PAXOS Made Much More Scalable with RDMA

Paper #52

Abstract

Consensus protocol (typically, PAXOS) enforce a strongly consistent order of inputs for the same program replicated on a group of machines (or replicas), tolerating various failures. Therefore, PAXOS is widely served in numerous systems, including ordering and fault-tolerance services. Unfortunately, despite much effort, the group size of traditional PAXOS protocols can hardly go up to a dozen, because traditionally consensus messages go throught TCP/IP and thus consensus latency will increase almost linearly to the replica group size. This paper presents FALCON, a fast, scalable PAXOS runtime system by fully exploiting Remote Direct Memory Access (RDMA) features. Our key idea to achieve scalability is making PAXOS replicas receive consensus messages *purely on local memory*, just like making threads communicate with other threads efficiently via bare memory.

FALCON is the first to achieve low PAXOS consensus latency with high scalability (100+ replicas). Evaluation shows that the FALCON's consensus latency was faster than four popular PAXOS implimentations by 11.1x to 22.2x. When increasing the replica group size from three to 105, FALCON's consensus latency increases merely from 8.8 μs to 31.1 μs . FALCON is faster than a recent RDMA-based PAXOS protocol by up to 4.4x. It was able to support 9 widely used, unmodified server programs (e.g., Redis and `MySQL`) with low overhead. All FALCON source code, benchmarks, and raw evaluation results are available at `github.com/nsdi17-p52/falcon`.

1 Introduction

Consensus protocols (typically, PAXOS [63, 57, 56, 81]) plays a core role in datacenters and distributed systems, including ordering services [65, 50, 41], leader election [5], lock services [22], and fault-tolerance [51, 43, 29]. A PAXOS service runs the same program on a group of replicas and can enforce a strongly consistent, total order of inputs for this program as long as a quorum (typically, majority) of replicas still work normally.

Due to PAXOS's strong consistency and fault-tolerance, PAXOS is widely served in numerous systems. For instance, a DHT system Scatter [41] partitions distinct key ranges to PAXOS groups and runs 8 to 12 replicas in each group to agree on input requests for each key range. To improve throughput, Scatter further partitions the per-group key range for the replicas in each group so that each replica can server requests in parallel. The more replicas are in each group, the higher throughput Scatter can achieve. Moreover, recent state machine replication (SMR) systems [29, 51, 43] use PAXOS to greatly improve the availability of general server programs.

Unfortunately, despite these great advances, the high consensus latency of traditional PAXOS protools still makes many systems suffer. For efficiency, PAXOS typically assigns one replica work as the leader which invokes consensus requests, and the other replicas work as backups to agree on or reject these requests. To agree on an input, at least one message round-trip is required between the leader and a non-leader. Given that a `ping` in Ethernet takes hundreds of μs, a server program running in an SMR system with three replicas must wait at least this time before processing an input. This latency could be acceptable for infrequent leader elections [22, 5] or heavyweight transactions [29, 51], but prohibitive for key-value stores. To address this challenge, some systems [80, 43] batch requests into one consensus round. However, batching will only mitigate throughput lost and it will aggravate request latency.

As the replica group size grows, the consensus latency of traditional PAXOS protocols increases drastically because now a majority involves more replicas. To improve the scalability of consensus latency, one approach is invoking consensus in parallel. For instance, SPaxos and Zookeeper use multithreading [**?**, 5], and Crane [29] and libPaxos [76] use asynchronous IO (Libevent [59]). However, the high latency of a each round-trip still exists, and the synchronizations in these mechnisms will frequently invoke OS events such as context switches (each may take sub milli seconds). 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 33.3% to 77.7%, and 44.4% to 88.8% of this increase was spent in OS kernels and networking libraries.

Another approach to scale PAXOS is maintaining multiple instances of PAXOS and exploit parallilism among instances. Such approach includes partitioning a program and its data [41, **?**, **?**], splitting consensus loads [62, **?**], and hieratical replication [50, 41]. However, the core building block in these systems, PAXOS itself, still scales

poorly [65, 41].

Fortunately, as Remote Direct Access Memory (RDMA) becomes increasing commonplace, it becomes a possible solution to tackle the PAXOS's consensus latency, because it not only provides the option to bypass the OS kernel, but also provides dedicated, efficient TCP/IP hardware mechanisms. For instance, the fastest RDMA operation allows a process to directly write to the user space memory of a remote replica's process, completely bypassing the remote OS kernel or CPU (the so called "one-sided" operations). As a common RDMA practice, to ensure that such a write successfully resides in the memory of a remote process, the local process should wait until the remote NIC (network interface card) sends an ACK to the local host's NIC. Such a write round-trip takes only $\sim$3 μs in an evaluation [66].

