Title

Reducing the Energy Footprint of a Distributed Consensus Algorithm

Permalink

https://escholarship.org/uc/item/9bs3772z

Authors

Pâris, Jehan-François
Long, Darrell DE

Publication Date

2015

DOI

10.1109/edcc.2015.25

Copyright Information

Reducing the Energy Footprint of a Distributed Consensus Algorithm

Conference Paper · September 2015

DOI: 10.1109/EDCC.2015.25

2 authors:

Jehan-Francois Paris
University of Houston
165 PUBLICATIONS **2,570** CITATIONS
SEE PROFILE

Darrell D. E. Long
University of California, Santa Cruz
316 PUBLICATIONS **9,286** CITATIONS
SEE PROFILE

Reducing the Energy Footprint of a Distributed Consensus Algorithm

Jehan-François Pâris
Department of Computer Science
University of Houston
Houston, TX, USA 77204-3010
jfparis@uh.edu

Darrell D. E. Long[1]
Department of Computer Science
University of California
Santa Cruz, CA, USA 95064
darrell@cs.ucsc.edu

***Abstract*—The Raft consensus algorithm is a new distributed consensus algorithm that is both easier to understand and more straightforward to implement than the older Paxos algorithm. Its major limitation is its high energy footprint. As it relies on majority consensus voting for deciding when to commit an update, Raft requires five participants to protect against two simultaneous failures. We propose two methods for reducing this huge energy footprint. Our first proposal consists of adjusting Raft quorums in a way that would allow updates to proceed with as few as two servers while requiring a larger quorum for electing a new leader. Our second proposal consists of replacing one or two of the five Raft servers with witnesses, that is, lightweight servers that maintain the same metadata as other servers but hold no data and can therefore run on very low-power hosts. We show that these substitutions have little impact on the cluster availability but very different impacts on the risks of incurring a data loss.**

Keywords-Distributed computing; Fault-tolerant computing; Green computing; Distributed consensus; Paxos; Raft algorithm

I. INTRODUCTION

Distributed consensus algorithms allow multiple participants in a distributed system to agree on the values of some replicated data. They are essential to the development of fault-tolerant services because they allow multiple servers to act as one. They are also notoriously complex because they have to handle both server crashes and all kinds of communication failures.

We can distinguish two main classes of consensus algorithms depending on the kinds of faults they tolerate. Byzantine consensus algorithms assume that faulty components of a distributed system could send incorrect, inconsistent messages to their peers [8]. Other consensus algorithms assume that the system components will either operate correctly or stop operating. In other words, they will only experience *fail-stop* failures [17]. The best known non-Byzantine consensus algorithm is Leslie Lamport's Paxos algorithm [10, 11]. Paxos offers the two advantages of having been proved to be correct and being efficient in the standard case. At the same time, it is both hard to understand and difficult to implement [14]. Consider for instance the *Chubby* lock service [3, 4]. It was designed to provide coarse-grained locking for loosely-coupled distributed systems and offer limited amount of reliable storage. Its first version [3] relied on a third-party fault-tolerant database that had known replication-related bugs. It was later decided to replace this version with a locally developed solution that would use the Paxos algorithm [4]. Reporting on their experience, its authors mention that "[t]here are significant gaps between the description of the Paxos algorithm and the needs of a real-world system."

Ongaro and Ousterhout recently proposed the Raft consensus algorithm to overcome these limitations [13, 14]. As noted by Howard *et al.*, Raft is easier to understand and easier to implement than Paxos [6]. A remaining limitation of Raft is the large number of servers it requires. Because it uses majority consensus voting for deciding when to commit an update, Raft requires $2n + 1$ participants to protect against n simultaneous failures. As a result, most Raft clusters use five servers in order to be able to tolerate two failures. In other words, a single Raft cluster has the same power requirements as ***five*** non-replicated servers. These requirements are likely to remain acceptable as long as Raft supports lightweight services that can run on low-power nodes comprising an energy-efficient processor and no hard disk. This is much less true when applications require full-fledged servers.

