Pirogue, a lighter dynamic version of the Raft 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—Raft is a new distributed consensus algorithm that is easier to understand than the older Paxos algorithm. Raft's major drawback is its high energy footprint: as it relies on static quorums for deciding when it can commit updates, it requires five participants to protect against two simultaneous failures. We propose to reduce this footprint by replacing the static quorums that Raft currently uses by quorums that vary according to the number of currently available participants. We present first a modified dynamic-linear voting protocol that disables single-server updates and show that a Raft cluster with four participants managed by this protocol would be almost as available as a conventional Raft cluster with five participants and always tolerate the irrecoverable failure of any single participant without any data loss. In addition, we show a Raft cluster with three participants and a witness managed by an unmodified dynamic-linear voting protocol would be more available than a conventional Raft cluster with five participants and could still tolerate most irrecoverable failures of any single participant while maintaining recoverability.

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

I. Introduction

Distributed consensus algorithms allow multiple participants in a distributed system to agree on the values of the data they share. 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 communication failures of all kinds.

Since it was proposed in the late nineties, Leslie Lamport's Paxos algorithm [8, 9] has been the gold standard for distributed consensus algorithms because it has been proven correct and is efficient in the standard case. At the same time, it remains very hard to understand and quite difficult to implement [12]. To quote the authors of a full size implementation of Paxos, "[t]here are significant gaps between the description of the Paxos algorithm and the needs of a real-world system." [3]

Ongaro and Ousterhout recently proposed the Raft consensus algorithm to overcome these limitations [11, 12]. As noted by Howard et al. [5], Raft is easier to understand and easier to implement than Paxos. A remaining limitation of Raft is the large number of servers it requires. Because it uses majority consensus voting to decide 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***. This requirement is likely to remain acceptable as long as Raft supports lightweight services that can run on low-power nodes, thus maintaining a low energy footprint. This is much less true when applications require full-fledged servers.

We believe that reducing the energy footprint of the RAFT protocol is essential to ensure its continuous success. We addressed the issue in a previous paper [14] and proposed two approaches for reducing the number of servers involved in a Raft cluster without significantly altering the Raft algorithm [14]. Our first proposal consisted 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. For instance, 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 cluster with three nodes. Our second proposal consists of replacing some Raft servers with *witnesses*, that is, lightweight servers that maintain the same metadata as their peers but hold no data [13]. We found that clusters having one or two of their servers replaced by these witnesses provided similar cluster availabilities as a cluster with five full servers and very good to adequate protection against data loss depending on the number of servers replaced by witnesses.

The approach we propose here is more ambitious. Pirogue replaces the static voting quorums that Raft uses by dynamic quorums that shrink when the number of active servers diminishes and grow when this number increases. As a result, each Pirogue cluster will have to maintain a consistent view of the number and the identities of all its active servers. We found out that this additional complexity produced much higher service availabilities than our previous proposals:

[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.

978-1-4799-7575-4/14/$31.00 ©2015 IEEE

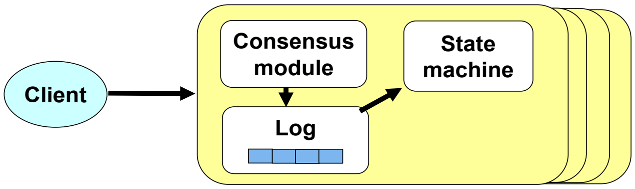

Fig. 1. The Raft architecture (after [12]).

- A Pirogue cluster with four servers managed by a dynamic linear voting protocol that does not allow single-server updates would provide nearly the same service availability as a Raft cluster with five servers and still guarantee that a single irrecoverable server failure would never result in a data loss.
- A Pirogue cluster with four servers and no restrictions on the size of its quorums would provide a better service availability than a Raft cluster with five servers.
- We can even replace one of the four servers in the above cluster by a witness and still provide better service availability than a Raft cluster with five servers.

