Examining Raft's behaviour during partial network failures

Chris Jensen
chris.jensen@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Heidi Howard
heidi.howard@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Richard Mortier
richard.mortier@cl.cam.ac.uk
University of Cambridge
Cambridge, United Kingdom

Abstract

State machine replication protocols such as Raft are widely used to build highly-available strongly-consistent services, maintaining liveness even if a minority of servers crash. As these systems are implemented and optimised for production, they accumulate many divergences from the original specification. These divergences are poorly documented, resulting in operators having an incomplete model of the system's characteristics, especially during failures. In this paper, we look at one such Raft model used to explain the November Cloudflare outage and show that *etcd*'s behaviour during the same failure differs. We continue to show the specific optimisations in *etcd* causing this difference and present a more complete model of the outage based on *etcd*'s behaviour in an emulated deployment using *reckon*. Finally, we highlight the upcoming PreVote optimisation in *etcd*, which might have prevented the outage from happening in the first place.

CCS Concepts: • **Computer systems organization → Availability**; • **Software and its engineering → Software testing and debugging**.

Keywords: Raft, Partial-Partition, etcd, Cloudflare

ACM Reference Format:
Chris Jensen, Heidi Howard, and Richard Mortier. 2021. Examining Raft's behaviour during partial network failures. In *1st Workshop on High Availability and Observability of Cloud Systems (HAOC'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. *https://doi.org/10.1145/3447851.3458739*

1 Introduction

Modern cloud services are deployed across geographically distributed datacenters for performance and resilience. To manage configuration data, these deployments require highly-available strongly-consistent databases like *etcd* [1], which is used in systems such as Kubernetes [2] and Openstack [3].

HAOC'21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8336-3/21/04.
https://doi.org/10.1145/3447851.3458739

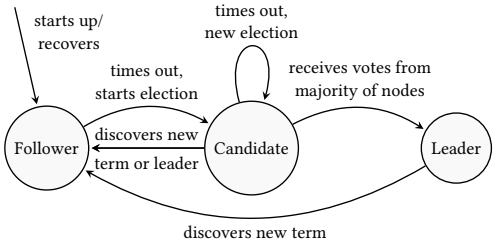


Figure 1. Transitions between the node states for Raft.

When implementing systems such as *etcd*, an implementation of Raft [20], many practical challenges arise that are not covered by the original description.

Challenges for practical deployments include how to ensure performance (the system achieves high throughput and low latency) and availability (the system can always respond correctly to client requests). However, although the protocol specification guarantees correctness, performance and availability depend on the state of the servers and network that connects them. A recent outage at Cloudflare [17] showed the importance of availability. There, despite a well-engineered deployment of *etcd*, they experienced an extended service impairment due to an intermittent failure in a load-balanced switch. They state that the failure caused a partial network partition within their *etcd* cluster which caused it to be live-locked until the failure was resolved.

Following a brief introduction to Raft and partial network partitions, we make the following contributions: (*i*) *reckon*,[1] an open source benchmarking tool that allows for a wide range of topologies and failures to be emulated; (*ii*) an elaborated version of Cloudflare's *etcd* postmortem, showing how Raft can experience multiple leader elections during a partial partition; (*iii*) we highlight some optimisations in *etcd* that make the Raft based model inaccurate, and present a more accurate model of the outage; (*iv*) we highlight PreVote as an existing solution for these problems.

2 Background and related works

Raft [19, 20] is a leader-based consensus protocol for providing strongly consistent state machine replication (SMR) [22] in the presence of node failures and faulty networks. From an application programmer's perspective, a deployment using

[1] *https://github.com/cjen1/reckon*

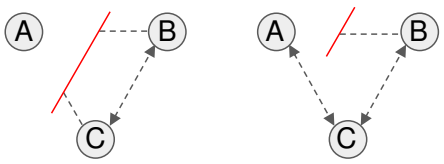


Figure 2. Full (left) vs partial (right) partition.