We propose to address this issue by investigating the possibility of implementing Raft on clusters with fewer than five full-fledged servers. We note that failures that affect the durability of the data stored on a Raft cluster are much less frequent than those that affect the availability of the service it implements. We then introduce solutions that address separately these two issues. Our first proposal consists of adjusting Raft quorums in a way that would allow updates to proceed with as little as two servers while requiring a larger quorum for electing a new leader. So, a Raft cluster with four nodes, an update quorum of two and a leader election quorum of three would offer good protection against data loss while remaining slightly more available than a conventional Raft with three nodes. Our second proposal consists of replacing one or two of the five servers with *witnesses* [15]. In this context, these witnesses will be lightweight Raft servers that maintain the same metadata as other servers but hold no data. As a result, they can run on very low-power hosts such as the Raspberry Pi [22]. We show that both configurations provide similar cluster availabilities as a Raft cluster with five full servers and very good to adequate protection against data loss depending on the number of servers replaced by witnesses.

[1] Supported in part by the National Science Foundation under awards CCF-1219163 and CCF-1217648, by the Department of Energy under award DE-FC02-10ER26017/DE-SC0005417 as well as by the industrial members of the Storage Systems Research Center.

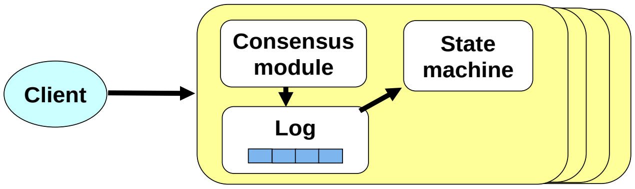

Fig. 1. The Raft architecture (after [OO14]).

The remainder of this paper is organized as follows. Section II reviews the main features of the Raft algorithm. Section III presents our overall approach for reducing its energy footprint. Section IV introduces our Raft cluster with four servers and evaluates its robustness in terms of system availability and risk of data losses. Section V does the same for RAFT configurations including witnesses. Finally, Section VI has our conclusions.

II. The Raft Algorithm

In this section, we present the salient features of the Raft algorithm focusing on its update algorithm and the protocol it uses for handling leader failures.

A. Overview

Ongaro and Ousterhout designed the Raft algorithm to run on clusters consisting of at least three servers, like the one in Fig. 1. Each of its component servers includes a log, a consensus module and a state machine. Raft guarantees that, at any time, a majority of these state machines will remain in agreement. In other words, each Raft cluster implements a replicated state machine [9, 18].

Raft uses a strong leader model where each cluster has a single leader that manages the whole cluster and other servers are mere followers. As a result, the cluster leader is solely habilitated to receive requests from clients, forward them to its followers and decide when they can be safely applied to everyone's state machines.

Leaders maintain their leadership status by sending periodic heartbeats to their followers. Any follower that stops receiving these heartbeats will call for an election and propose itself as the new leader.

Raft partitions time into *terms* of arbitrary length that are identified by consecutive sequence numbers. A new term starts each time a server calls for an election. It will either end if the election results in a split vote or continue as long as the newly elected leader maintains its leadership status. Terms play in Raft the role of a logical clock [7] and allow servers to detect obsolete information, such as requests from a stale leader. All communications between servers include the sequence number of the current term.

B. Normal operation

When a leader receives a request from a client, it appends it to its log, gives it a sequence number within the current epoch and forwards it to its followers through an AppendEntry remote procedure call. All followers that have up-to-date logs append the new command to their logs and notify the leader of that fact. Whenever the leader notices that some followers did not reply, it resends the command and repeats the process until all followers have acknowledged the request.

As soon as the leader has replicated the log entry on a majority of the servers, it *commits* it and applies it to its own state machine. This action also applies to all preceding entries in the leader's log, including entries created by a former leader in a previous epoch. Commit decisions are propagated to other servers by including the index of the last committed update in all future messages sent by the leader, including AppendEntry calls and heartbeats.

C. Handling leader failures

Raft uses a timeout mechanism for detecting leader failures: any follower that has not received a message from its leader in a given amount of time will call for an election, announce its candidacy for the cluster leadership position and vote for itself. A main problem with this solution is that split votes will occur whenever two followers call elections at the same time. Raft reduces, but does not completely eliminate this risk, by using randomized election timeouts.