The remainder of this paper is organized as follows. Section II reviews the relevant features of the Raft algorithm. Section III introduces our proposal and Section IV evaluates the performance of three Pirogue configurations in terms of system availability and risk of data losses. Finally, Section V 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 [16].

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 empowered 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 term and forwards it to its followers through an AcceptEntry() 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 term. Commit decisions are propagated to other servers by including the index of the last committed update in all future messages coming from the leader, including AcceptEntry() 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 for 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 to all other servers a message containing a summary of the state of its log. 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 a 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. Introducing Pirogue

The main reason for the high energy footprint of the Raft algorithm is its reliance on majority consensus voting [17, 18] for defining both its write quorum and its election quorum. Since these quorums must contain a majority of the cluster servers, a Raft cluster must comprise at least $2n + 1$ servers to be able to tolerate n server failures.

A much more efficient solution is to make these quorums dynamic and require them to contain a majority of the active servers of a given cluster, thus excluding servers that have crashed and have not been formally reintegrated. So if a cluster contains four servers, its initial majority will be defined as three servers out of four. Should one of the servers fail, the new majority will be two out of three. After a second server failure, it will be two out of two. What happens after a third failure depends on the way the protocol handles ties. The original *dynamic voting* (DV) protocol [4] did not assign weights to servers and did not include a tie-breaking rule. As a result, it required a minimum of two communicating servers to allow updates. A simple extension, known as *dynamic-linear voting* (DLV) [6], solves these ties by applying a total ordering to the servers. Overall, both DV and DLV require $n + 2$ servers per cluster to be able to tolerate n failures. Pirogue takes advantage of this property to reduce from five to four, the number of servers required to tolerate two failures, thus resulting in a 25 percent reduction of the energy footprint of Raft.

Since Pirogue relies on dynamic quorums to decide when to commit updates and to elect new leaders, it will have to maintain additional dynamic metadata. The simplest solution is to use *cohort sets* [10] and let Pirogue manage them on the top of the Raft update protocol.

A. Cohort sets

By definition, the cohort set of a Pirogue cluster represents the set of servers that are allowed to participate in a leader election. This set is customarily stored in a bitmap. Updating that bitmap is the responsibility of the leader of the cluster. It will do so whenever it detects (a) that a server has ceased to reply to an update request and (b) the recovery of a server that was previously unavailable.

Bitmap updates are pushed to other servers in the same way as regular updates: the server assigns to each update a sequence number within the present term and forwards it to its followers through an `AcceptEntry()` remote procedure call. All followers that have up-to-date logs append the new update to their logs and notify the leader of that fact. As soon as the leader has replicated the update on a majority of the servers in the *current cohort set*, it *commits* it, applies it to its own state machine and notifies its followers.

The leader will normally detect follower failures and recoveries by keeping track of which followers acknowledge its `AcceptEntry()` requests. Whenever user-generated update requests are less than frequent, this approach may result in unacceptable delays between the time a server fails and the time the cluster leader detects that failure. The simplest solution to this problem is to let the leader generate a dummy `AcceptEntry()` request whenever a fixed interval has lapsed without any user-generated update requests.

B. Handling leader failures

Under Pirogue, only servers that belong to the last committed cohort set are allowed to have their vote counted. Let us see how Pirogue achieves this goal.

1. Any candidate for the cluster leadership position will include its version of the cluster cohort set in the message it sends to other servers.
2. As in the Raft election protocol, servers receiving a vote solicitation will withhold their vote if any of the three following conditions holds:
   a) They believe they still have a leader,
   b) They have already voted for another candidate, or
   c) Their own log is more "up to date" than the candidate's log.
3. Servers voting for a candidate will attach to their votes their version of the cluster cohort set.
4. When a candidate tallies the votes it has received, it compares their versions of the cluster cohort set. In doing so, it ascertains the current value of that set and disregards the votes of all servers that are not in the current cluster cohort set when deciding the election outcome.

