Unbounded Pipelining in Dynamically Reconfigurable Paxos Clusters

David C. Turner

***Abstract*—Consensus is an essential ingredient of a fault-tolerant distributed system systems. When equipped with a consensus algorithm a distributed system can act as a replicated state machine (RSM), duplicating its state across a cluster of redundant components to avoid the failure of any single component leading to a system-wide failure. Paxos and Raft are examples of algorithms for achieving distributed consensus. Practical implementations of this kind of system must support dynamic reconfiguration in order to be able to replace failed components and perform other administrative tasks without downtime. Paxos can achieve high performance by pipelining (starting work on new requests before existing requests have completed) but typically bounds the length of the pipeline to ensure consistency during reconfiguration. Raft also supports pipelining and imposes no such bound on concurrent requests, preserving consistency instead by restricting which reconfigurations may be performed. This article shows how to extend Paxos to support a more general form of reconfiguration which subsumes the original bounded-pipeline approach as well as Raft-like fully-concurrent reconfigurations and more besides.**

***Index Terms*—Distributed algorithms, fault tolerance.**

I. Introduction

RELIABLE distributed systems must be able to tolerate a fault in any individual component without suffering a system-wide failure, and typically achieve this by ensuring that there is redundancy between the components. A replicated state machine (RSM) is a style of fault-tolerant distributed system in which a deterministic state machine is replicated across a set of distinct nodes [1]. Being deterministic, the nodes' states remain synchronised if they all start in the same state and perform the same sequence of transitions.

In order to arrange for each node to perform the same transitions the system may achieve consensus on (or *choose*) a sequence of values which describe the transitions. The sequence must be consistent across the whole system even in the presence of failures, and as long as there are not too many failures it must remain possible to continue to make progress. This is known as the distributed consensus problem, for which a number of solutions are known to exist, including Paxos [2] and Raft [3]. They typically run on a cluster of $2f + 1$ nodes, where f is the number of faulty nodes that should be tolerated, and consensus is achieved when a nonempty *quorum* of nodes (e.g. at least $f + 1$ of them) agree. The collection of quorums in use is known as the *configuration* of the cluster.

It is normally necessary to be able to dynamically reconfigure a cluster by adding or removing nodes while it is running, in order that parts of the system can be repaired or replaced without needing to take the whole system offline. It is crucial that all participating nodes agree on the cluster configuration, and this can be achieved by holding the configuration within the RSM itself and using the consensus algorithm to choose special reconfiguration commands when a configuration change is desired.

In Paxos, each value is chosen using a conceptually-separate instance of a two-phase consensus protocol known as *Synod*. The full Paxos algorithm essentially runs an infinite sequence of Synod instances in parallel, using uniformity of the instances to do so without requiring infinite time or resources. It starts by running phase I of all instances at once and then runs phase II of each instance in turn to yield the desired sequence of chosen values. It normally continues to run phase II for extended periods of time, but will return to phase I if certain nodes become faulty, or if messages between certain pairs of nodes cease to be delivered reliably for a period, or if the configuration changes. Raft's pattern of execution is similar.

Both algorithms can achieve high throughput by allowing for *pipelining* [4] whereby work may begin on an instance even before all previous instances have fully completed. It is a little tricky to ensure that this preserves consistency when the configuration is held in the RSM itself because a value may only be proposed once a quorum of nodes are ready for it, but there may be a configuration change in the pipeline which would change the quorums so as to make a proposal invalid. If this case is not handled carefully then it may lead to inconsistency. Paxos implementations typically solve this problem by limiting the length of the pipeline to some $\alpha > 0$ and requiring that a configuration change chosen at instance i does not take effect until at least instance $i + \alpha$, which means that when a proposal is made there can be no as-yet-undecided configuration change in the pipeline that could affect it. In contrast, Raft imposes no limit on the number of concurrently-running instances and instead restricts the reconfigurations that may occur to only allow ones that cannot result in inconsistency. An operator may then perform a sequence of these restricted reconfigurations in order to achieve an arbitrary reconfiguration.

In Paxos, the choice of the pipeline length parameter α must be made carefully. If it is too small then the system may suffer from poor performance due to lack of parallelism, but if it is too large then configuration changes can be unreasonably expensive to complete. It is more elegant [5] to limit the pipeline length only during reconfiguration and to allow the limit to vary while the system is running, but it would be better still if there were no need for a limit at all, even during reconfiguration.