When a former follower becomes a candidate for the cluster leadership position, it sends a message containing a summary of the state of its log to all other servers. Servers receiving that message will vote for the candidate unless any of following three conditions holds:

1. They believe they still have a leader,
2. They have already voted for another candidate, or
3. Their own log is more "up to date" than the candidate's log.

The last restriction ensures that candidate cannot collect a majority of the votes unless its log contains all committed updates.

The newly elected candidate will require all its followers to duplicate in their logs the contents of its own log. To achieve that, it will resend to each of its followers all its log entries starting from the last entry for which both servers agree.

D. Cluster membership changes

RAFT handles cluster membership changes by requiring the change to involve both a majority of the servers in the old cluster and a majority of the servers in the new cluster.

III. Reducing Raft Energy Footprint

The major motivation for having five servers in a Raft cluster is to allow the cluster to tolerate two simultaneous failures. This property holds because a Raft cluster with five servers will accept updates from it clients as long as three of its servers can participate in an update quorum and will never commit updates to fewer than three servers.

Requiring five servers in each Raft cluster is likely to remain acceptable as long as Raft supports lightweight services that can run on low-power architectures built around an energy-efficient processor, a small amount of memory and a flash drive replacing the hard disk. Exemplars of these architectures include Pergamum tomes, but without their attached disk, [20], FAWN nodes [1] and the Raspberry Pi [22]. This is much less true when applications require full-fledged servers that have a non-negligible energy footprint.

We propose to reduce this emerging footprint by running Raft on configurations with fewer conventional servers. At present, we do not want to alter in any way the logic of the Raft algorithm, which excludes the use of dynamic voting algorithms [5].

We define the *availability* of a Raft cluster as the fraction of time it will be able to process user requests and the *durability* of its log updates by the probability it will never lose any committed update. Typical Raft clusters satisfy these requirements by guaranteeing that:

1. The cluster will remain available as long as three of its five servers remain available, and
2. All log updates will always apply to at least three logs.

We believe that this second requirement is excessive. First, disk failures are much less frequent than server crashes. Even assuming a disk failure rate of 11.8 percent per year, which is typical for consumer disk drives at the very end of their useful lifetime [2], disk mean times to failure would remain close to eight years and a half and an individual disk would have a 99.97 percent probability of not failing over a 24-hour interval. This is to say that we may let a cluster occasionally run with a single server as long as it happens infrequently and for short time intervals.

When stronger durability guarantees are required, running the cluster with at least two operational servers should be enough. While irrecoverable read errors are still a possibility, their impact should remain negligible as long as the size of the log and its state machine are less than a few hundred megabytes.

The solutions we propose are tailored to three different scenarios with different demands on cluster availability and log update durability:

1. When log update durability is more important than cluster availability, a Raft cluster with four servers will offer both a much better update durability and a higher data availability than a Raft cluster with three servers.
2. When log update durability and cluster availability, are equally important, a Raft cluster with four servers and one witness will offer nearly the same availability than a Raft cluster with five servers and almost as strong guarantees on update durability.

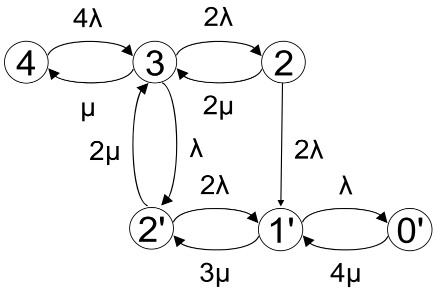

Fig. 2. A Raft cluster with four servers, an update quorum of two and an election quorum of three.

3. When cluster availability is more important than update durability, a Raft cluster with three servers and two witnesses will provide nearly the same availability than a Raft cluster with five servers but have much weaker guarantees on update durability.

IV. Raft clusters with four servers

The correctness of the Raft algorithm requires that its update quorum and its leader election quorum intersect in order to ensure that the new leader will always have the most up-to-date version of the update log and its associated state machine. The algorithm satisfies this condition by using majority consensus voting [19, 21]. Using distinct quorums would not affect the correctness of the algorithm as long as:

1. The update quorum and the leader election quorum intersect.
2. The leader election quorum remains greater than or equal to the update quorum.