Note that all servers in the current cluster cohort do not necessarily share the same view of that set. Consider, for instance a Pirogue cluster with four servers, respectively named A, B, C and D, and let us assume that server A is the current leader of the cluster. When all four servers are operational, the cluster cohort set is $\{A, B, C, D\}$:

A	B	C	D
$\{A, B, C, D\}$	$\{A, B, C, D\}$	$\{A, B, C, D\}$	$\{A, B, C, D\}$

and all four servers are in perfect agreement about that value.

After cluster D fails, the new cohort set of the cluster will be $\{A, B, C\}$:

A	*B*	*C*	[*D*]
{*A*, *B*, *C*}	{*A*, *B*, *C*}	{*A*, *B*, *C*}	{*A, B, C, D*}

Assume now that server *D* has recovered and that server *A* has added it to its version of the cohort set but has crashed before having propagated its new cohort set to either of its followers:

A	*B*	*C*	*D*
{*A*, *B*, *C*. *D*}	{*A*, *B*, *C*}	{*A*, *B*, *C*}	{*A*, *B*, *C*, *D*}

Since A failed to propagate the new value of the cohort set to a majority of the servers in the current cohort set, that current cohort set remains valid and server *A* remains in it.

C. *Reliability Issues*

Because Raft relies on majority consensus voting, a Raft cluster with five servers will protect the cluster and the data it holds against:

1. The simultaneous failure of two of its five servers, and
2. The irrecoverable failure of two servers.

This is not true for Pirogue clusters as they will occasionally run with only one or two operational servers. As a result, Pirogue only needs four servers to protect the service against the simultaneous failure of two of them. The drawback is that Pirogue will not always preserve the cluster state in the presence of two irrecoverable server failures. Depending on the way the cluster voting is set up, a Pirogue cluster may occasionally run with only one or two operational servers. As a result, there will be relatively brief intervals during which the cluster state may either be left unprotected or only protected against a single irrecoverable server failure. This is to say that any given Pirogue cluster will be able to tolerate more simultaneous server failures than a Raft cluster with the same number of servers, but will also offer less protection against irrecoverable server failures.

We do not believe it to be a major issue because most irrecoverable data losses result from irrecoverable disk failures and these failures are likely to be much less frequent than server crashes. Even assuming a disk failure rate of 11.8 percent per year, which is typical for consumer disks at the very end of their useful lifetime [2], disk mean times to failure would remain close to eight years and a half. We can expect most disks to be more reliable than that and solid-state devices to be even more reliable. When this is insufficient, Pirogue offers its users the option of requiring all updates to be propagated to all least two servers.

IV. Realizations

In this section, we evaluate three realizations of the Pirogue algorithm that correspond to three different tradeoffs between cluster availability, the risk of data loss and the energy footprint of the cluster. These three realizations are:

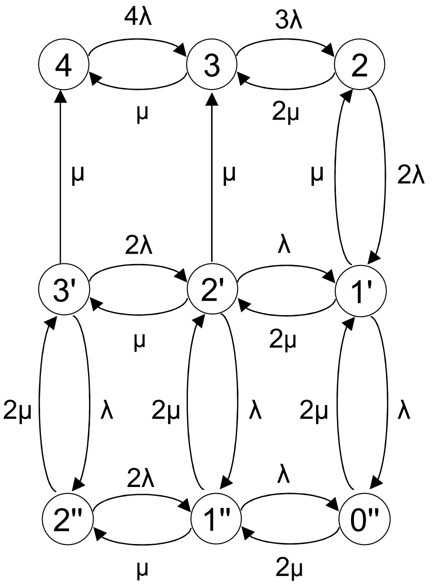

Fig. 2. A Pirogue cluster with four servers, that does not allow updates in single-server mode.

1. A four-server Pirogue cluster that requires all updates to be propagated to at least two servers;
2. A four-server Pirogue cluster that allows updates even when only a single server remains operational; and
3. A Pirogue cluster with three servers and one witness replacing the fourth server.

We will use three performance indexes to evaluate these three realizations:

1. The cluster *availability*, that is, the fraction of time it will be able to process user requests;
2. The cluster *exposure* to double irrecoverable server failures, that is, the fraction of time the cluster will run on two operational servers; and
3. The cluster *exposure* to single irrecoverable server failures, that is, the fraction of time the cluster will run on a single server.