However, due to the unrichness of RDMA primitives, it's technically challenging to build a PAXOS runtime system that fully exploits RDMA speed. For instance, one-sided RDMA operations eliminate remote replicas' participations, but traditional PAXOS protocols require non-leader replicas to examine the leader's consensus requests. To overcome this issue, DARE [75], a recent RDMA-based PAXOS protocol, proposes a sole-leader, two-round protocol. First, the leader uses RDMA to write the consensus requests to all replicas and polls RDMA ACKs to check whether the writes succeed. Second, for the successful writes, the leader does another round of RDMA writes to mark the writes as successful on other replicas, and poll ACKs on these writes. One a majority of successful writes in the second round, DARE reaches a consensus. Our evaluation shows that both the polling of RDMA ACKs and the two-round consensus incurred approximately linearly consensus latency: DARE's consensus latency increased by 21.12x as replica group size increased by 35x (§**??**).

Our key idea is that we should make all PAXOS replicas receive consensus messages purely on their local memory. By doing so, both the leader and non-leaders can receive consensus messages purely on their local memory, bypassing various inbound scalability bottlenecks, including RDMA ACKs and RDMA queue accesses. An analogy is that threads receive other threads' data and signals via bare memory, a fast and scalable multithreading pattern. Now the only RDMA primitive our PAXOS replicas involve is just sending RDMA writes (i.e., copying the data to be sent to NIC). Our evaluation showed that most of such outbound RDMA write operations took less than 0.2 μs, much faster faster than inbound RDMA operations.

In deed, this idea appears to pose reliability issues because now the leader lacks evidence on whether the remote RDMA writes succeed. Fortunately, the PAXOS protocol already tolerates various reliability issues, including message losses caused by hardware or software failures. A scalable RDMA-based PAXOS runtime system now just needs to carefully ensure the atomicity and integrity of RDMA writes among replicas (§4.1).

We have adopted this idea in FALCON,[1] a fast, scalable PAXOS protocol and its runtime system. FALCON supports general programs: within FALCON, a program just runs as is, and FALCON automatically deploys this program on replicas of machines. It intercepts inputs from a server program's inbound socket calls (e.g., `recv()`) and invokes our PAXOS protocol to efficiently enforce same inputs across replicas.

To practically improve the assurance that replicas run in sync, on top of FALCON's PAXOS protocol, we also build an efficient network output checking protocol that efficiently compares output across replicas. first computing an accumulated hash by intercepting a server program's outbound socket calls (e.g., `send()`), it then occasionally invokes an consensus to compare these hashes among replicas. This output checking protocol is just a practical feature that could improve assurance on keeping replicas in sync and promote FALCON's deployments.

We implemented FALCON in Linux. FALCON intercepts POSIX inbound socket calls (e.g., `accept()` and `recv()`) to coordinate inputs using the Infiniband RDMA architecture. FALCON intercepts POSIX outbound socket operations (e.g., `send()`) to invoke the output checking protocol. This simple, deployable interface design makes FALCON support general server programs without modifying them. To recover or add new replicas, FALCON leverages CRIU [28] to perform checkpoint/restore for general server programs on one non-leader replica, introducing little performance impact in normal case.

We compared FALCON with five popular, open source PAXOS-like implementations, including four traditional ones (libPaxos [76], ZooKeeper [5], CRANE [29] and SPaxos [**?**]) and a RDMA-based one (DARE [75]). S-Paxos is designed to achieve scalable throughput when more replicas are added. We also evaluated FALCON on nine widely used or studied server programs, including four key-value stores (Redis [77], `Memcached` [64], `SSDB` [79], and `MongoDB` [67]), one SQL server `MySQL` [12], one anti-virus server `ClamAV` [26], one multimedia storage server `MediaTomb` [11], one LDAP server OpenLDAP [73], and one advanced transactional database `Calvin` [80] (with ZooKeeper [5] as its SMR protocol). Our evaluation shows that

1. FALCON achieves both one order of magnitude better scalability and one order of magnitude faster consensus latency than literature. Figure **??** shows a sum-

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

mary. FALCON's consensus latency was faster than four popular PAXOS implimentations by 11.1x to 22.2x on three to nine replicas. FALCON is faster than DARE by 65.1% to 4.4x. When increasing the replica group size from three to 105 (a 35x increase), FALCON's consensus latency increases merely from 8.8 μs to 31.1 μs (a 3.89x, sub-linear increase).