Consider now a Raft cluster with four nodes, an update quorum of two and a leader election quorum of three. It would guarantee that all updates will always be applied to at least two servers and require three servers to recover from the loss of a leader. The main advantage of this scheme is that most updates will be applied to all servers. A smaller fraction of the updates will be applied to three of the four servers and even fewer updates applied to two of the four servers. Hence, the cluster will have a significantly lower risk of a data loss than a Raft cluster with three clusters.

The behavior of our four-server scheme can be described by its state transition diagram. As Fig. 2 shows, the cluster has six possible states. State <4> is the original state where all four servers are available. A failure of one of the servers would bring the cluster to state <3>. We note three distinct transitions from this state:

1. A recovery transition corresponds to a repair of the server that crashed and returns the cluster to state <4>.
2. A failure of one of the two followers of the current leader of the cluster will bring the cluster to state

<2>. Note that the cluster will still be able to accept log updates since it still satisfies the update quorum of the cluster.

3. A failure of the current leader of the cluster will bring the cluster to state <2'>. Note that the cluster will then be both unable to accept log updates because it lacks a leader and unable to elect a new leader because that would require three servers.

Both states <2> and <2'> have similar failure transitions to state <1'> and recovery transitions to state <3>. State <1'> corresponds to the state of the cluster when only one server remains available and the system cannot accept any update. The state has a failure transition to state <0'> and a recovery transition to state <2'>. Finally, state <0'> has a recovery transition to state <1'>.

To derive the cluster availability, we need to make a few additional assumptions. We will model our cluster as a set of servers with independent failure modes. Whenever a server fails, a repair process is immediately initiated for that server. Should several servers fail, this repair process will be performed in parallel on those servers. We assume that server failures are independent events and are exponentially distributed with mean λ. In addition, we require repairs to be exponentially distributed with mean μ. Both hypotheses are necessary to represent our system by a Markov process with a finite number of states.

The equilibrium conditions for our system are:

$$\begin{aligned}
4\lambda p_4 &= \mu p_3, \\
(3\lambda+\mu)p_3 &= \lambda p_4 + 2\mu(p_2 + p_{2'}), \\
(2\lambda+2\mu)p_2 &= 2\lambda p_3, \\
(2\lambda+2\mu)p_{2'} &= \lambda p_3 + 3\mu p_{1'}, \\
(\lambda+3\mu)p_{1'} &= 2\lambda(p_2+p_{2'}) + 4\mu p_{0'}, \\
4\mu p_{0'} &= \lambda p_{1'},
\end{aligned}$$

where p_i is the probability of the cluster being in state <i>, with the additional condition:

$$p_4 + p_3 + p_2 + p_{2'} + p_{1'} + p_{0'} = 1.$$

Solving the system of linear equations and substituting $\rho = \lambda/\mu$ we obtain:

1. The availability of the cluster $A_4(\rho)$

$$A_4(\rho) = p_4 + p_3 + p_2 = \frac{(1+5\rho+8\rho^2)}{(1+\rho)^5},$$

2. The probability $P_{4,2}(\rho)$ that the cluster will accept updates with only two of the four servers available:

$$P_{4,2}(\rho) = p_2 = \frac{4\rho^2}{(1+\rho)^5}.$$

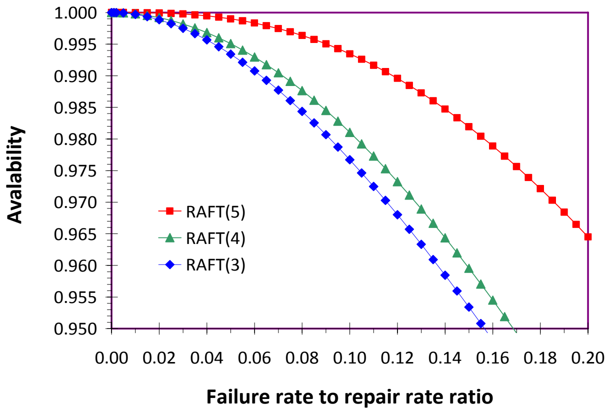

Fig. 3. Compared availabilities of Raft clusters with three, four and five servers.