To derive these values, we need to make a few working 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 the same way, we require repair times 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.

Even though it is not necessary, we will assume that cluster leaders can quickly detect server failures and repairs, possibly with the help of dummy queries as we sketched in Section III.

A. A four-server cluster prohibiting single-server updates

We will consider first a four-server Pirogue cluster that requires all updates to be propagated to at least two servers in order to protect the cluster state against single irrecoverable server failures.

The behavior of our cluster can be described by its state transition diagram. As Fig. 2 shows, the cluster has nine possible states. State <4> is the original state where all four servers are available. In that state the cluster quorums are three out of four. A failure of one of the servers will bring the cluster to state <3> and changes the cluster quorums to two out of three. A second failure would take the cluster to state <2> and the cluster quorums to two out of two. As a result, a cluster in state <2> will only have two up-to-date servers.

A third failure will move the cluster to state <1'>, which is an unavailable state since no possible quorum can be formed. We observe three distinct transitions from state <1'>:

1. A failure from the last server will bring the cluster to state <0">.
2. A recovery of the other up-to-date server l will bring the cluster back to state <2> and allow the cluster to process again client requests.
3. A recovery of either of the two other servers will bring the cluster to state <2'> and leave the server unable to process client requests.

All primed states are states where the cluster is unavailable even though they contain one operational server that has the most current version of the log. We see two possible recovery transitions from state <0">.

1. A recovery of either of the last two up-to-date servers will return the cluster to state <1'>.
2. A recovery of either of the other two servers will bring the cluster to state <1">.

All double primed states, such as state <1"> and state <0"> are unavailable states like all the primed states. Primed and double primed states differ in the composition of their sets of operational servers. All transitions from a double primed state to a primed state correspond to the recovery of one of the two up-to-date servers of the cluster. Similarly, all transitions from a primed state to a non-primed state correspond to the recovery of the other up-to-date server of the cluster and allow it to process again user requests. There is no state <3"> as any grouping of three recovering servers will necessarily contain one of the two up-to-date servers.

The equilibrium conditions for our system are:

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

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

$$\textstyle\sum_i p_i = 1.$$

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

1. The availability of the cluster $A_{4R}(\rho)$
$$\begin{aligned}A_{4R}(\rho) &= p_4 + p_3 + p_2\\ &= \frac{3\rho^6+23\rho^5+68\rho^4+97\rho^3+68\rho^2+21\rho+3}{(\rho+1)^5(3\rho^3+8\rho^2+6\rho+3)},\end{aligned}$$
2. The cluster exposure to double irrecoverable server failures $E_{4R,2}(\rho)$
$$E_{4R,2}(\rho) = p_2 = \frac{3\rho^6+17\rho^5+39\rho^4+42\rho^3+18\rho^2}{(\rho+1)^5(3\rho^3+8\rho^2+6\rho+3)}.$$

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 elections. Observing that the availability of a single server A is

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

we have

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

for a Raft cluster with three servers, and

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

for a Raft cluster with five replicas.

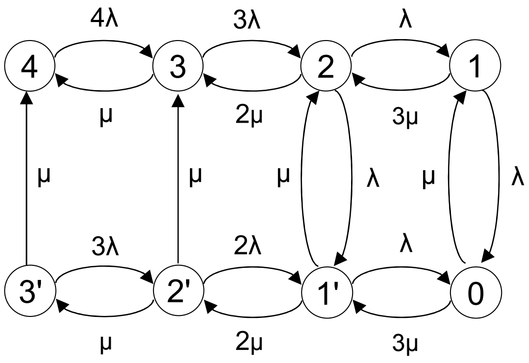

Fig. 3. A Pirogue cluster with four servers, that allows updates in single-server mode.

B. A four-server cluster allowing single-server updates

Let us now consider what would happen if we removed the requirement that all updates must be propagated to at least two servers and allowed the cluster to operate when a single server remains operational.