Manuscript received 14 November 2016.
The author is with the Operations and Planning Systems division of Tracsis plc, Leeds LS2 9DF, United Kingdom (email: dct25-a1vwi@mythic-beasts.com)

Here it is shown that the pipeline may indeed safely remain unbounded even during a configuration change as long as the reconfiguration satisfies certain conditions described in section IV-A below. It is also shown that if these conditions are not satisfied then a reconfiguration can still take place as long as the pipeline length is temporarily limited.

The algorithm is presented here in its entirety for the sake of consistency of notation and because it modifies the original algorithm in ways that invalidate its consistency and liveness proofs. We begin with a recap of the Synod algorithm in section III and follow this by covering the full Paxos algorithm in section IV, then in section V it is shown how this work generalises and unifies the previously-known reconfiguration processes supported in Paxos and Raft. Reworked proofs of liveness and consistency are included in section IV-C and appendices B and C and differences from the original are highlighted throughout. The appendices are informal versions of formal proofs performed using the Isabelle/HOL proof assistant [6].

II. Related Work

Lamport's original presentation of Paxos [2] introduced the bounded-pipeline technique for supporting reconfiguration in which the pipeline length was defined to be 3. He later clarified that the value '3' was intended to stand for an arbitrary $\alpha > 0$ in [7]. Later still Lamport and Massa [8] drew a distinction between Static Paxos in which the configuration may not change and Dynamic Paxos which supports reconfiguration but requires a bounded pipeline. Dynamic Paxos was used as a basis for Cheap Paxos in which the system automatically reconfigures itself to achieve higher resilience to failures that do not occur simultaneously, and which uses a heterogeneous set of nodes to reduce the costs of operating a cluster.

Malkhi, Lamport and Zhou [9] proposed Stoppable Paxos, an alternative method for reconfiguring clusters in which an RSM may be stopped, reconfigured, and then restarted with the new configuration. Stoppable Paxos improves on Dynamic Paxos by removing the need for a pipeline limit when the configuration is static and by allowing a different limit to be selected for different reconfigurations.

Malkhi, Lamport and Zhou [10] then proposed Vertical Paxos which keeps the RSM running throughout a reconfiguration, but requires a separate oracle to manage the configuration. The Egalitarian Paxos of Moraru, Andersen and Kaminsky [11] uses a similar reconfiguration scheme. The problem with needing an external oracle is that for full resilience it must be possible to reconfigure the oracle itself, which requires another oracle and so on *ad infinitum*.

Chandra, Griesemer and Redstone [12] noted that the details of reconfiguration are "relatively minor" but "subtle" and do not give any details on the reconfiguration scheme used in their Chubby system. It seems likely that they also used the bounded-pipeline approach. In contrast, Birman, Malkhi and van Renesse [13] noted that allowing for $\alpha > 1$ may require an unacceptably complex implementation, leading many real-world systems to disable pipelining by setting $\alpha = 1$, or even to disable reconfiguration entirely.

Bortnikov et al. [14] implemented a reconfigurable RSM using static RSMs which may pass responsibility to each other in a manner that is similar to that of Stoppable Paxos. They improve the performance of their system compared with Stoppable Paxos by allowing each RSM to speculatively start executing commands before the transfer of responsibility has been fully agreed.

Ongaro and Ousterhout [3] developed the Raft protocol which supports an unbounded pipeline throughout the reconfiguration process without inconsistency by instead limiting the reconfigurations that can be performed. They performed a formal proof of the correctness of Raft without reconfiguration, and an informal argument that consistency is preserved when a single node is added or removed from the cluster. Raft uses simple majorities of the set of nodes as its quorums.

Viewstamped Replication and Zab are two other well-known consensus protocols. Viewstamped Replication supports reconfiguration as described by Liskov and Cowling [15] in a similar fashion to Vertical Paxos. Reed and Junqueira [16] initially presented Zab without reconfiguration and this feature was subsequently added by Shraer, Reed, Malkhi and Junqueira [17], using a limited-pipeline approach much as in Dynamic Paxos.