Deriving the same values for conventional Raft clusters with respectively three and five servers is a much easier task because both clusters require a majority of their servers to be available in order to perform both log updates and leader election quorums. Observing that the availability of a single server A is

$$A = \frac{1}{1+\rho},$$

we have

$$\begin{aligned}
A_3(\rho) &= A^3 + 3A^2(1-A) \\
&= \frac{1+3\rho}{(1+\rho)^3} \\
P_{3,2}(\rho) &= 3A^2(1-A) \\
&= \frac{3\rho}{(1+\rho)^3}
\end{aligned}$$

for a Raft cluster with three servers, and

$$\begin{aligned}
A_5(\rho) &= A^5 + 5A^4(1-A) + 10A^3(1-A)^2 \\
&= \frac{1+5\rho+10\rho^2}{(1+\rho)^5} \\
P_{5,2}(\rho) &= 0.
\end{aligned}$$

Fig. 3 displays the availabilities offered by the three configurations for values of the failure rate-to-repair rate ratio ρ. A zero value indicates a server that would never crash and a 0.20 value a server that would be available 83.3 percent of the time. Conversely, a server that would be available 95 percent of the time would have a ρ ratio equal to 0.0526. We immediately note that a Raft cluster with four servers offers a much lower availability than a Raft cluster with five nodes and a barely better availability than a cluster with three nodes. To understand what causes this relatively poor performance, let us return to the state transition diagram

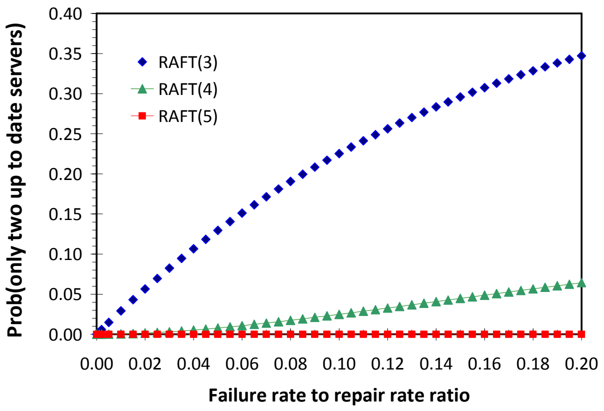

Fig. 4. Compared probabilities of accepting updates when only two participating servers remain available for Raft clusters with three, four and five servers.

displayed in Fig. 2. In contrast to a Raft cluster with five servers that tolerates all double failures, a Raft cluster with four servers will only tolerate the successive failures of any given servers and the leader of the remaining three servers.

A simple way of comparing the log update durabilities offered by the three configurations is to estimate the likelihood that they will perform log updates with fewer than three participating servers. Since none of the three configurations will perform these updates when only one server remains present, our comparison will focus on the likelihood that the three configurations will perform log updates with two participating servers. As Fig. 3 shows, the comparison is much more favorable to the Raft cluster with four servers. It will perform much fewer updates on two servers than a cluster with three servers and very few of them as long as $\rho \leq 0.05$.

We can safely conclude that Raft clusters with four servers present a viable alternative to Raft clusters with five servers whenever the application requires:

1. The same availability guarantees as those offered by a Raft cluster with three servers.
2. The same update durability guarantees as those offered by a Raft cluster with five servers.

V. Raft Clusters with Witnesses

Another option for reducing the energy footprint of the Raft algorithm is to replace some of its servers by lightweight entities that could run on very low-power nodes such as FAWN nodes [1] or the Raspberry Pi [22].

Witnesses [15] allow that option. They are small entities that contain enough metadata to participate in all quorums but hold no data. In our context, it means that a witness will keep track of the sequence number of the current term and the indexes of all log updates, but not their contents. In the same way, it will not have an associated state machine but will keep track of the epoch number and the index of the last known update applied by the leader to its state machine.

An important consequence of these limitations is that witnesses do not hold enough information to act as leaders of a cluster. To enforce this restriction, we will prevent witnesses from calling for an election. Witnesses will still listen to heartbeats from their cluster leader but will not act upon the discovery of a leader failure.