As Fig. 3 shows, the cluster has eight possible states. State <4> is the original state where all four servers are available. In that state the cluster quorums are three out of four. A failure of one of the servers will bring the cluster to state <3> and changes the cluster quorums to two out of three. A second failure would take the cluster to state <2> and the cluster quorums to two out of two. As a result, a cluster in state <2> will only have two up-to-date servers.

A third failure will create a tie. Dynamic-linear voting will break it by comparing the linear ordering of the server that failed with that of the server that remained operational. As a result, we will observe two failure transitions from state <2>, namely:

1. A failure transition to state <1>, which corresponds to a failure of the server that *follows* the surviving server in the linear ordering of all servers and leaves the cluster in state <1>, still able to process client requests. The cluster quorums are now one out of one.
2. A failure transition to state <1'>, which corresponds to a failure of the server that *precedes* the surviving server in the linear ordering of all servers and leaves the cluster in state <1'>, unable to process client requests.

Note that all primed states are states where the cluster is unavailable because they correspond to cluster configurations where the set of operational servers do not contain the server that has or could have the sole up-to-date version of the log.

Both states <1> and <1'> have identical failure transitions to state <0'> but their recovery transitions are quite different. When the cluster is in state <1>, a recovery of any of the three failed servers will always bring the cluster to state <2> while state <1'> has two possible recovery transitions, namely:

1. A recovery transition to state <2>, which corresponds to the recovery of the server that has or could have the sole up-to-date version of the log.
2. A recovery transition to state <2'>, which corresponds to the recovery any of the three other servers.

Once the system is in a primed state it has to wait for the recovery of the server that has or could have the sole up-to-date version of the log in order to be again able to process client requests. For instance, state <3'> is an unavailable state even though three of the four servers of the cluster are operational

The equilibrium conditions for our system are:

$$
\begin{aligned}
4\lambda p_4 &= \mu(p_3 + p_{3'}),\\
(3\lambda+\mu)p_3 &= 4\lambda p_4 + 2\mu p_2 + \mu p_{2'},\\
(3\lambda+\mu)p_{3'} &= \mu p_{2'},\\
(2\lambda+2\mu)p_2 &= 3\lambda p_3 + 3\mu p_1 + \mu p_{1'},\\
(2\lambda+2\mu)p_{2'} &= 3\lambda p_{3'} + 2\mu p_{1'},\\
(\lambda+3\mu)p_1 &= \lambda p_2 + \mu p_{0'},\\
(\lambda+3\mu)p_{1'} &= \lambda p_2 + 2\lambda p_{2'} + 3\mu p_{0'},\\
4\mu p_{0'} &= \lambda p_1 + \lambda p_1',
\end{aligned}
$$

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

$$\sum_i p_i = 1.$$

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

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

$$
\begin{aligned}
A_4(\rho) &= p_4 + p_3 + p_2 + p_1\\
&= \frac{6\rho^6+35\rho^5+102\rho^4+152\rho^3+113\rho^2+39\rho+6}{(\rho+1)^4(\rho^3+17\rho^2+15\rho+6)},
\end{aligned}
$$

2. The cluster exposure to double irrecoverable server failures $E_{4,2}(\rho)$

$$E_{4,2}(\rho) = p_2 = \frac{18\rho^4+42\rho^3+36\rho^2}{(\rho+1)^3(6\rho^3+17\rho_2+15\rho+6)}.$$

3. The cluster exposure to single irrecoverable server failures $E_{4,1}(\rho)$

$$E_{4,1}(\rho) = p_1 = \frac{6\rho^6+17\rho^5+24\rho^4+12\rho^3}{(\rho+1)^4(6\rho^3+17\rho^2+15\rho+6)}.$$