An interesting alternative approach to reconfiguration was proposed by Jahl and Meling [18] which uses eventual consistency rather than consensus to determine the configuration of the system, and therefore supports reconfiguration as long as it is still possible to communicate with a quorum of nodes even if consensus cannot be achieved due to an inability to elect a distinguished leader. In such a situation consensus-based approaches such as the one presented here would fail to make any further progress, whereas one based on eventual consistency could be reconfigured into one in which a leader may be elected and thus in which further progress can be made.

III. The Synod Algorithm

Synod [2] is an algorithm for achieving consensus on a single value in a distributed system comprising a set of nodes which can communicate by sending messages to each other. The system is asynchronous but not Byzantine, in the sense that messages may be delayed, reordered, duplicated and dropped but not corrupted, and processes may run arbitrarily slowly or even stop but may not deviate from their specifications.

Let $\mathbb{B}$ be a set of *ballot identifiers* with a wellfounded total order $\prec$. Let $\mathbb{A}$ be a set of *node identifiers* and let $\mathbb{V}$ be the set of values that may be chosen.

The Synod algorithm involves five kinds of message, in two phases, as described below. Throughout, $a \in \mathbb{A}$ and $b, b' \in \mathbb{B}$. Phase I starts with the broadcast of a *prepare* message $\mathbf{prepare}(b)$ to which each node may respond with a *promise*, either a *free promise*, written $\mathbf{promised}(a, b)$, or a *forced promise*, written $\mathbf{promised}(a, b; b')$, where a identifies the responding node. Phase II starts with the broadcast of a *proposal*, written $\mathbf{proposed}(b)$, to which each node a may respond with an *acceptance*, written $\mathbf{accepted}(a, b)$. Once

Fig. 1. Invariants preserved by the Synod algorithm

S1) For all $b_1 \succ b_2 \in \mathbb{B}$ there are sets of quorums $Q^{\mathrm{I}}(b_1)$ and $Q^{\mathrm{II}}(b_2) \subseteq \mathcal{P}\mathbb{A}$ such that if $\mathbf{proposed}(b_1)$ and $\mathbf{chosen}(b_2)$ then $Q^{\mathrm{I}}(b_1) \frown Q^{\mathrm{II}}(b_2)$.
S2) If $\mathbf{promised}(a, b)$ then $\neg\mathbf{accepted}(a, b')$ for all $b' \prec b$.
S3) If $\mathbf{promised}(a, b; b')$ then $b' \prec b$, $\mathbf{accepted}(a, b')$, and b' is the greatest such ballot in the sense that $\neg\mathbf{accepted}(a, b'')$ for all b'' having $b' \prec b'' \prec b$.
S4) If $\mathbf{proposed}(b)$ then there is a quorum $q^{\mathrm{I}} \in Q^{\mathrm{I}}(b)$ such that for every node $a \in q^{\mathrm{I}}$ either $\mathbf{promised}(a, b)$ or else there exists a b' such that $\mathbf{promised}(a, b; b')$; if also $P \triangleq \{b' \mid \exists a \in q^{\mathrm{I}}.\mathbf{promised}(a, b; b')\} \neq \varnothing$ then $v(b) = v\big(\max(P)\big)$.
S5) If $\mathbf{accepted}(a, b)$ then $\mathbf{proposed}(b)$.
S6) If $\mathbf{chosen}(b)$ then there is a quorum $q^{\mathrm{II}} \in Q^{\mathrm{II}}(b)$ such that $\mathbf{accepted}(a, b)$ for every $a \in q^{\mathrm{II}}$.

phase II is complete a *success* message, written $\mathbf{chosen}(b)$, is broadcast. It is convenient also to use these symbols as predicates indicating whether the corresponding messages have been sent.

There is a function $v : \mathbb{B} \to \mathbb{V}$ assigning a value to each ballot, discussed in more detail in section III-A below.

The system satisfies a set of invariants listed in fig. 1, from which it follows that consistency is guaranteed in the sense that if $\mathbf{chosen}(b)$ and $\mathbf{chosen}(b')$ then $v(b) = v(b')$ as shown by theorem 8 in appendix B.

Each node operates as a state machine whose transitions are caused by the receipts of messages. Each phase is considered to be complete for a particular ballot when appropriate messages have been received from sufficiently many nodes, where "sufficiently many" is defined in terms of sets of quorums of nodes in the system's configuration.

