LRab: Line-Rate Consensus

Ke Wang, Michael Wei

Abstract

Consensus is a fundamental building block for designing reliable, distributed systems. Distributed systems achieve consensus today with complex protocols which are implemented in software. While the flexibility of software simplifies programming consensus protocols, modern software is fundamentally asynchronous, which not only makes reasoning about failures difficult, but is also a bottleneck which limits the scalability of a distributed system. Hardware, on the other hand, offers much stronger timing guarantees but are notoriously difficult to program. Emerging technologies have helped bridge the gap between hardware and software, offering the performance of hardware with the flexibility and dynamism of software. This paper describes the design of LRab, which leverages both P4 and RDMA to provide consensus at line-rate, free of software bottlenecks.

1 Introduction

Distributed system developers must solve the problem of *consensus*, which entails getting multiple processes in a system to agree on some value, in order to build reliable distributed systems. Historically, consensus is provided by a number of consensus protocols implemented in software, with Paxos [10] being perhaps the most well known. These protocols must grapple with the fact that software is asynchronous: modern software runs on top of shared resources multiplexed by an operating system, which itself runs on a general purpose processor at the whim of hardware interrupts, speculative execution, the I/O and memory subsystem. This asynchrony not only makes it difficult to implement these protocols correctly as it is tricky to distinguish between a failed node and one that is merely slow, but also limits the rate consensus can be achieved, since slow nodes can hamper consensus progress.

Several emerging technologies now make it possible to program or dynamically control parts of a system which were previously controlled by fixed-function hardware. P4 [5] is a domain-specific language targeted at packet-forwarding data planes, such as switches, enabling programmers to dynamically change the behavior of switches. RDMA enables remote access to a host's memory without the involving the host operating system or interrupting the processor. These technologies however, are not a panacea: since they target hardware they are limited in their expressiveness. For example, P4 is devoid of control flow statements such as loops and has limited support for state. A consensus protocol which takes advantage of this new technology must be aware of these limitations.

In this paper, we present LRab, which provides consensus at line rate by leveraging P4 and RDMA so that parts of the protocol can be implemented in hardware, which not only removes bottlenecks associated with asynchronous software, but simplifies the design of the overall consensus protocol. While other systems may have implemented various consensus protocols in hardware, LRab is a complete system for providing consensus as a basic primitive for distributed application and addresses not only consensus, but failure handling and durability as well. LRab has the following properties:

- Fab provides the basic guarantees of consensus to clients: *reliable delivery* and a *total order* of proposals.
- LRab leverages the presence of a SDN to handle failure detection and reconfiguration of both software and hardware (such as switches) while ensuring the safety of the protocol.
- LRab exploits switching hardware through P4 to eliminate traditional bottlenecks in consensus protocols while remaining sensitive to hardware limitations.
- Proposals in LRab are made durable through RDMA to non-volatile memory, which provides consistent, predictable access times to storage.

In the following sections, we present the design of LRab and contrast LRab to traditional asynchronous software protocols. We then present a preliminary evaluation of a prototype of LRab in simulation, showing the scalability of LRab and the potential for LRab to drive consensus at line-rate once the appropriate hardware becomes available.

2 System

LRab consists of three basic components: a controller, switches and a set of replicas. The basic architecture of LRab is shown in figure 1.

The LRab *controller* is a software-defined network (SDN) controller. SDNs are common throughout the modern data center and separates the system which makes decisions about where data is sent (the controller) from the system that actually moves data to its destination (the data plane). LRab assumes the presence of a SDN controller which is capable of programming switches using P4. In LRab, we use the SDN controller to determine membership (which switches and replicas are participating in LRab), and to react to failures.