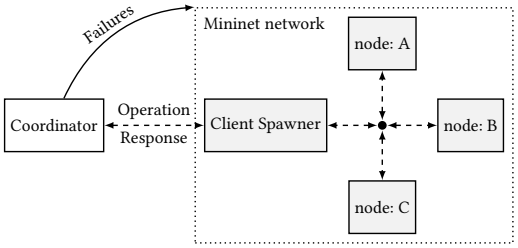


Figure 3. An exemplar deployment of *reckon* used in Section 4. A *Client Spawner* emulates some number of clients accessing the system from a given network gateway. An external *Coordinator* generates the mininet network and passes the requests to the client spawner, as well as injecting failures into the system.

SMR behaves as a single machine even when implemented using physically distributed hosts. SMR is achieved in Raft based systems by replicating a shared log, often read and write requests, which can then be applied to a state machine.

Raft replicates the shared log by using rounds of decisions, denoted by monotonically increasing *terms*, where within a *term* a leader is elected to decide the ordering of new entries appended to it. To prevent old leaders from influencing future *terms*, every node will only participate in the newest (highest) *term* it has heard of. Since each node includes its current *term* in every message, messages from older (lower) *terms* are rejected, and alert the sender to the new *term*.

Raft's mechanism for transitioning to a new *term* is based on first determining if the leader for that *term* has crashed (or does not exist). Each follower tracks when it last heard from the leader, and if that is greater than an *election timeout*, as shown in Figure 1, it will become a candidate and call an election. It does this by incrementing its *term*, sending election requests to the other nodes and finally waiting for a majority to vote for it. If the sender's log is at least as up to date as the recipient's, and the recipient has not voted for any other node during that term, it will vote for the sender. In the situation that a candidate does not receive sufficient votes within its timeout, it will retry with a higher *term*. Additionally, leaders periodically send empty heartbeat requests to their followers to avoid being declared crashed.

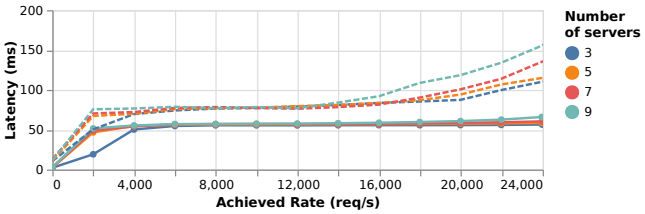
When a client wants to submit a request to be replicated, it first contacts a random server, learns of the leader, and dispatches its request directly to it. The leader will then add that entry to its log before replicating it to the other nodes. If a follower of the leader receives the message, it will add the entry to its log and reply with an acknowledgement. Once an entry has been added to the log of a majority of nodes, the leader will mark it as committed, apply the request to its state machine and return the result to the client.

The main reason for SMR as implemented in Raft and similar protocols is that they increase the reliability of a system by tolerating some nodes crashing, or being partitioned away from the majority. For example, if a Raft cluster experiences a full network partition, and the leader is partitioned from the majority, then, having received no messages from the leader, a node in the majority can call and win an election. Although a node which is fully partitioned away from the majority can be treated as having crashed, this is not the case with partial partitions. For example, as seen in Figure 2 given three fully-connected nodes, A, B and C, a full network partition results in A being unable to communicate with B and C, while in a partial network partition A can still communicate with C. This means that some information from A can reach B via C. Since they are non-trivial to reason about and can have a large impact on production cloud systems, partial partitions are a problem that has recently drawn the attention of the research community [5, 6].

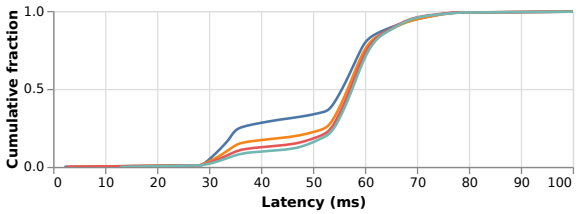
As critical components in modern distributed systems, there have been many studies of the performance of Raft and other SMR protocols [13, 18, 21, 24]. However, most either do not consider failures or consider only node crashes.