2. FALCON is general. For all nine evaluated programs, FALCON ran them without any modification except `Calvin` (we added a 23-line patch to make `Calvin`'s client and server communicate with sockets).
3. FALCON incurs low overhead on nine widely used server programs. With nine 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 PAXOS leader failures, FALCON's leader election latency was reasonable and scalable.

Our major contribution is the idea of pure remote-memory consensus. This simple yet effective idea leads to FALCON, a fast, scalable PAXOS runtime system. FALCON has the potential to largely improve the scale and speed of existing PAXOS services. For instance, previously Scatter deployed 8 to 12 replicas in each PAXOS group [41], now it can deploy one order of magnitude more replicas in each group with much faster consensus latency. Moreover, a general and deployable service, FALCON may largely promote the deployments of PAXOS and provide strong fault-tolerance and consistency to various systems.

The remaining of this paper is organized as follows. §2 introduces background on PAXOS and RDMA features. §3 gives an overview of our FALCON system. §4 presents FALCON's consensus protocol and its runtime system. §5 describes the output checking protocol. §6 compares DARE with FALCON, and discusses FALCON's current limitations and applications in other areas. §7 presents evaluation results, §8 discusses related work, and §9 concludes.

2 Background

This section introduces the background of two key techniques in FALCON, the PAXOS consensus protocol (§2.1) and RDMA features (§2.2).

2.1 PAXOS

PAXOS [81, 57, 56, 24, 58, 63] 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 [76, **?**, **?**, 43, 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 33.3% to 77.7%, and 44.4% to 88.8% 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 commonplace in datacenters due to its extreme 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 [49, 66, 75]; 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 [66]. 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 [49] 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 for an fixed-sized 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 [75], and we found that, although this protocol 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 to 79 μs; when the CQ has one or more ACKs randomly arrived from other replicas, it took 260 to 410 μs. As the number of ACKs is propotional to the replica group size, polling ACKs become a major scalability bottleneck (§7).

Fortunately, depending on the application logic, we can do *selective signaling* [49]: it only checks for an ACK after pushing a number of WRITEs (previous WRITEs may already succeed before this ACK-checking starts). To achieve scalability, FALCON's protocol uses selective signaling and eliminates ACK pollings on every consensus round.

3 FALCON Overview

FALCON runs replicas of a server program in a datacenter. Replicas connect with each other using RDMA QPs. Client programs located in LAN or WAN networks. The leader handles client requests and runs our RDMA-accelerated PAXOS protocol to coordinate inputs among replicas.

Figure 1 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

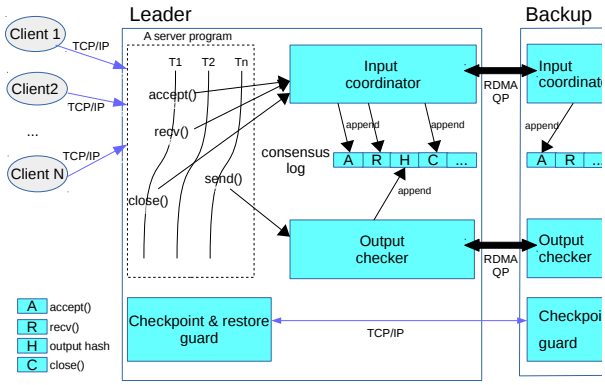

Figure 1: *The* FALCON *Architecture.* FALCON components are shaded (and in blue).

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.

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.

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

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 Input Consensus Protocol

This section first presents FALCON's RDMA-accelerated PAXOS consensus protocol, including the fast,scalable algorithm in normal case (§4.1), handling concurrent connections (§4.2), and the leader election protocol (§4.3). It then analyzes the reliability of our protocol (§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 2 shows the format of a log entry in FALCON's consensus log. Most fields are regular as those in a typical PAXOS protocol [63] 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 3 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 [75] 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 2).

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 [35, 49] leverages the left-to-right ordering of RDMA WRITEs

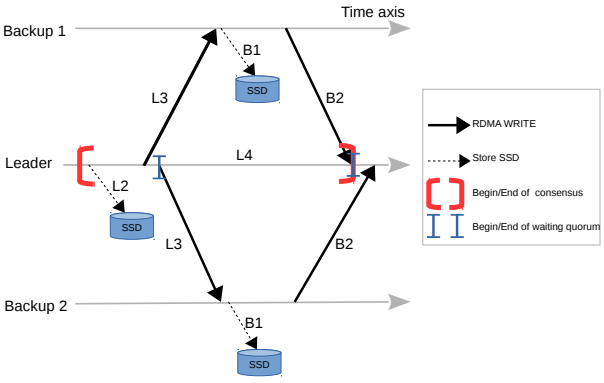

Figure 3: FALCON *consensus algorithm in normal case.*

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 [83] 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 [63]. 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 3. 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 2).