The main advantage of witnesses is that a cluster comprising n conventional servers and m witnesses will provide almost the same cluster availability as a cluster comprising $n + m$ conventional servers as long as the number of conventional servers exceeds the number of witnesses. To understand why these availabilities may differ, let us consider the case of a Raft cluster with four replicas, respectively identified as A, B, C and D, and a single witness W. To simplify our notations, let us assume that the state of each entity can be entirely determined from their current epoch number e and the index of the last request r they committed.

We start by assuming that all cluster entities are operational and up to date:

A	B	C	D	W
e_A= 5	e_B= 5	e_C= 5	e_D= 5	e_W= 5
r_A= 14	r_B= 14	r_C= 14	r_D= 14	r_W= 14

Assume now that servers C and D fail. Since three of the five original entities remain available, the cluster will keep processing log update requests:

A	B	[C]	[D]	W
e_A= 5	e_B= 5	e_C= 5	e_D= 5	e_W= 5
r_A= 22	r_B= 22	r_C= 14	r_D= 14	r_W= 22

Assume now that servers A and B fail and servers C and D recover *after* servers A and B have failed

[A]	[B]	C	D	W
e_A= 5	e_B= 5	e_C= 5	e_D= 5	e_W= 5
r_A= 22	r_B= 22	r_C= 14	r_D= 14	r_W= 22

All attempts to elect a new cluster will fail because witness W cannot serve as the new leader and cannot vote for either server C or server D because it cannot vote for any server whose log is less up to date than its own log record. This would not have happened if W had been a conventional Raft server, because it would have then invited servers C and D to vote for him and they would have accepted. Fortunately for us, this scenario is very rare as long as the participating entities do not fail too often for long periods of time.

One drawback of replacing conventional Raft servers by witnesses is that it allows updates to be applied to fewer servers than before. Returning to the case of our cluster with four conventional servers and a single witness, we observe that valid update quorums can now consist of two servers and one witness while any update quorum for a Raft cluster with five servers will necessarily involve three of the five servers.

Replacing two of the five servers by witnesses has an even more drastic impact on the durability of log updates as a valid update quorum can now consist of the two witnesses and a single replica.

We analyzed the performances of Raft cluster configurations respectively consisting of three servers and two witnesses and four servers and a single witness using the same techniques we used for the case of a Raft cluster with four servers. The necessity to handle situations where the cluster remained unavailable when witnesses were up to date but the only available servers were not up to date resulted in two fairly complex state transition diagrams. For instance, the state transition diagram for a cluster consisting of three conventional Raft servers and two witnesses comprised 38 distinct states. In contrast, the state diagram for a cluster with four replicas and one witness only had 32 distinct states with 11 of these states corresponding to states where two of the five entities are operational. Space considerations prevent us from discussing here these two diagrams. We will instead refer the interested reader to a previous paper where we describe in some detail a simpler configuration consisting of two file replicas and one witness [15].

We avoided the tedious process of computing algebraic solutions of the two systems of equations corresponding to these two configurations by using the Maxima symbolic algebra package [12]. Most of our results were quotients of polynomials of degree 10 to 17 in ρ.

Fig. 5 compares the availabilities offered by the two new configurations with witnesses with those offered by Raft configurations with three or five servers. As we can see, the availabilities offered by the configuration with four servers and one witness (RAFT(4+1)) are practically undistinguishable from those offered by a configuration with five servers. In addition, the availabilities offered by the configuration with three servers and two witnesses (RAFT(3+2)) remain fairly close to these values as long as $\rho \leq 0.05$, which will hold as long as individual servers remain up 95 percent of the time. In other words, replacing one or two of the five servers of a Raft cluster by witnesses will have no significant impact on the service availability as long as its individual components remain operational most of the time.

The same is not true for the durability of the log updates. As Fig. 6 shows, the configuration with four servers and one witness still provides much better guarantees of durability than a configuration with three servers because much fewer log updates will be recorded on two servers. Conversely, the configuration with three servers and two witnesses offer lesser guarantees of durability than a configuration with three servers because an equal fraction of log updates will be recorded on only two of the three servers and, as Fig. 7 shows, a small but significant fraction of log updates will be recorded on a single server.