One approach, which allows one to test a wide variety of failures, is to build a model of the system and simulate injecting failures into it. An example of this is Paxi [4], a prototyping framework for modelling strongly-consistent replication protocols, that supports partial network failures. However, it requires protocols to be re-implemented within its framework and so modelling real-world systems results in a parallel implementation being maintained. Having parallel implementations can be problematic: it is common (as we will see in this paper with *etcd* and Raft) for the implementation to diverge from the specification. Another separate modelled implementation is an opportunity for further divergence.

Another approach is to try multiple deployments and introduce failures into those. Both Jepsen [12] and NEAT [6] use this approach. Jepsen is a database testing framework, which takes a deployment of servers and applies requests against them. Concurrently, it injects failures into these systems, for example killing nodes while the system is running and checks the transactions for properties such as transaction isolation and linearisability. Similarly, NEAT takes a deployment of servers and injects partial partitions into it. It has shown that partial partitions can cause data-loss due to conflicting views of reality between nodes.



(a) Median (solid line) and 99th percentile (dotted line) latency at the achieved throughput when the target is increased from 1 to 24k req/s.



(b) Measured latency when driven at 10,000 req/s.

Figure 4. Write performance of *etcd* deployed with $N = 3, 5, 7, 9$ nodes.

3 Emulating consensus

Reckon allows investigators to cheaply reproduce real world failures by taking a midpoint between modelling everything and physically reproducing the failure. Specifically, we use Mininet [10, 15] to run a production system inside of an emulated deployment and to inject failures into it. Although this approach sacrifices some accuracy in terms of network and node resource constraints, it allows us to cheaply test real systems in a wide range of deployment and failure scenarios.

Mininet uses network namespaces and cgroups as lightweight virtualisation mechanisms to emulate each node, connecting them using virtual Ethernet devices and software switches. The result is that both node and network properties can be varied dynamically by altering the Mininet configuration. Though originally focused on testing new Software Defined Networking approaches [14], it also allows arbitrary programs to be run on the emulated nodes. *Reckon* thus builds a specified topology in Mininet, deploys production consensus systems and simulated clients to the virtualised nodes, and exercises network configuration APIs to inject failures. Figure 3 shows the setup used in Section 4.

A test begins with the Coordinator transferring a time series of requests to the clients before waiting for a ready signal from each client. Once all ready signals are received it sends a synchronised start signal to all clients. This reduces control path interference in the test results.

A key concern for a modern load generator is to avoid *coordinated omission* [23], where a synchronous closed-loop load generator delays sending subsequent requests, while waiting for an exceptionally slow request to complete. Although most load generators use a closed-loop client model, most clients of cloud systems are open-loop, each submitting only a small number of requests. This results in a mismatch when only a single high latency request is observed when in an open-loop model multiple requests would.

Two existing approaches to resolving this concern are as follows: (*i*) use an open-loop load generator, as used in NoSQLMark [9]; and (*ii*) record request latency starting from when the request should have been sent rather than when it was actually sent, as used by YCSB [8]. Although the latter approach is simpler to implement and can have higher performance, it also exhibits stalling artefacts during failures where some requests do not return while others succeed. In these cases, for each request a thread sends, if it does not return that thread can no longer apply requests, thus eventually all threads will block. As a result, we use an open-loop system model for the clients.

To provide some basic confidence in the tool, Figure 4 reports request latency and throughput of *etcd* running inside *reckon*. For *etcd*, the maximum throughput we can test is 24 000 req/s and achieve a latency of 60 ms to 100 ms. At target rates above 24 000 req/s, the arrival rate of requests will exceed *etcd*'s maximum service rate and hence the number of concurrent requests within *etcd* increases. Once a limit is breached, *etcd* rejects requests with the error "too many requests", invalidating those tests.