Third (**B3**, not shown in Figure 3), the follower does a regular PAXOS check on `last_committed` and executes all socket calls that it has not executed before this viewstamp. It then "executes" each log entry by forwarding the socket calls to the local server program. This forwarding faithfully builds and closes concurrent connections between the follower and the local server program according to the socket calls in the consensus log.

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 has already made viewstamps [63] 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 [75] 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 [63] 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 [56] 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

To minimize protocol-level bugs, FALCON's PAXOS protocol mostly sticks with a popular, practical implementation [63], especially the behaviors of senders and receivers (§4.1 and §4.3). For instance, both FALCON's normal case algorithm and the popular implementation [63] 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 [72, 56, 57, 81] or implement [24, 63] verify [86, 42]. Aligning with a practical PAXOS implementation [63] 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

This section presents FALCON's output checking protocol for detecting and recovering from replicas' execution divergence. This section first introduces how FALCON computes and compares network outputs among replicas (§5.1), and then introduces its checkpoint and rollback mechanism to deal with divergence (§5.2).

5.1 Checking Network Outputs

One main issue is that network outputs and their physical timings are pretty 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 overcome this timing issue, 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, after every T_{comp} (by default, 1000 in FALCON) local hash values are generated, FALCON invokes an output checking protocol to check this hash across replicas. The index of this hash in the generated sequence is consistent across replicas because each replica runs the same mechanism to generate the hash sequence.

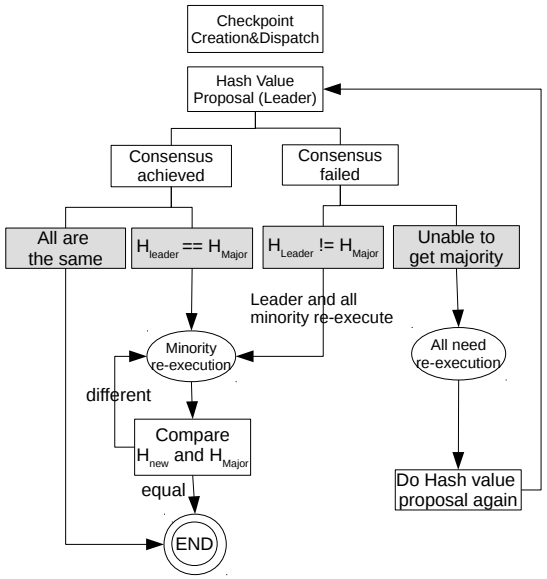

Figure 4: ***Workflow to Handle Network Output Divergence.***

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. To make an effort to collect more remote hash values, FALCON waits for $T_{waitack}$ (by default, 20 μs) and then starts to detect divergence on hash values.

Figure 4 shows the workflow on how the leader checks present replies and handles divergence, which includes four shaded cases: (1) all hashes are the same; (2) leader's hash equals a majority of replicas, but minor replicas' hashes diverge; (3) leader's hash diverges from a majority of replicas'; and (4) no majority has the same hash value. The first three cases should be the normal case unless a program tends to frequently compute outputs on random functions (e.g., a scientific simulator).

Currently FALCON has not incorporated an input replaning approach (e.g., Eve [51] sequentially re-executes requests after detecting a execution divergence) because we found that re-execution already empirically avoided divergence. Previous work [61, 74] also confirmed that it was extremely unlikely to trigger a concurrency bug twice in two consecutive executions.

5.2 Checkpoint and Restore Implementations

A guard process is running on each replica to checkpoint and restore the local server program. The guard does two tasks. First, FALCON 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 physical duration T_{ckpt} (one hour in FALCON).

Such a checkpoint operation and its duration are not sensitive to FALCON's 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.

The second task for guards is that all guards in all alive replicas handle rollback requests once divergence is detected (§5.1). According to the rollback workflow, a backup guard which receives a rollback request from the leader guard will kill the local server process and roll back to a previous checkpoint before the last successful hash check.

6 Discussions

This section discusses FALCON's limitations (§6.1) and its applications in other research areas (§6.2).

6.1 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 [51, 63] in PAXOS protocols can also be leveraged to intercept these functions and make them produce same results among replicas.

FALCON's output checking protocol may have false positives or false negatives, because it is just designed to try to improve assurance on whether replicas run in sync. A server program running in FALCON may have false positive when it uses multiple threads to serve the same client connection and uses these threads to concurrently add outputs (e.g., `ClamAV` in our evaluation). Running a deterministic multithreading scheduler [15, 31] with the server program addresses this problem.