LRab *switches* are P4-programmable switches that connect replicas in LRab. The controller elects exactly one of these switches to be a *sequencer*, which is responsible for enforcing the ordering and safety requirements of LRab. LRab utilizes P4 for several reasons. First, P4 is much more expressive than OpenFlow, another popular language for programming switches. Second, several manufacturers have announced P4 capable hardware is on the horizon, and the P4 specification is open source, so P4 switches may soon become standard within the datacenter.

LRab *replicas* are servers running the *libLRab* software. Replicas make proposals, which are guaranteed to be reliably delivered and totally ordered by LRab. Replicas also act as the durable memory of LRab, and store the ordered proposals. LRab is quorum based, so with $2f+1$ replicas in the system, LRab can tolerate f failures. LRab uses RDMA to non-volatile memory located on each replica to provide LRab with consistent, predictable access times to storage.

In the next sections, we describe how the controller, switches, and replicas interact and the API that LRab presents to applications.

2.1 libLRab API

Each replica is composed of a client application and libFab as shown in Figure 1. Currently, libLRab is implemented in Python. The libLRab API provides two simple functions for client applications to interact with, which we describe below:

- *query()*: queries the recent consensus results since last call and returns a sequence of messages to be committed.

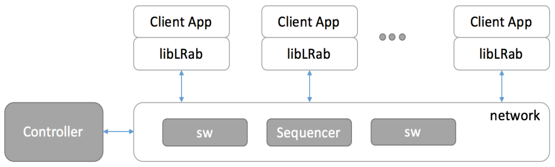

Figure 1: Architecture of LRab.

- *submit(message_id, group_id, msg)*: issues a message *msg* to a group identified by *group_id*. The function is blocking and returns success or failure. Success means a group of replicas agree to deliver the proposed message. The client application can decide what to do in case of failure, for example, it can re-submit the message, but must use the same *message_id* as the original message.

For each message, LRab guarantees the following properties:

- *Reliable delivery:* if a replica delivers message r, then eventually all non-faulty replicas will deliver message r.
- *Total order:* if a replica delivers message s prior to message t, then all non-faulty replica will deliver message s before message t.

2.2 libLRab roles

Each replica plays three roles at the same time:

- *message proposer*: Once libLRab receives *submit()* call from upper application it proposes the request to the group.
- *message acceptor*: Once libLRab receives proposed message from other replicas, it decides whether or not to accept it.
- *learn a result*: libLRab learns the result of a message: whether the message should be proposed or discarded. For agreed messages, libLRab stores them and returns it during the next *query()* call from upper application.

2.3 Switch sequencer

All replicas in LRab are connected by programmable switches. Initially, the first message from a replica will be redirected to the controller. The controller elects a single switch to act as the sequencer for this application. It installs flow rules to enforce all messages to go through one common switch. As in Figure 1, the selected switch

app:	group id
replica:	replica id within a group
slot_index:	index of the consensus slot
round:	round number of this slot
session:	current session number
msg_type:	type of the message

Table 1: Format of the LRab message header

acts as a sequencer and attaches a monotonically increasing sequence number to messages. Other switches in the network carry out regular routing. The controller choses an aggregate switch or a core switch to minimize the the penalty of enforcing all messages to go through the same switch. Other work [13] has shown this only adds 5 μs extra latency in a typical fat-tree topology. In addition, a core switch is less likely to fail compared to end hosts [7] such that a expensive fail-over process will happen rarely.

2.4 Consensus Protocol

We now describe the core consensus protocol of LRab. We first describe the messaging formats used and detail the message flows during normal operation and various failure scenarios.

2.4.1 Message Formats and Types

LRab is built on UDP. LRab adds a header before the application payload, as outlined in Table 1. LRab currently uses three message types, which we describe below:

- *Request*: Sent when LibFab replica proposes a message. *replica* is set to the ID of sender replica. *slot_index* is set to -1. At this time, replica does not know the index yet. *round* is set to 0 to indicate this is a new message.
- *Accept*: Sent when a LibFab replica accepts the message. *slot_index*, *round* and *session* of *Acc* message is set to be the same as in the Req message to indicate which message is being accepted.
- *Commit*: Sent when switch sequencer decides to commit a message. *slot_index* is the same as in the *Req* to indicate which message to be committed. *replica* is set to 0 to indicate this is a switch generated message.