C. A three-server cluster incorporating a witness

Let us now return to our example and consider the server that is the last in the linear ordering of all four servers. That server will never be allowed to become the single remaining operational server of the cluster because it will never be selected by the tie-breaking rule of the dynamic-linear voting protocol. As a result it could be replaced by a *witness* without affecting the availability of the cluster. Witnesses are small entities that contain enough metadata to participate in all quorums but hold no data [13]. In our context, it means that witnesses will keep track of the sequence number of the current term, cohort set contents, and the indexes of all log updates, but not their contents. In the same way, they will lack an associated state machine but will keep track of the term number and the index of the last known update applied by the leader to its state machine. Since witnesses hold no data, they can run on very low-power nodes such as FAWN nodes [1] or the Raspberry Pi [19], thus reducing the energy footprint of the whole cluster.

The sole drawback of that approach is a higher exposure to single and double irrecoverable server failures because:

- Some configurations that previously consisted of two operational servers will now consist of a single operational server and a witness, and
- Some configurations that previously consisted of three operational servers will now consist of two operational servers and a witness.

A good combinatorial approximation of this impact is to assume that 3/4 of configurations that comprised three operational servers will now comprise two operational servers and a witness while 1/2 of those that comprised two operational clusters will now comprise a single operational server and a witness. As a result:

1. The cluster exposure to double irrecoverable server failures $E_{3+1,2}(\rho)$:

$$E_{3+1,2}(\rho) = 3p_3/4 + p_2/2,$$

2. The cluster exposure to single irrecoverable server failures $E_{3+1,1}(\rho)$:

$$E_{3+1,1}(\rho) = p_2/2 + p_1.$$

where the p_i's are the same as those obtained for the configuration with four servers.

D. Comparing the three realizations

Fig. 4 compares the availabilities offered by the three Pirogue realizations with those offered by Raft configurations with three or five servers for values of the failure rate-to-repair rate ratio ρ between zero and 0.20. 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.

As we can see, the availabilities offered by the Pirogue configurations with four servers and three servers and a witness (PIROGUE(4)) and (PIROGUE(3+1)) provide slightly higher cluster availabilities than the original Raft protocol with five servers (RAFT(5)). Conversely, the configuration with four servers that disallows single-server updates (RESTRICTED PIROGUE(4)) performs slightly worse.

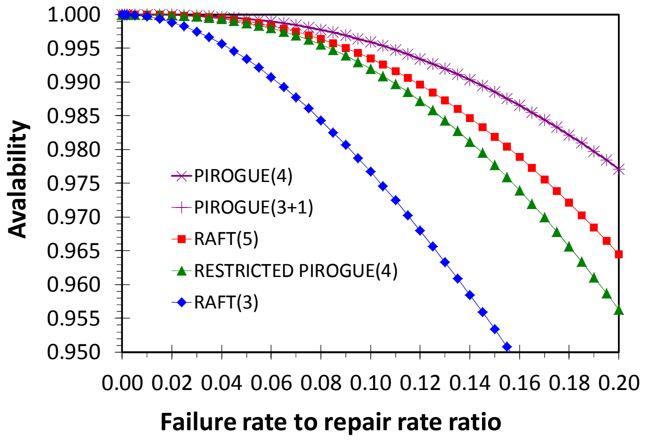

Fig. 4. Compared availabilities of the three Pirogue configurations.

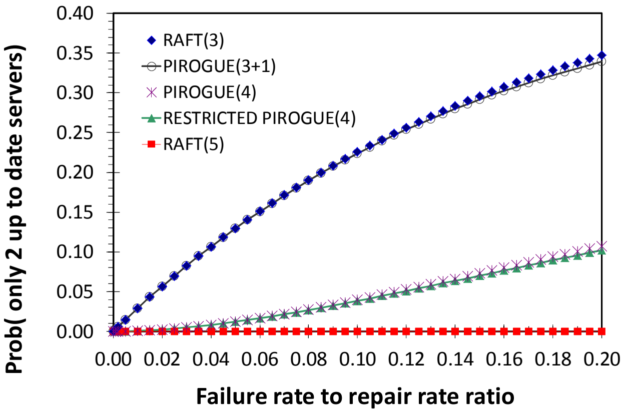

Fig. 5. Compared exposures to double irrecoverable server failures of the three Pirogue configurations.