A server program may also have false negative when it triggers a software bug but the bug does not propagate to network outputs. From client programs' point of view, such bugs do not matter. Moreover, FALCON already checkpoints file system state to mitigate this issue.

Currently FALCON has not incorporated read-only optimization [51] because its performance overhead compared to the evaluated programs' unreplicated executions is already reasonable (§7.2). 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.

6.2 FALCON Has Broad Applications

In addition to greatly improving the availability of general server programs, we envision that FALCON can be applied in broad areas, and here we elaborate three. First, FALCON's RDMA-accelerated PAXOS protocol and its implementation could be a template for other replication protocols (e.g., byzantine fault-tolerance [53, 23]).

Second, by efficiently constructing multiple, equivalent executions for the same program, FALCON can benefit distributed program analysis techniques. Bounded by the limited computing resources on single machine, recent advanced program analysis frameworks become distributed [25, 70, 47, 84, 30] in order to offload analyses on multiple machines. FALCON can be leveraged in these frameworks so that developers of analysis tools can just focus on their own analysis logic, while FALCON's general replication architecture handles the rest.

Moreover, program analyses developers can tightly integrate their tools with FALCON. For instance, they can proactively diversify the orders of socket calls in FALCON's consensus logs among replicas to improve replicas' tolerance on security attacks [87].

Third, FALCON can be a core building block in the emerging datacenter operating systems [88, 45, 4]. As a datacenter continuously emerges a computer, an OS may be increasingly needed for such a giant computer.

FALCON's fast, general coordination service is especially suitable for such an OS's scheduler to maintain a consistent, reliable view on both computing resources and data in a datacenter. For instance, FALCON's latency is largely between less than 30 μs, much smaller than a typical process context switch (a few hundreds μs).

7 Evaluation

Our evaluation used three Dell R430 servers as SMR replicas. Each server has Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. Each machine has a Mellanox ConnectX-3 Pro Dual Port 40 Gbps NIC. These NICs are connected using the Infiniband RDMA architecture. The `ping` latency between every two replicas are 84 μs (the IPoIB round-trip latency).

To mitigate network latency of public network, all client benchmarks were ran in a Dell R320 server (the client machine), with Linux 3.16.0, 2.2GHz Intel Xeon 12 hyper-threading cores, 32GB memory, and 160GB SSD. This server connects with the server machines with 1Gbps bandwidth LAN. The average `ping` latency between the client machine and a server machine is 301 μs. A larger network latency (e.g., sending client requests from WAN) will further mask FALCON's overhead.

We evaluated FALCON on nine widely used or studied server programs, including four key-value stores Redis, `Memcached`, `SSDB`, `MongoDB`; `MySQL`, a SQL server; `ClamAV`, a 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 that leverages ZooKeeper as its SMR service. 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 fault-tolerance of SMR is especially attractive to these programs.

Table 1 introduces the benchmarks and workloads we used. To evaluate FALCON's practicality, we used the server developers' own performance benchmarks or popular third-party. 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).

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.

We spawned up to 32 concurrent connections, and then we measured both response time and throughput. We also measured FALCON's bare consensus latency. All evaluation results were done with a replica group size of three except the scalability evaluation (§7.4). Each performance data point in the evaluation is taken from the mean value of 10 repeated executions.

The rest of this section focuses on these questions:

§7.1: Is it easy to run general server programs in FALCON?

§7.2: What is FALCON's performance compared to the unreplicated executions? What is the consensus latency of FALCON's PAXOS protocol?

§7.3: What is FALCON's performance compared to existing SMR systems?

§7.4: How scalable is FALCON on different replica group sizes?

§7.5: How fast can FALCON checkpoint and recover replicas when output divergence is detected?

7.1 Ease of Use

FALCON is able to run all nine 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`.

7.2 Performance Overhead

Figure 5 shows FALCON's throughput and Figure 6 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 5, 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