In more detail, a node may emit $\mathbf{proposed}(b)$ when it considers phase I to be complete at ballot b, which is when promises for b have been received from a quorum of nodes, and similarly may emit $\mathbf{chosen}(b)$ when it considers phase II to be complete at b, which is when acceptances of b have been received from a quorum of nodes.

The phase-I and phase-II quorums are defined so as to always contain at least one node in common, but may vary depending on the ballot b and the phase as discussed in section III-B below. In particular, the quorums need not all mutually intersect, as discovered independently by Howard, Malkhi and Spiegelman [19].

A. Implementing the value function

In an implementation of a RSM, the values chosen represent the transitions that the state machines must perform. It is possible that these values may be expensive to transfer between nodes because they could carry a large quantity of data.

In the original presentation of the Synod algorithm, the values of ballots are carried along with their identifiers in the messages $\mathbf{promised}(a, b; b', v(b'))$, $\mathbf{proposed}(b, v(b))$ and $\mathbf{chosen}(b, v(b))$. This means that, in a unicast network of $2f+1$ nodes, each value is included in $2f$ messages (at least f proposals and at least f success messages) even in the absence of faults, which is twice as many as necessary. Furthermore, a simple method for detecting faults is to insist that each node sends at least one complete message within a certain period of time, but this method is unsatisfactory if messages can be unboundedly large.

Observation O4 in [8] notes that these values can be replaced in some cases by hashes, but this idea can be taken a step further and the values can be completely elided from the messages that take part in the Synod protocol, allowing considerably more freedom in the implementation of the function v without sacrificing consistency.

By allowing the values to be communicated using a separate mechanism from the consensus messages themselves it is possible to seek optimisations that rely on the fact that the values may be large but need not move quickly whereas the consensus messages are small but must be transported with low latency to ensure the system has good performance.

Although it appears that the function v is fixed, in practice it is allowed to change as the system runs. Treating it as fixed simplifies the consistency proof and highlights that its values need not be included in all messages, but means that the system cannot be shown to satisfy any useful liveness properties. To recover liveness, note that if the invariants of fig. 1 are satisfied with a value function v then they continue to be satisfied if v is replaced by another value function v' that agrees with v on proposed ballots, i.e. where $v(b) = v'(b)$ if $\mathbf{proposed}(b)$ but not necessarily otherwise. Since only $\mathrm{owner}(b)$ may propose b, if it has not yet itself proposed a value for b it can deduce that $\neg\mathbf{proposed}(b)$ and therefore freely modify $v(b)$.

It is also important for liveness that the implementation of v is resilient to the same failure modes as the rest of the system. Since $\mathrm{owner}(b)$ is, in a sense, responsible for the value of the ballot b, it is possible to think of v as an insert-only set of pairs $\{\langle b, v(b)\rangle \mid \mathbf{proposed}(b)\}$ which is an example of a convergent replicated data type [20] and can therefore be implemented simply and robustly in a distributed system without needing to rely on a consensus algorithm. Indeed the original presentation can be seen as containing such an implementation, where the inclusion of values in all messages ensures convergence occurs as quickly as possible, and replicating this set across all $2f + 1$ nodes ensures v itself may be resilient to as many as $2f$ failures. Cheap Paxos [8] is cheaper partly because it replicates v across just the $f + 1$ primary processors, with the f auxiliary processors storing just the hashes of values to ensure integrity.

It is also worth comparing this approach to that of Vertical Paxos [10] which takes great care to ensure that the system state is completely transferred between nodes before they start to participate fully in the cluster, with certain optimisations

in recognition of the fact that this state transfer could be an expensive and time-consuming operation involving a very large quantity of data. However if the implementation of v is separated out then the quantity of data that must be transferred as part of the consensus algorithm becomes small enough that it needs no special treatment and a consensus-free technique may be used to implement v more efficiently.

B. Per-phase quorums

The consistency property of the Synod algorithm relies on the fact that the set of nodes involved in completing phase I must always intersect the set of nodes involved in completing phase II so that there is at least one node involved in both phases. Write $Q_1 \frown Q_2$ iff every $q_1 \in Q_1$ and $q_2 \in Q_2$ have $q_1 \cap q_2 \neq \varnothing$, and write $Q^{\mathrm{I}}(b)$ and $Q^{\mathrm{II}}(b)$ for the sets of phase-I and phase-II quorums for ballot b respectively.

