fast and scalable [59]. To improve speed and scalability, various advanced replication models have been developed [63, 56, 38, 46]. Since consensus protocols play a core role in datacenters [79, 42, 4] and distributed systems [25, 56], a variety of study have been conducted to improve different aspects of consensus protocols, including performance [63, 53, 68], understandability [65, 52], and verifiable reliability rules [78, 39]. Although FALCON tightly integrates RDMA features in PAXOS, its implementation mostly complies with a popular, practical approach [57] for reliability. Other PAXOS approaches can also be leveraged in FALCON.

Five systems aim to provide SMR or similar fault-tolerance guarantees to server programs and thus they are the most relevant to FALCON. They can be divided into two categories depending on whether their protocols run on TCP/IP or RDMA. The first category runs on TCP/IP, including Eve [47], Rex [40], Calvin [73], and Crane [27]. Evaluation in these systems shows that SMR services incur modest overhead on server programs' throughput compared to their unreplicated executions. However, for some other programs (e.g., key-value stores) demanding a short response time, FALCON is more suitable because these systems' consensus latency is at least 10X slower than FALCON's (§7.1).

Notably, Eve [47] presents an execution state checking approach based on their PAXOS coordination service. Eve's completeness on detecting execution divergence relies on whether developers have manually annotated all thread-shared states in program code. FALCON's output checking approach is automated (no manual code annotation is needed), and its completeness depends on whether the diverged execution states propagate to network outputs. Eve and FALCON's checking approaches are complementary and can be integrated.

RDMA-based techniques. RDMA techniques have been implemented in various architectures, including Infiniband [1], RoCE [2], and iWRAP [3]. RDMA have been leveraged in many systems to improve application-specific latency and throughput, including high performance computing [36], key-value stores [60, 45, 32, 44], transactional processing systems [76, 33], and file systems [77]. These systems are largely complementary to FALCON. It will be interesting to investigate whether FALCON can improve the availability for both the client and server for some of these advanced systems within a datacenter, and we leave it for future work.

Nondeterminism. Nondeterminism [49, 31, 15, 30, 64, 54, 29, 28, 14] is pervasive in both application programs and OS kernels, and it often comes with concurrency bugs [55]. To mitigate nondeterminism, deterministic multithreading techniques [18, 54, 14, 21, 31, 64, 15, 17, 43, 16] and deterministic replay techniques [41, 37, 71, 34, 48, 75, 35, 67, 50, 13, 62] have been developed. Much of these techniques can greatly

improve software reliability, but they often come with a performance slowdown. FALCON can run these techniques with the server program to mitigate replica divergence caused by concurrency bugs.

10 Conclusion

We have presented FALCON, a fast, scalalbe RDMA-based PAXOS protocol with its runtime system. Evaluation on popular PAXOS protocols and widely used server programs shows that FALCON is fast, scalable, and robust. FALCON has the potential to improve the consistency and fault-tolerance of various systems in datacenters. FALCON is deployable: all source code, benchmarks, and raw evaluation results are available at `github.com/nsdi17-p52/falcon`.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

[14] 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.

[15] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, Mar. 2010.

[16] T. Bergan, L. Ceze, and D. Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 677–692. ACM, 2013.

[17] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[18] E. Berger, T. Yang, T. Liu, D. Krishnan, and A. Novark. Grace: safe and efficient concurrent programming. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 81–96, Oct. 2009.

[19] `http://www.sleepycat.com`.

[20] 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.

[21] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '09)*, pages 97–116, Oct. 2009.

[22] 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.

[23] 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.

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

[25] 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. Oct. 2012.

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

[27] 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.

[28] 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.

[29] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 337–351, Oct. 2011.

[30] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Oct. 2010.

[31] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, Mar. 2009.

[32] 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.

[33] 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.

[34] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, Dec. 2002.

[35] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, Mar. 2008.

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

[37] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

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

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

[40] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European*

Conference on Computer Systems (EUROSYS '14), page 11. ACM, 2014.

[41] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[42] 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.

[43] N. Hunt, T. Bergan, , L. Ceze, and S. Gribble. DDOS: Taming nondeterminism in distributed systems. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, 2013.

[44] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

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

[46] 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.

[47] 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.

[48] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[49] O. Laadan, N. Viennot, C. che Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Oct. 2011.

[50] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

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

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

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

[54] 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.

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

[56] 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.

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

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

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

[60] 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.

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

[62] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In

Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09), pages 73–84, Mar. 2009.

[63] 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.

[64] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, Mar. 2009.

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

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

[67] 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.

[68] 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.

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

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

[71] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[72] `ssdb.io/`.

[73] 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.

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

[75] `http://www.vmware.com/solutions/vla/`.

[76] 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.

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

[78] 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.

[79] 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.