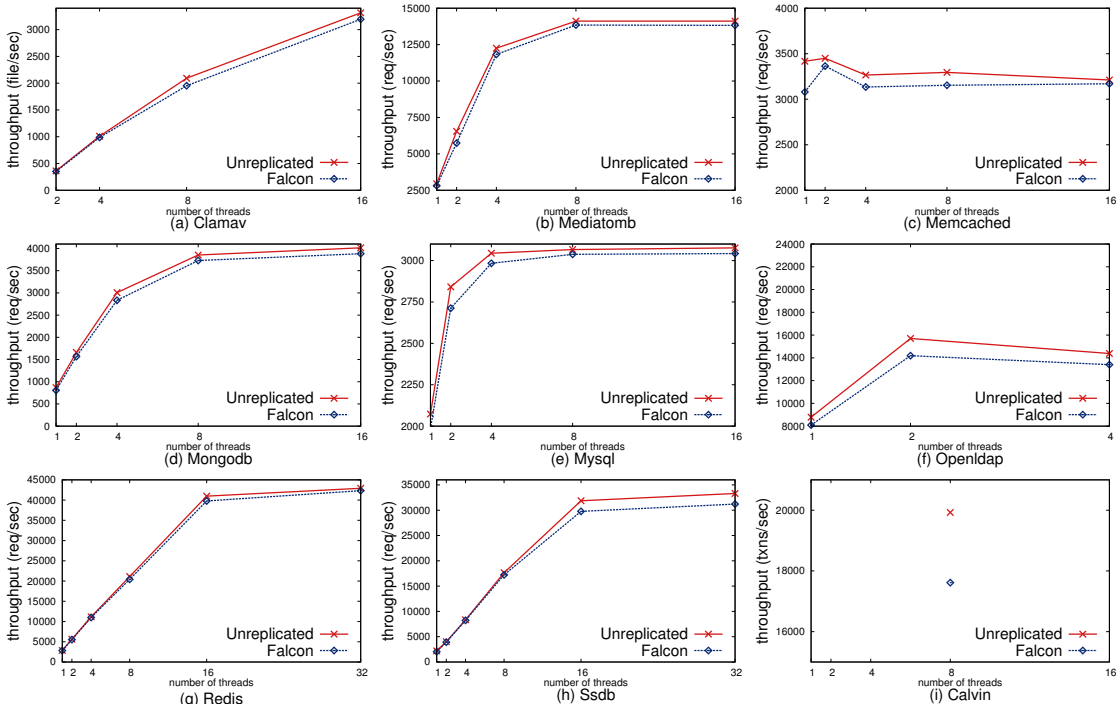

Figure 5: FALCON *throughput compared to the unreplicated execution.*

except `Memcached`. A prior evaluation [43] 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 (§5.1).

Program	# Calls	Input	SSD time	Quorum time
`ClamAV`	30,000	42.0	4.9 μs	7.2 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	6.4 μs	12.0 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 2: *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 2 shows these statistics per 10K requests, 8 or max (if less than 8) threads. According to the consensus algorithm steps in Figure 3, for each socket call, FALCON's leader does an "L2": SSD write (the "SSD time" column in Table 2) 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 summing these two time columns, overall, a FALCON input consensus took only 9.9 μs (Redis) to 39.6 μs (`MongoDB`). 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 5 and response times Figure 6.

This small latency suggests that, even if clients are deployed within the same datacenter network, FALCON may still achieve acceptable overhead on many programs (although FALCON's deployment model is running server replicas in a datacenter and clients in LAN or WAN).

7.3 Comparison with Other SMR systems

We compared FALCON with `Calvin`'s SMR system because `Calvin`'s input consensus uses ZooKeeper, one of the most widely used coordination service built on

