Network-Assisted Raft Consensus Algorithm

Yang Zhang† Bo Han* Zhi-Li Zhang† Vijay Gopalakrishnan*

†*University of Minnesota* **AT&T Labs – Research*

ABSTRACT

Consensus is a fundamental problem in distributed computing. In this poster, we ask the following question: can we *partially* offload the execution of a consensus algorithm to the network to improve its performance? We argue for an affirmative answer by proposing a network-assisted implementation of the Raft consensus algorithm. Our approach reduces consensus latency, is failure-aware, and does not sacrifice correctness or scalability. In order to enable Raft-aware forwarding and quick response, we use P4-based programmable switches and offload partial Raft functionality to the switch. We demonstrate the efficacy of our approach and performance improvements it offers via a prototype implementation.

1. INTRODUCTION

Distributed systems often require participants to agree on some data value that is needed during computation. Consensus algorithms (*e.g.,* Paxos [9], ZAB [7] and Raft [10]) facilitate the participants to reach consensus, even in the face of failures. These consensus mechanisms tend to incur high overheads in terms of latency since they involve multiple rounds of communication. This is especially true when strong consistency guarantees are desired. Even without failure, consensus requires at least the round-trip time between servers running consensus algorithms. Thus, offloading application-level implementation of a consensus algorithm to the network offers the potential to reduce the consensus latency.

Several recent projects investigate the offloading of consensus algorithms to the network. NetPaxos [6] proposes implementing Paxos in the network by utilizing OpenFlow switches. NetPaxos can also be implemented using P4 [5], a domain specific language that allows the programming of packet-forwarding data plane [4]. István *et al.* [8] take the efforts of implementing the entire ZAB consensus algorithm [7] on FPGA devices using a low-level language. This hardware-based solution, however, may not be scalable as it requires the storage of potentially large amounts of consensus states, logic, and even the application data. In contrast, we propose a network-assisted Raft consensus algorithm [10] that takes advantage of programmable P4 [4] switches and offloads certain Raft functionality to the network. We focus on Raft since it has formally proven to be safe and is more understandable than Paxos [10]. More importantly, Raft has been used in the implementation of popular SDN controllers such as OpenDayLight [1].

Our goal in this poster is to improve the performance of Raft without sacrificing correctness and scalability. Raft has 3 roles: a leader who maintains consensus in a centralized way, a follower who passively responds to Raft remote procedure calls (RPCs), and a candidate who is converted from a follower during leader election when the original leader fails. The basic version of Raft has only 2 RPCs: RequestVote issued by a candidate during election and AppendEntries issued by the leader to send heartbeats or log entries. Thus, it has only 4 message types (2 RPCs and the corresponding responses) compared to 10 types in ZAB. In our scheme, we offload the processing of AppendEntries message to the programmable switches.

The unique feature of our proposed solution is that we duplicate only the necessary logic to switches which act as a cache to reduce consensus latency. Thus, we minimize the storage of replicated log entries and state machine in switches. Note that, the entire Raft algorithm is still running on server machines. This partial offloading architecture helps improve the performance of Raft, especially the consensus latency, without sacrificing scalability.

2. SYSTEM DESIGN

2.1 Overview

There are three key requirements for our network-assisted Raft. First, we should guarantee the correctness of Raft algorithm when offloading its processing logic to the network. Second, the Raft logic on a switch should be able to respond to most requests directly for improved performance. Third, the Raft logic on a switch should safely discard obsolete log entries and even state machine for scalability. In the basic Raft consensus algorithm, there are three major elements: *leader election*, *log replication*, and *log commitment*. In order to satisfy the above requirements, our system consists of two key components: a front-end implemented in a switch taking care of *log replication* and *log commitment*, and a back-end in a server running a complete Raft implementation[1]. The front-end enhances Raft in two aspects: first, it is able to perform Raft-aware forwarding; second, it can quickly respond to Raft requests by rewriting the incoming packets. The job of back-end is to maintain complete states on the server for responding to certain requests (described in the next section) which might not be fulfilled by the front-end.

2.2 Raft-aware P4 Switch

The front-end runs in a P4 switch. The switch parses Raft requests and caches Raft states using P4's primitive actions. Upon receiving a request, the front-end parses it and rewrites the message to construct the corresponding response. It also forwards the original packet to the back-end for liveness check; this is part of leader election. As usual, the back-end sends a response to the switch, but the front-end does not forward the message and only extracts necessary flow control information from it. For certain requests,

[1] *Leader election* and *log replication* are duplicated at front-end and back-end to improve performance and scalability.

the front-end may not be able to generate a response due to the limited information available on the switch; such a request will be served normally by the back-end. For example, when a new server joins the cluster, it needs to fetch all the logs. This may not be available at a front-end, and in such instances a back-end would serve the request.

Now we discuss how a front-end can forward certain Raft messages without involving its back-end. In Raft, requests from a client can only be handled by the leader. In the bootstrap phase, the client randomly picks a server in the cluster to communicate with. If the selected server is not a leader, it notifies the client of the leader's IP address (if known). The client then issues a new request to the leader. In our design, since a front-end is aware of Raft, the front-end of the selected server can forward the request to the leader directly and reduce the communication overhead.