For this test, and the tests in Section 4, we use a simple topology of all nodes and a single client spawner connected to a central switch. We do not limit the latency or bandwidth of those links and thus, in theory, the links are instantaneous with unlimited bandwidth, but in practice, they are limited by the underlying hardware. The nodes are running *etcd* v3.4.14 with an election timeout of 500 ms and a heartbeat of 100 ms, while the client spawner uses a single Golang v1.11 *etcd* client v3.4.14 to asynchronously apply requests. The requests are write requests to random 8 byte keys with 10 byte payloads. The host system is running an Intel Xeon 4112 (16 core) processor, with 196GB of RAM and a Dell 4T7DD SSD. Although each test has been run five times, for clarity we only show a single representative trace.

4 Reproducing Cloudflare's outage

On November 2nd 2020, Cloudflare experienced a failure for six minutes in a load-balanced switch, during which time their *etcd* cluster became unavailable [17]. As their database cluster management system uses *etcd* for cluster member discovery and coordination, *etcd* being unavailable caused it to promote a new primary database. This in turn caused a rebuild of all their replicas, putting their control plane in a degraded state for six and a half hours.

3

They state that the failed switch created a partial partition, blocking communication between two nodes (A and B) of the three-node cluster. Since the follower (A) was no longer receiving any communication from its leader (B), it "repeatedly initiated leader elections, voting for itself, while node [C] repeatedly voted for node [B] which it could still connect to". This re-elected B and restarted the cycle. Since "leader elections are disruptive, blocking all writes until they're resolved", this makes the cluster unavailable until the failure is fixed.

We demonstrate the utility of *reckon* by emulating the effect of the failed switch directly on an *etcd* cluster and comparing its behaviour to what we would expect from Raft.

We model the scenario as follows. Our network is three nodes connected directly to a central switch, as well as a client spawner attached to the switch (Figure 3). To mimic the failed switch and subsequent partial partition, we inject *iptables* rules to block traffic. Finally, as the main impact of the failure was to make the cluster unavailable to writes, we use a write-only workload of 1000 reqs/s from a single open-loop client.

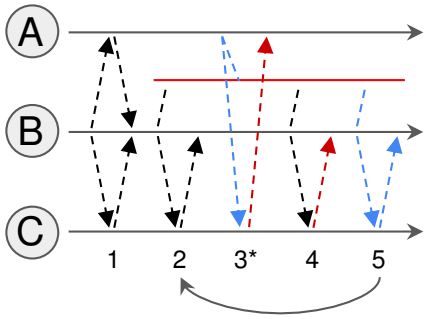
Raft's expected behaviour during a partial partition is shown in Figure 5a. Although there may be multiple leader elections, on each iteration C has an opportunity to call an election which it can win with a vote from A. C's promotion would result in the partial partition no longer affecting the cluster. Note that this possibility makes Cloudflare's situation more difficult to reproduce since their elections repeatedly "did not promote a leader node [A] could reach".

4.1 Leader crash

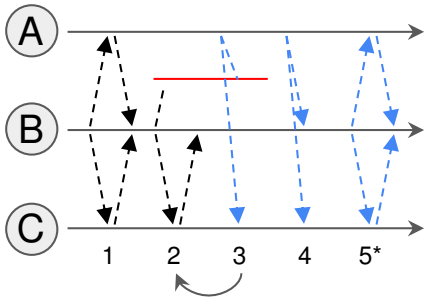
We begin by showing the standard behaviour of *etcd* when a leader crashes (Figure 6a). When B is killed at 20 s, an election occurs and, per the Raft specification, requests submitted during this time are delayed until the election completes. When B is brought back online at 40 s, it receives all the requests it has missed, spiking its bandwidth to 2.5MB/s. Additionally, since B coming online does not affect the availability of the system, there is no change in successful requests.