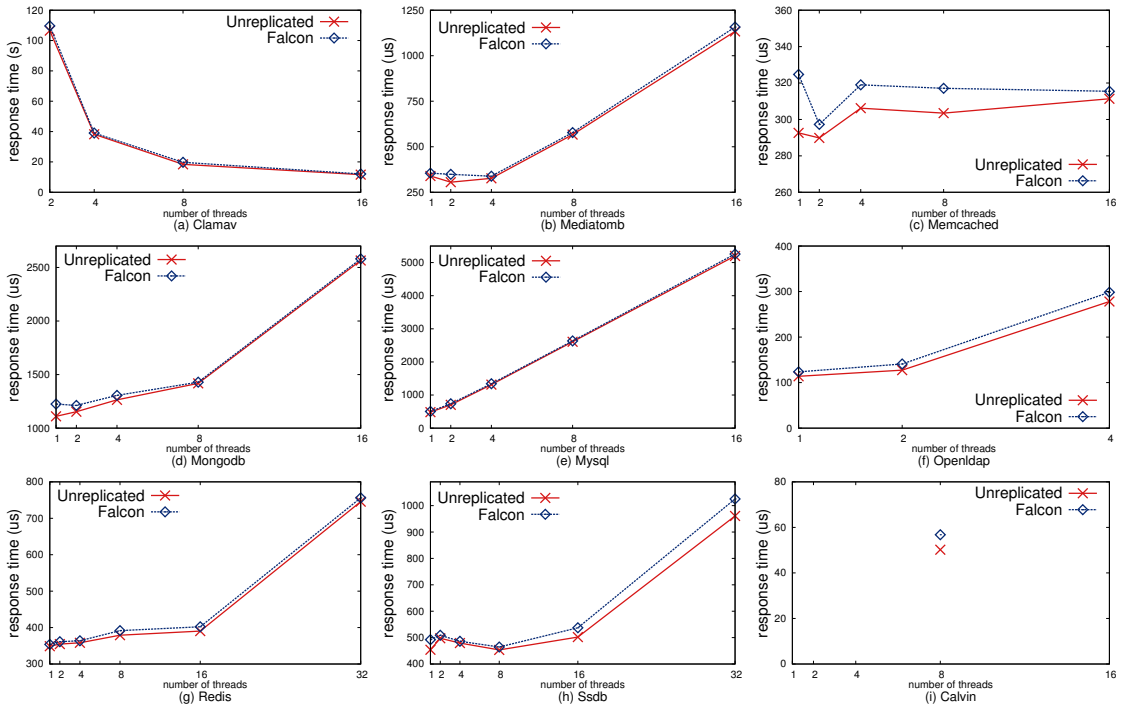

Figure 6: FALCON *response time compared to the unreplicated execution.*

TCP/IP. To conduct a fair comparison, we ran `Calvin`'s own transactional database server in FALCON as the server program, and we compared throughputs and the consensus latency with `Calvin`'s consensus protocol ZooKeeper.

As shown in Table 3, `Calvin`'s ZooKeeper replication achieved 19.9K transactions/s with a 511.9 μs consensus latency. FALCON achieved 17.6K transactions/s with a 12.5 μs consensus latency. The throughput in `Calvin` was 13.1% higher than that in FALCON because `Calvin` puts transactions in a batch with a 10 ms timeout, it then invokes ZooKeeper for consensus on this batch. The average number of bytes in `Calvin`'s batches is 18.8KB, and the average number of input bytes in each FALCON consensus (one for each `read()` call) is 93 bytes. Batching helps `Calvin` achieve good throughput. FALCON currently has not incorporated a batching technique because its latency is already reasonable (§7.2).

Notably, FALCON's consensus latency was 40.1X faster than ZooKeeper's mainly due to FALCON's RDMA-accelerated consensus protocol, although we ran `Calvin`'s ZooKeeper consensus on IPoIB. A prior SMR evaluation [75] also reports a similar 320 μs consensus latency in ZooKeeper. Two other recent SMR systems Crane [29] and Rex [43] may incur similar consensus latency as ZooKeeper's because all their consensus protocols are based on TCP/IP. Overall, this FALCON-`Calvin` comparison suggests that `Calvin` will be better if clients prefer high throughput, and FALCON will be better if clients demand short latency.

Performance metric	ZooKeeper	FALCON
Throughput (requests/s)	19,925	17,614
Consensus latency (μs)	511.9	12.5

Table 3: *Comparison with `Calvin`'s ZooKeeper replication.*

7.4 Scalability on Replica Group Size

Because we had only three RDMA machines in the evaluation time, we picked Redis as the testing program and ran five to seven FALCON instances on three machines: one machine held the leader instance, and each of the other two machines held two to three backups.

Table 4 shows the Scalability results. Given the same workload, FALCON's five-replica setting had a 17.9K requests/s and 10.1 μs consensus latency, and its seven-replica setting had a 17.4K requests/s and 11.0 μs consensus latency. Compared to the three-replica setting, FALCON achieved a similar consensus latency from three to seven replicas, because a FALCON consensus latency mainly contains two SSD stores and one WRITE round-trip (see Table 2).

We didn't consider these initial scalability results general; we found them promising. Given that each RDMA NIC hardware port supports up to 16 outbound RDMA WRITEs with peak performance [49], and our NIC has dual ports, we anticipate that our algorithm may scale up to around 32 replicas under current hardware techniques. We plan to buy more servers and verify whether FALCON's scalability is bounded by RDMA NIC capacity.

7.5 Checkpoint and Recovery

FALCON's output checking protocol found different output results on `ClamAV`, OpenLDAP, and `MediaTomb`. The output divergence of `ClamAV` was caused by its threading model: `ClamAV` uses multiple threads to serve a directory scan request, where the threads scan files in parallel and append the scanned files into a shared output buffer protected by a mutex lock. Therefore, sometimes `ClamAV`'s output will diverge across replicas. Running a deterministic multithreading runtime with `ClamAV` avoided this divergence [29]. OpenLDAP and `MediaTomb` contained physical timestamps in their replies. `Calvin` did not send outputs to its client.