In a practical system, especially when *log replication* and *log commitment* are implemented in switches, the log and state machine of Raft cannot grow unbounded due to limited memory space. We address this using an approach similar to (but simpler) the snapshot mechanism described in Raft [10]. In our scenario, a front-end discards obsolete information because its back-end always keeps the necessary information. However, the mechanism for discarding state machine is different from discarding obsolete log entries because the front-end needs to know whether a requested item is already in the switch's state machine or the server's state machine. Thus, before discarding a state machine cached in a switch, it needs to maintain a dictionary for existing keys in the system.

2.3 Handling Failures

We now discuss how we handle failures in this partial offloading architecture. Because a front-end records the gap between two consecutive Raft heartbeats, if the gap is larger than a timeout, it assumes that the back-end is down. The front-end will not discard any log and state machine and sync them after the failed back-end is recovered. Communication failure between the front-end and back-end is equivalent to a back-end failure.

If a front-end fails, more specifically, the switch running a front-end fails, its back-end times out and issues a RequestVote message which cannot reach others. The back-end then increases its term (defined as a representation of virtual time in Raft) indefinitely. Later on, when the switch is recovered, it will retrieve previous state by forwarding requests to the back-end and observing its response. At this point, the back-end may have the largest term number, but may not have up-to-date log. Thus, it would initiate a leader election, during which it can get the most recent log from the existing leader.

3. EVALUATION

In this section, we present preliminary experimental results to demonstrate the feasibility and efficiency of our proposed approach. We use the setup in Figure 1, which consists of four Docker containers and P4 switches (simulated by p4factory [3]) for our experiments. We run LogCabin [2] in three containers (leader in red and followers in green) and run three Raft-aware P4 switches (in blue), serving together as a Raft cluster. The box in grey acts as a client, and the P4 switch in yellow performs regular forwarding. We instrument LogCabin to measure the interval of RPC calls and

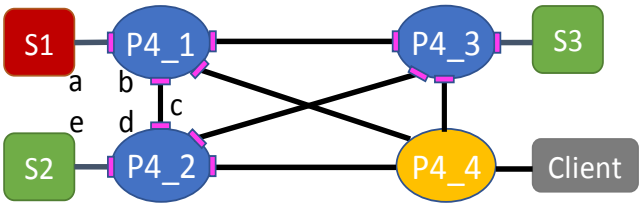

Figure 1: Experimental Setup

run tcpdump on the NICs in pink to record timestamps.

Table 1: Decomposed Latency between Raft Leader and a Follower. *a*: RPC latency at leader side + bidirectional latency between *S1* and *P4_1*; *b*: bidirectional latency in *P4_1*; *c*: bidirectional latency between *P4_1* and *P4_2*; *d*: bidirectional latency in *P4_2*; *e*: bidirectional latency between *P4_2* and *S2* + latency at Raft follower

	a	*b*	*c*	*d*	*e*	sum
Heartbeat (Vanilla)	106	233	6	231	94	670
Heartbeat (ours)	110	244	4	243	n/a	601
Write (Vanilla)	779	243	8	251	1206	2487
Write (ours)	n/a	479	7	451	n/a	937

We measure the latency (in µs) between a Raft leader and a follower for a heartbeat message and the client's write request, as shown in Table 1. The latency is decomposed into several fine-grained segments. We can observe latency savings for both heartbeat and write request. Moreover, we verify that the proof-of-concept does not add significant memory usage compared to the one performing just regular forwarding. Note that we are currently using a simulated P4 switch. We expect better performance when running the front-end on a real hardware P4 switch.

4. CONCLUSIONS & FUTURE WORK

By leveraging programmable devices, we propose to partially offload certain Raft functionality to a P4 switch for improving its latency, while not sacrificing scalability. We build a proof-of-concept using an open-source Raft implementation and a P4 switch simulator. We demonstrate the feasibility of our proposed solution via preliminary experiments, which show promising results. As our future work, we plan to formally validate the correctness of such a decoupled architecture and simplify the back-end implementation, since certain functions have already been implemented in the front-end. We also plan to compare our scheme implemented in real P4 switches with other existing FPGA-based solutions.

5. REFERENCES

[1] OpenDaylight Controller:MD-SAL. https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture:Clustering:2-Node.

[2] Logcabin: A distributed storage using raft. https://github.com/logcabin, 2016.

[3] P4 behavioral simulator. https://github.com/p4lang/p4factory, 2017.

[4] P. Bosshart et al. P4: Programming Protocol-Independent Packet Processors. In *CCR*, 2014.

[5] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM CCR*, 2016.

[6] H. T. Dang et al. Netpaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[7] P. Hunt et al. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. ATC*, 2010.

[8] Z. István et al. Consensus in a box: Inexpensive coordination in hardware. In *Proc. NSDI*, 2016.

[9] L. Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 1998.

[10] D. Ongaro et al. In Search of an Understandable Consensus Algorithm. In *Proc. ATC*, 2014.