The final feature of this figure is that rather than solely messaging the leader, the client messages all nodes. This is due to load-balancing behaviour in the clients and the servers. Specifically, for each request, *etcd* clients dispatch it to a node chosen by some strategy. If the recipient is not the leader, it will forward the request to the leader. In *etcd* v3.4 it maintains a separate TCP connection to each server and uses a simple round-robin load-balancing strategy.[2] Therefore, the client sends an equal proportion of traffic to each node and once B has failed, there is no more traffic between it and the client. Sometime after B recovers the client retries the connection and, finding it working, adds it back to the pool.

[2] *https://github.com/etcd-io/etcd/pull/9860*



(a) Possible behaviour of Raft: ① Initially B is the leader and can replicate entries (black) to A and C. ② During the partition B can no longer replicate entries to A. ③ Since A is not receiving messages from B, from its' perspective B has crashed, so it increments its term and calls an election (blue). The partition blocks the election request to B, however C still receives it and hence updates its term before rejecting the request (red) due to A's missing log entries. (∗) At this point, C can call an election. ④ If B tries replicating an entry, C rejects it since B now has a lower term. As a result, B updates its term and resigns. ⑤ B can now call an election where C will vote for it, returning the cluster to the same state as at the start of the partition (②).

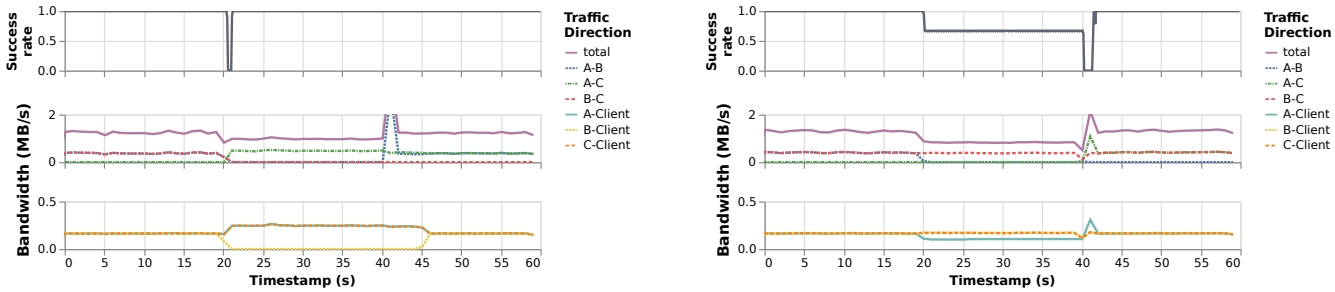


(b) *etcd*'s behaviour during a partial partition: ① Initially B is the leader and can replicate entries to A and C. ② During the partition B cannot replicate entries to A. ③ From A's perspective B has timed out and it calls an election, but in contrast to Raft, C ignores this request. As a result, B does not need to step down and can continue to replicate requests. ④ When the partition is removed, A's election request is received by B, which causes B to resign, and ⑤ call another election, which it can win. (∗) After B steps down C also has a chance to be elected.

Figure 5

4.2 Partial partition

Figure 6b shows a trace of *etcd* during a partial partition. In contrast to the expected behaviour of Raft, *etcd* does not experience any leader elections during the partition and only experiences a single one afterwards. Additionally, even though there are no leader elections, *etcd* clients experience degraded performance throughout the partition, with a third of requests being substantially delayed. A message trace of *etcd*'s behaviour during the partial partition, reconstructed from its logs, is shown in Figure 5b.



(a) Leader crash and subsequent recovery.

(b) Partial partition between A and its leader (B).

Figure 6. Rate of successful (completed in < 0.1 second) requests with bandwidth used under different conditions. Bandwidth direction has been normalised to the naming convention in Figure 5. Failures are injected at 20s, recovery is at 40s.

Examining some optimisations that *etcd* has accumulated explains why *etcd*'s behaviour diverges from Raft's.