Each FALCON periodic checkpoint operation cost 0.12s to 11.6s on the evaluated server programs. A checkpoint duration depends on whether a server's process has updated much of its memory or modified files in its current working directory. `ClamAV` incurred the largest checkpoint time (11.6s) because it loaded and scanned files in a `/lib` directory. To evaluate the efficiency of FALCON's rocovery, we ran `ClamAV` and tried to randomly trigger its output divergence. `ClamAV`'s rocovery time varied from 1.47s to 1.49s for backups and the leader, including extracting files in its local directory, restoring process state with CRIU, and reconnecting RDMA QPs with remote replicas.

Performance metric	Three	Five	Seven
Throughput (requests/s)	18,054	17,890	17,410
Consensus latency (μs)	9.9	10.9	11.0

Table 4: FALCON's *scalability on replica group size.*

8 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 [63, 57, 56, 81, 69] and implementations [24, 63, 22]. PAXOS is notoriously difficult to be fast and scalable [65]. To improve speed and scalability, various advanced replication models have been developed [69, 62, 41, 50]. Since consensus protocols play a core role in datacenters [88, 45, 4] and distributed systems [27, 62], a variety of study have been conducted to improve different aspects of consensus protocols, including performance [69, 58, 75], understandability [72, 57], and verifiable reliability rules [86, 42]. Although FALCON tightly integrates RDMA features in PAXOS, its implementation mostly complies with a popular, practical approach [63] 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 [51], Rex [43], Calvin [80], and Crane [29]. 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 (§7.3).

Notably, Eve [51] 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.

The second category includes DARE [75], a coordination protocol that also uses RDMA to reduce latency. Part of FALCON's implementation was inspired by DARE. FALCON differs from DARE in two major aspects.

The first difference is in a reliability model level. DARE's model is different from standard PAXOS's: DARE assumes that a replica's memory is still accessible to remote replicas even if this replica's CPU fails, so that DARE's leader can still write to remote backups. With this reliability model, DARE requires four one-sided RDMA writes in each consensus round between the leader and a backup. DARE's paper shows that the MTTF (mean time to failure) of memory and CPU are similar.

FALCON's reliability model complies with standard PAXOS's: memory and CPU may fail, thus consensus requests must be written to stable storage. FALCON requires two one-sided RDMA writes and two SSD writes on a consensus round between the leader and a backup. DARE reported a $\sim$15 μs consensus latency for write requests and $\sim$7 μs for read requests on a 64-byte payload. FALCON's consensus latency is compatible with DARE's on a similar payload size (e.g., FALCON's consensus latency for `SSDB` was 14.6 μs in Table 2).

The second difference is in an application level. DARE's evaluation used a 335-line, single-threaded key-value store. FALCON aims to support general programs.

RDMA 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 [39], key-value stores [66, 49, 35, 48], transactional processing systems [83, 36], and file systems [85]. 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 [54, 34, 15, 33, 71, 60, 32, 31, 14] is pervasive in both application programs and OS kernels, and it often comes with concurrency bugs [61]. To mitigate nondeterminism, deterministic multithreading techniques [18, 60, 14, 21, 34, 71, 15, 17, 46, 16] and deterministic replay techniques [44, 40, 78, 37, 52, 82, 38, 74, 55, 13, 68] 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.

9 Conclusion

We have presented FALCON, a fast, general SMR system through building an RDMA-accelerated PAXOS protocol to efficiently coordinate inputs. On top of this protocol, FALCON introduces an automated, efficient output checking protocol to detect and recover out divergence. Evaluation on widely used server programs shows that FALCON is fast, scalable, and robust. FALCON has the potential to promote the adoption of SMR and to improve the reliability of general programs. All FALCON source code, benchmarks, and 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] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, Oct. 1999.

[24] 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.

[25] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, June 2008.

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

[27] 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.

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

[29] 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.

[30] H. Cui, R. Gu, C. Liu, and J. Yang. Repframe: An efficient and transparent framework for dynamic program analysis. In *Proceedings of 6th Asia-Pacific Workshop on Systems (APSys '15)*, July 2015.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2013.

[48] 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.

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

[50] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, 2010.

[51] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[52] 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.

[53] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Oct. 2007.

[54] 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.

[55] 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.

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

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

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

[59] libevent. `libevent.org/`, 2015.

[60] 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.

[61] 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.

[62] 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.

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

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

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

[66] 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.

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

[68] 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.

[69] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[70] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Thirteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 308–318, 2008.

[71] 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.

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

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

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

[75] 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.

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

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

[78] 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.

[79] `ssdb.io/`.

[80] 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.

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

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

[83] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, SOSP '15, Oct. 2015.

[84] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing data race detection. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 27–38, Mar. 2013.

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

[86] 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.

[87] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan. Concurrency attacks. In *the Fourth USENIX Workshop on Hot Topics in Parallelism (HOTPAR '12)*, June 2012.

[88] 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.