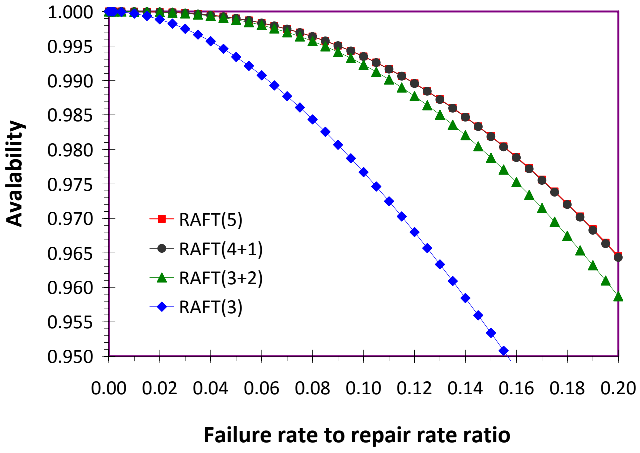

Fig. 5. Compared availabilities of Raft clusters with and without witnesses.

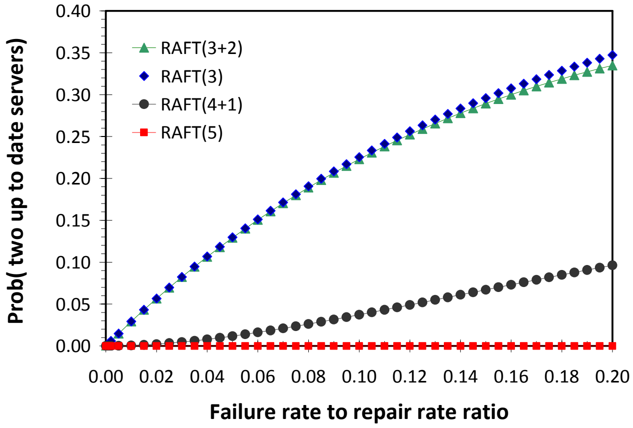

Fig. 6. Compared probabilities of accepting updates when only two servers remain available for Raft clusters with and without witnesses.

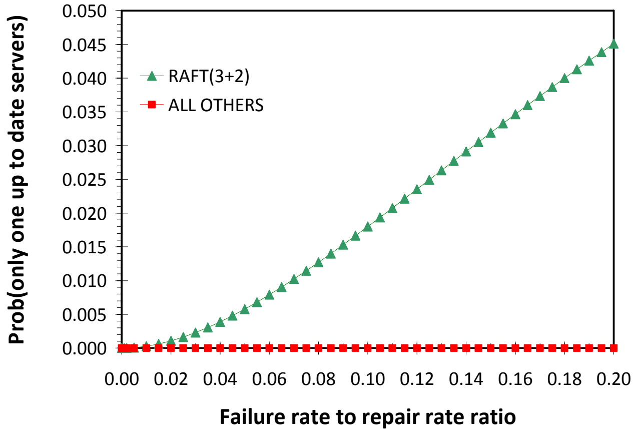

Fig. 7. Compared probabilities of accepting updates when only one server remains available for Raft clusters with and without witnesses.

VI. Conclusion

While the Raft consensus algorithm is both easier to understand and more straightforward to implement than the older Paxos algorithm, it requires five servers to ensure both the availability and the durability of its log updates.

We have proposed two methods for reducing this huge energy footprint. Our first solution consists of adjusting Raft quorums in a way that would allow updates to proceed with as little as two servers while requiring a larger quorum for electing a new leader. Our Markov analysis showed that our solution offered a much better protection against data loss than a Raft cluster with three servers and much less impressive improvements of the service durability.

Our second proposal consists of replacing one or two of the five Raft servers with witnesses, that is, lightweight servers that maintain the same metadata as other servers but hold no data and can therefore run on very low-power hosts. Our Markov analyses showed that these substitutions have little impact on the cluster availability and no significant impact on the risks of incurring a data loss for the configuration consisting of four replicas and a witness and a a much higher risk of data losses for the configuration consisting of three replicas and two witnesses. In other words, teams wishing to implement a highly available version of the Raft protocol should not hesitate to replace one of its four servers by a witness, thus saving nearly 20 percent of its energy footprint. While savings of up to 40 percent could be achieved by replacing two of the five replicas by witnesses, this solution should only be considered when the durability of log updates is less important.