2.4.2 Normal Operation of Fab

Figure 2 depicts the normal message flow of LRab where no replica nor sequencer fails, and all messages arrive in a timely manner. In this example, replica R2 proposes a message by sending a *Request* message. The message is directed to switch sequencer. The *replica_id* is set to R2.

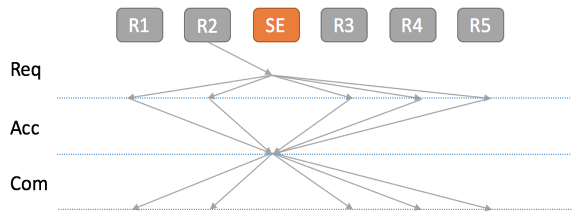

Figure 2: One example of normal operation. Replica R2 proposes a message to the group.

The sequencer switch maintains the current index of this group in a register. On receiving the *Request* message, it updates the index and writes it to the *slot_index* field of the packet. The message is now in slot *slot_index*. The switch then broadcasts this message to all acceptors.

When receiving a *Request* message, a replica accepts it unless it has already responded to to a *Prepare* message with a higher round number. The replica writes to disk before it sends back an *Accept* message to the switch. The switch maintains the current round number for each of the concurrent slots. After the switch collects accept messages with the same round from at least a quorum of acceptors for that *slot_index*, it broadcasts a *Commit* message to all replicas indicating the request can be committed.

On receiving a *Commit* message, a replica stores the message. To guarantee in-order delivery, the replica will only return the committed messages to the application if and only if all proposed messages before s have reached a consensus. The replica will return them together upon the next *query()* call.

2.4.3 Pipelining and flow control

We described how LRab reaches consensus for a single message, but LRab supports message pipelining for higher throughput. Multiple messages can be proposed at the same time, and the consensus protocol is run concurrently for each message. However, registers, which retain state in the switch are expensive and limited. For each concurrent slot in Fab, the switch needs to maintain a register to record the round and a register to count the number of accept messages in that round. This limits the total number of concurrent slots. To address this issue, LRab implements flow control at the switch sequencer. Each replica as well as the switch sequencer maintains two pointers: *current_index*, the highest index number seen by that participant (replica or switch) and *commit_index*, the slot number up to which all slots have reached a consensus.

Total number of concurrent slots is bound by

$$Max\_slots = current\_index - commit\_index \qquad (1)$$

Prepare:	Sent by the switch to all replicas when the switch times out for a consensus slot. It acts as a prepare message for retry. *slot_index* is the slot number whose consensus has failed, *round* is chosen to be a higher value.
Promise:	Sent by a replica in response to the *Prepare* message.
Expire:	Sent by a replica to switch when replica times out for slot.

Table 2: Additional message types

at the switch. The switch will discard any new requests if its *commit_index* exceeds *commit_index* by M. In such scenarios, switch sends a *Reject* message back to the replica. Index number from *commit_index+1* to *commit_index+M* is called the flow control window.

2.4.4 Replica recovery

Replica recovery is simplified as there is no special replica acting as the leader. If a replica crashes and recovers, it first reads from NVM the state of the system before its crash. This state includes all the messages to be delivered and the largest slot number it saw before crash. It then queries from a random neighbor replica for missing messages. Replica can accept new proposed messages and learns the result for them during its recovery.

2.4.5 Message loss

Request or accept messages lost can result in the failure of a consensus, since the switch sequencer may not collect enough accepts from a quorum of acceptors. In these situations LRab needs to retry until it eventually reaches the consensus. To this end, LRab introduces a round number for each slot. The initial proposal of a message is in round 0. When switch does not collect enough accept messages for that slot in round 0, it will time out. The switch will broadcast a *Prepare* message with a higher round number to all replicas.