One of these optimisations is *CheckQuorum*. As discussed in Section 6 of the Raft paper [20], *CheckQuorum* means that if a node has a stable leader it will ignore election requests from other nodes. It promotes stability when cluster members change and "if a leader is able to get heartbeats to its cluster, then it will not be deposed by larger term numbers."

Although *etcd* enables this optimisation for followers, it is currently disabled for leaders.[3,4] This means that rather than A's election request causing C to reject B's replication request, it is ignored. Hence, B does not learn of A's higher *term* during the partition and can continue to service requests. Once the partition is healed, B receives A's requests causing it to step down and another election is called.

The other optimisations are the client request forwarding and round robin dispatch behaviour discussed in Section 4.1. Considering requests that arrive at A during the partition since A believes the leader has crashed, it has nowhere to forward these requests. These requests time out and are successfully retried against another node. Since the client uses a simple round-robin stragtegy to choose targets, it will dispatch two third of its requests to B and C which will proceed as normal and a third to A which will be delayed. Thus, we arrive at a third of requests sent during the partition being delayed and, by our threshold, unsuccessful. Previous versions of *etcd*'s client (<v3.4) would stop using the connection after detecting multiple failing requests. However the approach was scrapped because it was "so tightly coupled with old gRPC interface, that every single gRPC dependency upgrade broke client behavior."[5] Although these requests are delayed, they are still eventually successful, and hence are a *performance* issue rather than an *availability* issue.

The final behaviour is that when the partition is healed either B or C could be elected (in Figure 6b, C is elected). Since

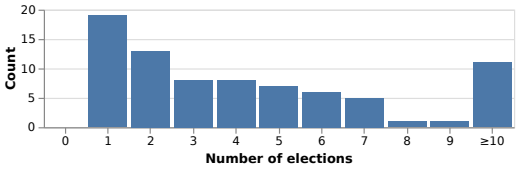


Figure 7. Number of elections before *etcd* elects C and recovers from an intermittent partial partition of A from B. We stop the test after 10 partial partitions, and the first one is always between the initial leader and a follower.

A did not receive updates from B during the partition, its log will be missing entries. Thus, when an election is called A can vote for either B or C, since they have longer logs. Therefore, after the partition both B and C can be elected having received a vote from A and themselves.

4.3 Intermittent partial partition

Using the behaviour shown in Figure 6b we can start to more fully reproduce the failure observed by Cloudflare. In the postmortem, Cloudflare states that the switch failure meant that each server "only sees an issue with some of its traffic due to the load-balancing nature of LACP". We model this failure as an intermittent partial partition that transitions repeatedly between the partition being in place or removed.

Unfortunately this does not produce repeated leader elections throughout the failure. Instead, there is a window in which C can call, and win, an election. Thus subsequent partitions will not affect the cluster. Figure 7 shows the number of iterations before the cluster recovers.

4.4 Intermittent full partition

If instead we intermittently *fully* partition A from *both* B and C, as in Figure 2 and Figure 8, then no matter which of B or C is elected, A cannot reach it when the partition is put back in place, causing the cycle to repeat. Additionally, as the time between failures decreases it will cause a greater proportion of time to be spent in leader elections. Although this

[3] *https://github.com/etcd-io/etcd/issues/12673*
[4] *https://github.com/etcd-io/etcd/commit/a7a867c1*
[5] *https://etcd.io/docs/v3.4/learning/design-client/*

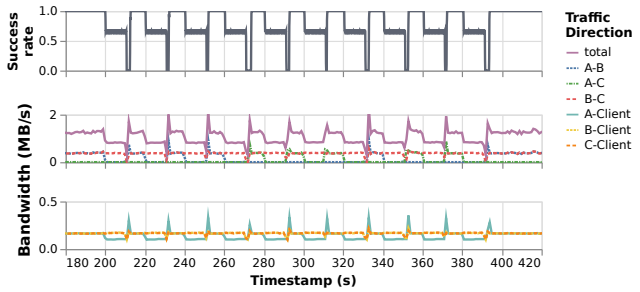


Figure 8. Intermittent full partition of a follower (A) from the remainder of the cluster (B and C), transitioning every 10s between the partition being in place or healed.