Two potential avenues for further work are replacing Raft static voting protocol by a dynamic voting protocol [5] and allowing failed witnesses to be promptly regenerated on spare sites [16].

References

[1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan and V. Vasudevan, "FAWN: A fast array of wimpy nodes," Proc. 22nd ACM Symposium on Operating System Principles, Big Sky, MT, pp. 1–14, Oct. 2009.

[2] Brian Beach, "How long do disks last?" https://www.backblaze.com/blog/how-long-do-disk-drives-last/, retrieved March 25, 2015.

[3] M. Burrows,"The Chubby lock service for loosely-coupled distributed systems." Proc. 7th Symposium on Operating systems Design and Implementation (OSDI '06), Seattle, WA, pp. 335–350, Nov. 2006.

[4] T. D. Chandra, R. Griesemer, J. Redstone,Paxos made live: an engineering perspective," Proc. 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '07), Portland, OR, pp. 398–407, Aug. 2007.

[5] D. Davcev and W.A. Burkhard, "Consistency and recovery control for replicated files." Proc. 10th ACM Symposium on Operating System Principles, (1985) pp. 87–96.

[6] H. Howard, M. Schwarzkopf, A. Madhavapeddy, J. Crowcroft, "Raft refloated: do we have consensus?" ACM SIGOPS Operating Systems Review: Special Issue on Repeatability and Sharing of Experimental Artifacts, 49(1):12–21 , Jan.2015

[7] L. Lamport, "Time, clocks and the ordering of events in a distributed system," Communications of the ACM, 21(7): 58–65, July 1978.

[8] L. Lamport, R. Shostak, M. Pease, "The Byzantine generals problem," ACM Transactions on Programming Languages and Systems 4(3): 382–401, July 1982.

[9] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems, 6(2):254–280, Apr. 1984.*

[10] L. Lamport, "The part-time parliament," ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

[11] L. Lamport, "Paxos made simple," ACM SIGACT News,32(4):18–25, Dec. 2001.

[12] Maxima, a Computer Algebra System, http://maxima.sourceforge.net/, retrieved March 28, 2015.

[13] D. Ongaro, J. Ousterhout, "In search of an understandable consensus algorithm" (Extended Version). Tech Report. May, 2014. http://ramcloud.stanford.edu/Raft.pdf

[14] D. Ongaro, J. Ousterhout, In search of an understandable consensus algorithm. Proc. 2014 USENIX Annual Technical Conference (ATC '14), Philadelphia, PA, pp. 305–319, June 2014.

[15] J.-F. Pâris, Voting with witnesses: a consistency scheme for replicated files, Proc. 6th International Conference on Distributed Computing Systems (DCS '86), Cambridge, MA, pp. 606–612, May 1986.

[16] C. Pu, J. D. Noe, A. Proudfoot, "Regeneration of replicated objects: A technique and Its Eden implementation," Proc. 2nd International Conference on Data Engineering (ICDE '86), Los Angeles, CA, pp.175–187, Feb. 1986

[17] R. D. Schlichting, F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems," ACM Transactions on Computer Systems 1(3):222–238, Aug. 1983

[18] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial." ACM Computing Surveys, 22(4):299–319, Dec. 1990.

[19] J. Seguin, G. Sergeant, and P. Wilms, "A majority consensus algorithm for the consistency of duplicated and distributed information," Proc. First International Conference on Distributed Computing Systems, Huntsville, AL, pp. 617–624, Oct. 1979.

[20] M. W. Storer, K. Greenan, E. L. Miller, K. Voruganti, "Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage," Proc. 6th USENIX Conference on File and Storage Technologies (FAST 2008),San Jose, CA, pp. 1–6, Feb. 2008.

[21] R. H. Thomas, "A majority consensus approach to concurrency control, ACM Transactions on Database Systems," 4(2):180–209, June 1979.

[22] E. Upton, G. Halfacree, Raspberry Pi User Guide, Wiley, Sep. 2014