In the original presentation of the Synod algorithm any (weighted) majority of the nodes could be used as a quorum, so that $Q^{\mathrm{I}}(b_1) = Q^{\mathrm{II}}(b_2)$ and hence $Q^{\mathrm{I}}(b_1) \frown Q^{\mathrm{II}}(b_2)$ for all b_1 and b_2 since all majorities intersect.

Theorem 8 shows that consistency can still be guaranteed even if sometimes $Q^{\mathrm{I}}(b_1) \neq Q^{\mathrm{II}}(b_2)$, as long as $Q^{\mathrm{I}}(b_1) \frown Q^{\mathrm{II}}(b_2)$ when $\mathbf{proposed}(b_1)$, $\mathbf{chosen}(b_2)$ and $b_1 \succ b_2$, as described in invariant S1. This weaker invariant is the key to allowing more general reconfigurations to take place safely as described in section IV-A below.

IV. The Paxos algorithm

Conceptually, Paxos is a sequence of distinct instances of the Synod algorithm all running simultaneously. To achieve this, the messages of the Synod algorithm above are indexed with the instance number $i \in \mathbb{N}$: $\mathbf{promised}_i(a,b)$, $\mathbf{promised}_i(a,b;b')$, $\mathbf{proposed}_i(b)$, $\mathbf{accepted}_i(a,b)$ and $\mathbf{chosen}_i(b)$. There is another kind of message known as a *multi-promise*, written $\mathbf{promised}_{\geq i}(a,b)$, which can be thought of as standing for the infinite set of free promises $\{\mathbf{promised}_j(a,b) \mid j \geq i\}$. Prepare messages $\mathbf{prepare}(b)$ apply to all instances so are not indexed.

As in the Synod algorithm, phase I starts with a broadcast of $\mathbf{prepare}(b)$ for some b to which each node a may respond with a set of promises $\mathbf{promised}_i(a,b)$, $\mathbf{promised}_i(a,b;b')$ and $\mathbf{promised}_{\geq i}(a,b)$ according to its past behaviour. Each phase II instance i operates just as in the Synod algorithm, starting with a broadcast of $\mathbf{proposed}_i(b)$ to which each node a may respond with an acceptance $\mathbf{accepted}_i(a,b)$ and once acceptances have been received from a quorum of nodes it follows that $\mathbf{chosen}_i(b)$ may be broadcast.

There is a function $v_i : \mathbb{B} \to \mathbb{V}$ for each instance i giving a value to each ballot, and theorem 10 shows that whenever $\mathbf{chosen}_i(b)$ and $\mathbf{chosen}_i(b')$ it follows that $v_i(b) = v_i(b')$.

The invariants listed in fig. 2 are roughly the same as for many other presentations of Paxos with the addition of constraints on the configurations associated with instances and ballots as discussed below. Note that invariants P2, P3 and P4 limit the acceptances that can be sent as well as the promises, so there is no need to define separate invariants concerning the sending of acceptances.

A. Configuration changes

A fixed cluster configuration is a pair $\langle Q^{\mathrm{I}}, Q^{\mathrm{II}} \rangle$ of sets of quorums satisfying $Q^{\mathrm{I}} \frown Q^{\mathrm{II}}$, where Q^{I} and Q^{II} are the sets of quorums to use in phase I and phase II respectively.

Changes to the cluster configuration are modelled by a sequence of configurations $\langle Q^{\mathrm{I}}_0, Q^{\mathrm{II}}_0 \rangle, \langle Q^{\mathrm{I}}_1, Q^{\mathrm{II}}_1 \rangle, \ldots$ that also satisfy $Q^{\mathrm{I}}_e \frown Q^{\mathrm{II}}_{e+1}$ for all e. The integer subscript is called the *era* of a configuration. Intuitively the cluster is "in era e" while instances are being chosen using $\langle Q^{\mathrm{I}}_e, Q^{\mathrm{II}}_e \rangle$. While a change from era e to $e+1$ is in progress some instances may use the interim configuration $\langle Q^{\mathrm{I}}_e, Q^{\mathrm{II}}_{e+1} \rangle$, and once the change to era $e+1$ is complete instances will use $\langle Q^{\mathrm{I}}_{e+1}, Q^{\mathrm{II}}_{e+1} \rangle$. This intuition is captured more precisely in section IV-D below, and since $Q^{\mathrm{II}}_e \frown Q^{\mathrm{I}}_e \frown Q^{\mathrm{II}}_{e+1} \frown Q^{\mathrm{I}}_{e+1}$ it follows that consistency is preserved throughout by the observation of III-B above.