Replicas reply with a *Promise* message containing the value it has accepted for this slot (if any) along with the round number in which it accepted. A round number -1 indicates the replica has never answered a *Request* message for this slot. This process is similar to the prepare phase in Paxos [10], but instead of letting the switch to collect *Promise* messages and propose based on the value in those *Promise* messages, the switch offloads this duty to replicas by relaying *Promise* messages to all replicas. A replica who collects enough *Promise* messages will re-propose for that slot based on the *Promise* messages it receives. *round* and *slot_index* of the re-proposed message remain the same as those in *Promise* message. A switch can distinguish a retried proposal from initial proposal by looking at the *slot_index*. *slot_index* is -1 for initial proposal and larger than 0 for retried proposal. Each time a switch retries it uses a higher *round* number. Multiple replicas may re-propose the message and switch discards every of those except for the first one. Redundant re-proposed messages can be detected as they have the same *round* numbe and *slot_index* number.

The Switch offloads the responsibility to replicas to collect *Promise* messages because it is expensive to store the payload of a packet in the switch. In standard Paxos, when a replica collects promise messages from a majority of acceptors, it should choose the accepted value with largest round number. This requires the replica to store and update the accepted value as it processes promise messages. Switching hardware, however, is designed for high speed processing or packets, not for efficient storage. Therefore switch only relays the *Promise* messages without processing them. *Commit* message loss will not prevent a slot from reaching a consensus. It will, however, cause certain replicas unaware of the result of a slot.

2.4.6 Time out

In general, it is difficult to include time-out mechanism in switch. The best a switch can do is to enable time out in a flow table [5][6]. Flow entries expires after a certain time and triggers an event [5]. In LRab, the switch relies on replicas to time out to trigger a different round for consensus. A replica maintains a timer for each slot. If the replica does not learn the result of that slot before timing out, it sends an *Expire* message to switch. When receiving an *Expire* message, switch first checks if the result for that slot is available. If so, switch replies with a regular *Commit* message to inform the replica the result. If not, switch will start a new round by broadcasting *Prepare* messages as detailed above.

A replica may never see a request of a slot if the *Request* message gets lost. Replica can detect the miss of a *Request* by observing the *slot* number. Missing *Request* messages create discontinuity in the slot number sequences. Once a replica gets aware of a missing slot, it sends an *Expire* message immediately.

2.4.7 Switch Recovery

Switch sequencers can also fail, although rarely. In LRab, the controller monitors the liveness of switch sequencer. When the switch fails, the controller is notified and selects a new switch to act as the sequencer. It also reconfigures the network to force all requests go through the new switch by modifying the flow tables in switches.

In LRab, switch recovery has two components. First, the new switch maintains a total order of instance number, maintained as a session number. Every time a new switch is selected, the controller picks up a new, higher session number for that switch. All slots assigned by the new switch have a higher session number than those assigned by the old switch. Second, the new switch should complete all unresolved slots left by the old switch by making sure all *Commit* messages sent out by previous switches will eventually be delivered to all replicas.

In order to finish all work left by the old switch, the new switch needs to determine the boundary of slot index in the old session, the largest index seen by all replicas. The new switch broadcasts an *Initiate* message to all replicas with *session* number set to the new value. The *Initiate* message sent by new switch queries all replicas for their *current_index*. When receiving an *Initiate* message, replicas reply with *Maxslot* message with *slot_index* set to replica's *current_index* from the old session and *session* set to the new session number the same as *Initiate* message. The switch collects the value of *current_index* from a quorum of *Maxslot* messages and chooses the maximum value T. T is guaranteed to be the largest index of committed slot from old switch among all replicas.