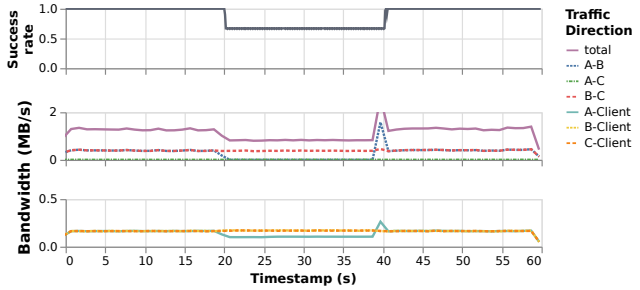


Figure 9. Partial partition when *PreVote* is enabled

explanation diverges from Cloudflare's network level failure, it more closely replicates the application level behaviour of multiple leader elections causing a prolonged outage.

5 Conclusion

In this paper, we have used *reckon* to reproduce Cloudflare's November 2020 outage in their *etcd* cluster. We show that due to various optimisations in *etcd*, the explanation based on Raft is incomplete. Specifically, although with Raft we would expect a leader election soon after the partial network partition is put in place, we find that due to *etcd*'s *CheckQuroum* optimisation there are no leader elections except one when the partition is healed. Additionally, we find that during the partition, a third of requests are delayed due to both *etcd*'s client dispatch strategy and *etcd*'s request forwarding behaviour. Finally, we conclude that an intermittent full partition of a follower from the remainder of the cluster may have caused the outage. Although this diverges from Cloudflare's network level partial partition, it more closely reproduces the high-level behaviour of repeated leader elections throughout the failure.

This class of failures, where a faulty node repeatedly calls leader elections, disrupting the cluster is well known and a theoretical fix exists. This fix is the *PreVote* optimisation, which requires that before any node calls an election, it first checks that it can be elected by requesting pre-votes from nodes that would vote for it. This means that when the

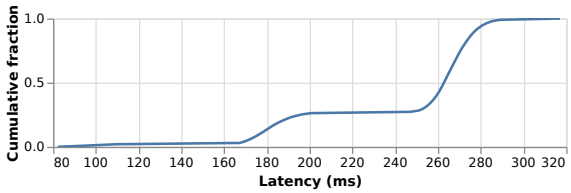


Figure 10. CDF of write request latency in an edge style deployment. This emulates 7 colocated nodes with 40 ms links between them, with write requests equally distributed among the colocated clients. In this topology single round-trip-time (RTT) (80 ms) commits can only occur if the client is co-located with the leader and dispatches directly to it. Otherwise they achieve a minimum of either 2∗RTT (160 ms) by dispatching directly to the leader (or to the co-located node which forwards to the leader); or 3 ∗ RTT (240 ms) if they choose to send to an intermediate node.

follower A is partitioned from the leader, although it will time out and try calling an election, it will not get sufficient pre-votes to do so, thus preventing it from disrupting the cluster. PreVote has been available in *etcd* since v3.4,[6] however, it was not enabled by default due to concerns over its production-readiness. It will be enabled, by default, in the upcoming *etcd* v3.5 release. Figure 9 is the same test as in Figure 6b except with *PreVote* enabled, and shows that there are no leader elections throughout the partition nor an election afterwards. Although *PreVote* would likely have prevented the outage observed at Cloudflare, since it does not change client request semantics, any requests sent to A during the partition are still substantially delayed. Another potential fix is via Overlay Networks [5, 7, 16]. These effectively reroute around partial partitions, resulting in B's messages to A going via C. Additionally they may solve the more general issue of delayed client requests since if a client can reach both sides of a partition then communication between the sides can occur via the client.