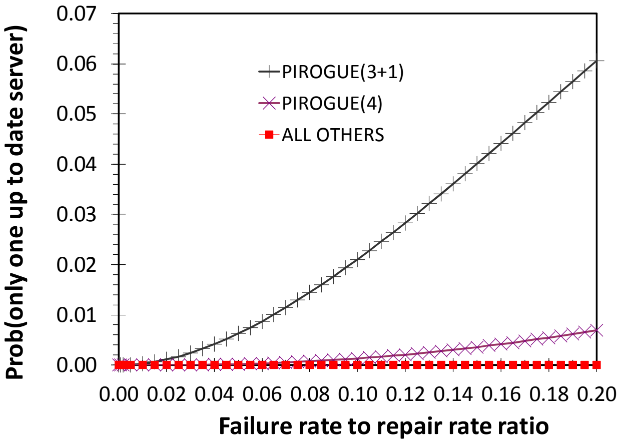

Fig. 6. Compared exposures to single irrecoverable server failures of the three Pirogue configurations.

Given a typical crash rate of at most one failure per month and a typical repair time of 24 hours, the λ/μ ratio of a typical cluster should rarely exceed 0.033. At that rate, the availabilities of all three Pirogue configurations are indistinguishable from that of a Raft cluster with five servers.

The same is not true for the durability of the log updates. As Fig. 5 shows, the Pirogue configuration with three servers and one witness has the same exposure to double irrecoverable server failures as a conventional Raft configuration with three servers (RAFT(3)). Conversely this exposure remains quite limited for Pirogue configurations with four servers.

These trends are confirmed when we consider exposure to single irrecoverable server failures. As Fig. 6 shows, the Pirogue configuration with three servers and one witness has become significantly vulnerable to these failures when the failure rate to repair rate ratio ρ exceeds 3 to 5 percent. Conversely the Pirogue configuration with four servers that allows log updates in single-server mode remains much less exposed as long as system crashes are not frequent. Both the Pirogue configuration that prohibits single-user updates and the original Raft configuration have a zero exposure because they do not allow log updates when a single server is operational.

V. 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 shown how this footprint could be reduced by replacing the static quorums that Raft currently uses by quorums that vary according the number of currently available participants. We have presented first a Pirogue configuration with four servers that disallows single-user updates and shown that this configuration would be nearly as available as a conventional Raft cluster with five participants and would always tolerate the irrecoverable failure of any single participant without any data loss. In addition, we have shown that a Pirogue cluster with three servers and a witness managed by an unmodified dynamic-linear voting protocol would be more available than a conventional Raft cluster with five participants and could still tolerate most irrecoverable failure of any single participant without any data loss.

At typical server and crash rates, the three Pirogue configurations that we have presented offer availabilities that are indistinguishable from that of a conventional Raft cluster with five servers even though these Pirogue configurations require either four servers or three servers and a lightweight witness. In other words, switching from Raft to Pirogue could result in power savings between 20 and 40 percent.

A potential avenue for further work is allowing failed witnesses to be promptly regenerated on spare sites [15].

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] 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.

[4] 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.

[5] 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

[6] S. Jajodia and D. Mutchler, Dynamic voting algorithms for maintaining the consistency of a replicated database, *ACM Trans. on Database Systems,* 15(2):230–280, June 1990.

[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. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

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

[10] D. D. E. Long and J.-F. Pâris, Voting without version numbers, Proc. 1997 International Phoenix Conference on Computer and Communication (IPCCC'97), Phoenix, AZ, pp. 139-145, Feb. 1997.

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

[12] 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.

[13] 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.

[14] J.-F. Pâris and D. D. E. Long, Reducing the Energy Footprint of a Distributed Consensus Algorithm, *Proc.*11th *European Dependable Computing Conference* (EDCC 2015), Paris, France, Sep. 2015,to appear.

[15] 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

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

[17] 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.

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

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