Switch then broadcasts a *Bound* message to inform this boundary to all replicas. After receiving this boundary, each replica checks if it misses any slot, meaning a slot either the replica does not know the decision or the replica has not seen the slot. Replica treats the missing slots the same way as a time out for that slot and triggers the retry process. After a replica resolves all missing slots, it sends a *Ready* message to the switch. When the switch collects *Ready* messages from a quorum of replicas, it can move forward and start normal operation (accept new requests). During the switch fail-over, no new request that can be accommodated before new switch receives *Ready* messages from a quorum of replicas. This has a slight negative impact on throughput but the failure of root switch happens infrequently.

In the switch fail-over procedure, determining the boundary from old switch is necessary for replicas to know what slots they are missing. In normal cases when switch does not crash, any missing slot will be detect eventually as new messages come with higher slot number. This is not the case when the switch crashes. Replicas no longer receive messages with higher slot number in the same session. For example, before the switch crashes, all replicas but one receive the *Request* message and reply back. Consensus is reached. Before switch broadcasts *Commit* message to all replicas, it crashes. For those replicas who only miss the *Commit* message, a time-out will be triggered. However, the replica who does not see the *Request* as well as *Commit* message will never know the existence of the last slot. Therefore, determining the boundary is required for the new switch.

2.5 Optimization

Current Fab uses simple majority as a quorum. Given $2f+1$ replicas, any $f+1$ replicas form a quorum. For each proposed message, switch broadcasts to all replicas. In order to achieve higher throughput, switch only needs to send to a quorum of replicas. Only unsuccessful will switch retry using a new quorum. This can reduce the load on each replica. By carefully constructing quorums, with n replicas, the throughput (proposals per second) can increase to $\sqrt{n}$ [17] compared to broadcasting to all replicas.

After the switch sends out a *Prepare* message, it relays the *Promise* messages to all replicas. A more efficient way is to only relay to a certain replica. That replica will complete the prepare phase. If unsuccessful, switch repeats the process again but relays to a different replica.

2.6 Experiment setup

Our initial evaluation uses Mininet [20] to simulate a star topology with one switch and seven replicas. The switch is written in P4 [5] language and compiled into a C program using P4.org compiler [2]. We run both our system and a standard Paxos implementation, libpaxos [1] for comparison. Libpaxos is a commonly used [15, 8, 19] benchmark to evaluate consensus systems.

2.6.1 Latency

Figure 4 shows the latency distribution of LRab and Libpaxos, with different number of acceptors. The latency is measured from when a request is proposed to when that request gets committed. In general requests in LRab take half of the latency of that in Libpaxos. For example, with three acceptors, the 50 percentile latency for LRab and Libpaxos are 1.690ms and 3.324ms each. The 95 percentile latency for LRab and Libpaxos are 2.632ms and 6.551ms. Similar results apply in different number of acceptors.

The reason why LRab has a smaller latency is because messages in LRab traverses one less round trip. Requests get sorted as they pass through the switch, and there is no need to direct requests to a leader replica.

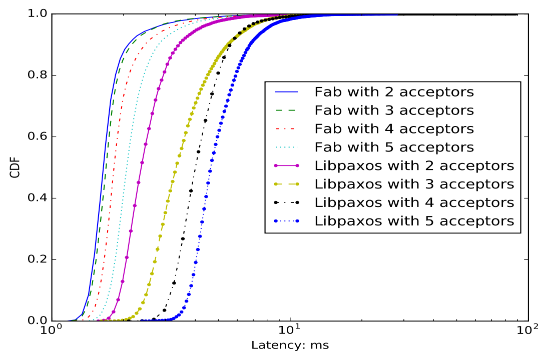

Figure 3: Latency

2.6.2 Throughput

We set the bandwidth of links between each replica and switch to be 10Mbps. This is a limitation as currently all replicas as well as switch are run in the same machine. A low bandwidth network can prevent the system from hitting CPU bottleneck. Following table shows the throughput of two systems with different number of acceptors.