We believe that *reckon* can be a valuable tool for both researchers looking to test a production system's availability in various deployments and failure scenarios, and for industry practitioners looking to pre-emptively test edge cases during development. In the future, we aim to expand our analysis to systems such as Zookeeper [11], as well as edge networks. To that end, we can currently emulate a simple edge deployment. In Figure 10 we depict the CDF of request latency in an edge style deployment and show that single round-trip commits can only occur for a small fraction of requests. We have made *reckon* available under open-source licences, and we would warmly welcome pull requests for new topologies, failure modes, and other features.

Acknowledgements. This work was supported in part by EPSRC EP/R03351X/1 and EP/M02315X/1.

[6] *https://kubernetes.io/blog/2019/08/30/announcing-etcd-3-4/*

References

[1] [n.d.]. etcd Project Homepage. *https://etcd.io*. Accessed : 2021-01-08.

[2] [n.d.]. Kubernetes Project Homepage. *https://kubernetes.io*. Accessed : 2021-01-08.

[3] [n.d.]. Openstack Project Homepage. *https://www.openstack.org*. Accessed : 2021-01-08.

[4] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1696–1710. *https://doi.org/10.1145/3299869.3319893*

[5] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. *https://www.usenix.org/conference/osdi20/presentation/alfatafta*

[6] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 51–68. *https://www.usenix.org/conference/osdi18/presentation/alquraan*

[7] Yair Amir and Claudiu Danilov. 2003. Reliable communication in overlay networks. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* 511–520. *https://doi.org/10.1109/DSN.2003.1209961*

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. *https://doi.org/10.1145/1807128.1807152*

[9] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. 2017. Coordinated Omission in NoSQL Database Benchmarking. In *Datenbanksysteme für Business, Technologie und Web*. 215–225.

[10] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible Network Experiments Using Container-Based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (Nice, France) *(CoNEXT '12)*. Association for Computing Machinery, New York, NY, USA, 253–264. *https://doi.org/10.1145/2413176.2413206*

[11] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*. *https://dl.acm.org/doi/10.5555/1855840.1855851*

[12] Kyle Kingsbury. [n.d.]. Jepsen: Distributed Systems Safety Research. *https://jepsen.io*

[13] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. *https://doi.org/10.1145/3342195.3387545*

[14] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteves Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE* 103, 1 (2015), 14–76. *https://doi.org/10.1109/JPROC.2014.2371999*

[15] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (Monterey, California) *(Hotnets-IX)*. Association for Computing Machinery, New York, NY, USA, Article 19, 6 pages. *https://doi.org/10.1145/1868447.1868466*

[16] Zhi Li, Lihua Yuan, Prasant Mohapatra, and Chen-Nee Chuah. 2007. On the analysis of overlay failure detection and recovery. *Computer Networks* 51, 13 (2007), 3828–3843.

[17] Tom Lianza and Chris Snook. [n.d.]. A Byzantine failure in the real world. *https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/*. Accessed : 2021-01-08.

[18] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman. 2014. The Performance of Paxos in the Cloud. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS '14)*. IEEE Computer Society, USA, 41–50. *https://doi.org/10.1109/SRDS.2014.15*

[19] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph.D. Dissertation. Stanford University. *https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf*.

[20] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320. *https://dl.acm.org/doi/10.5555/2643634.2643666*

[21] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Dumitrel Loghin, Beng Chin Ooi, and Meihui Zhang. 2019. Blockchains and Distributed Databases: a Twin Study. arXiv:1910.01310 [cs.DB]

[22] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. *https://doi.org/10.1145/98163.98167*

[23] Gil Tene. [n.d.]. How NOT to measure latency. *https://www.infoq.com/presentations/latency-response-time/* Accessed : 2021-02-09.

[24] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and How to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 445–454. *https://doi.org/10.1145/3293611.3331595*