	3 accptors	5 acceptors	7 acceptors
LRab	705 requests/s	700 requests/s	682 requests/s
Libpaxos	412 requests/s	307 requests/s	233 requests/s

Table 3: Average throughout comparison

Libpaxos has a lower throughput because there is no distinguished learner. All of the accept messages must go to the learner. Accept messages saturate the link and lower the effective bandwidth. As number of acceptors increase, throughput will become even lower. LRab, on the other hand, does not have this issue. Switch is responsible to collect the accept messages and only a single commit message will be sent to the learner. In this case switch acts as a distinguished learner without adding one additional hop of latency.

3 Related Work

Lamport's Paxos [10] algorithm was the original pioneer in the field of distributed consensus and forms the basis of LRab, as well as many popular consensus algorithms such as Raft [18] and Zab [9]. We observe that above algorithms are highly leader-driven, a single participant (referred to as leader, master or primary) is responsible achieving consensus. The remaining participants (referred to as followers, backup or replicas) are essentially passive. Commands for the system must be directed to the leader, who is then responsible for ordering commands, proposing commands, determining when a command is decided, handling failures and notifying participants of decided values. The leader is the bottleneck for the throughput of such systems. In fact, adding further participants, adds to the load on the leader and reduces further throughput.

Some Paxos variants [12, 11, 16] have chosen to remove the leader. The leader however, serves a vital role in determining a total ordering on commands. Without it, conflict can become common and performance quickly degrades.

LRab utilizes the network switch for its fast I/O capacity and advantageous position in the network to perform this ordering without becoming a bottleneck. Alternative approaches include S-Paxos [4], which removes the command dissemination responsibilities from the leader, Mencius [14], which rotates the leader for each log index and Corfu [3], which uses a dedicated sequencer with client-driven chain replication. Paxos made switch-y [6] is another recent work trying to build consensus into switch data plane. They implement coordinator and acceptor in the switch. However the work focuses mainly on the normal operation of Paxos while leaving message loss and node fail over handling to upper application. In addition, the coordinator acts only as a sequencer, not a distinguished learner, as it does in LRab. They also require the switch to store the value of message and therefore has to limit the size of messages to be 32 bytes most, which is undesirable.

Network sequencer is studied in work [13]. They argue that an ordered delivery network can greatly simplify the process of consensus. Based on their ordered network model, they propose a Paxos variant that speculatively commit the commands. They still requires a leader to disseminate committed log messages to other replicas. An expensive view change protocol is also required when that leader dies.

4 Conclusion

Asynchronous software increases the complexity and limits the scalability of distributed systems. New programmable hardware on the horizon can help reduce asynchrony, but these new technologies are not as expressive as hardware and have significant limitations. LRab bridges those limitations, leveraging hardware to provide consensus which scales with the ever-growing speed of networking hardware. We have shown how a design of LRab which leverages the P4 packet processing language and RDMA, and presented a preliminary evaluation to show both the limitations of software-based systems and the performance LRab can provide.

References

[1] http://libpaxos.sourceforge.net, 2013.

[2] http://p4.org, 2016.

[3] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.

[4] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120, Oct 2012.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(1):18–24, 2016.

[7] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 350–361. ACM, 2011.

[8] Zsolt Istvan, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, 2016.

[9] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[10] Leslie Lamport. The part-time parliament. *FAST*, 3:15–30, 2004.

[11] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[12] Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2015.

[13] JiaLin Li, Ellis Michael, Naveen Kr.Sharma, Adriana Szekeres, and Dan R.K. Ports. Fast replication with nopaxos: Replacing consensus with network ordering. submitted.

[14] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[15] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P Birman. The performance of paxos in the cloud. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 41–50. IEEE, 2014.

[16] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[17] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 214–225. IEEE, 1994.

[18] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[19] Marius Poke and Torsten Hoefler. Dare: high-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[20] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc), 2012.