To achieve this there are also two nondecreasing integer-valued functions, both written $e(\cdot)$, which respectively assign an era $e(i)$ to each instance i, and an era $e(b)$ to each ballot b. Intuitively $e(b)$ records which quorums may be used in phase I to decide that $\mathbf{proposed}_i(b)$ can be sent, and $e(i)$ records which quorums may be used in phase II to decide that $\mathbf{chosen}_i(b)$ can be sent. More precisely a node may emit $\mathbf{proposed}_i(b)$ only if it has received promises for ballot b in instance i from a quorum of nodes in $Q^{\mathrm{I}}_{e(b)}$, which implies that $e(b) \leq e(i)$, and similarly a node may emit $\mathbf{chosen}_i(b)$ only if $e(i) \leq e(b)+1$ and it has received $\mathbf{accepted}_i(a,b)$ from a quorum of nodes in $Q^{\mathrm{II}}_{e(i)}$. These extra conditions on the eras of ballots and instances in messages ensure that if $\mathbf{chosen}_i(b)$ then $e(b) \leq e(i) \leq e(b)+1$ and hence $Q^{\mathrm{I}}_{e(b)} \frown Q^{\mathrm{II}}_{e(i)}$ as required to ensure consistency, as shown in lemma 9.

B. Dynamic configuration changes

As in Dynamic Paxos, the configurations $\langle Q^{\mathrm{I}}_0, Q^{\mathrm{II}}_0 \rangle, \langle Q^{\mathrm{I}}_1, Q^{\mathrm{II}}_1 \rangle, \ldots$ and era numbers $e(i)$ are themselves chosen by consensus. In contrast, era numbers $e(b)$ for ballots b are fixed in advance and not chosen by consensus.

In more detail, configurations are held within the RSM as a finite sequence $\langle Q^{\mathrm{I}}_0, Q^{\mathrm{II}}_0 \rangle, \langle Q^{\mathrm{I}}_1, Q^{\mathrm{II}}_1 \rangle, \ldots, \langle Q^{\mathrm{I}}_{e_{\max}}, Q^{\mathrm{II}}_{e_{\max}} \rangle$ and the era numbers of instances are held similarly as a nondecreasing sequence $e(0), e(1), \ldots, e(i_{\max})$. The transitions that affect these sequences may append one or more elements, but may not change any existing elements.

The sequences are always long enough to make progress, in the sense that $e_{\max} \geq e(i_{\max})$ and if $\mathbf{chosen}_j(b)$ for all $j < i$ then $i_{\max} \geq i$.

Note that a node may emit a promise for b at instance i only if $e(b) \leq e(\min(i, i_{\max}))$. This allows nodes to make promises for an instance i even if $e(i)$ is not yet known, i.e. if $i > i_{\max}$. Invariant P7 requires that $i \leq i_{\max}$, and hence $e(i)$ is known, before $\mathbf{chosen}_i(b)$.

C. Liveness

To be useful, a consensus algorithm must not only guarantee consistency but also ensure that it cannot "get stuck", i.e. it is always possible to make progress by eventually choosing a value for each instance. It is known to be impossible to

Fig. 2. Invariants preserved by the Paxos algorithm

P1) There are configurations $\langle Q^{\mathrm{I}}_0, Q^{\mathrm{II}}_0\rangle, \langle Q^{\mathrm{I}}_1, Q^{\mathrm{II}}_1\rangle, \dots$ where $Q^{\mathrm{II}}_e \frown Q^{\mathrm{I}}_e \frown Q^{\mathrm{II}}_{e+1}$ for each e.
P2) If $\mathbf{promised}_{\geq i}(a, b)$ then $e(b) \leq e(\min(i, i_{\max}))$ and $\neg\mathbf{accepted}_j(a, b')$ for all $j \geq i$ and all $b' \prec b$.
P3) If $\mathbf{promised}_i(a, b)$ then $e(b) \leq e(\min(i, i_{\max}))$ and $\neg\mathbf{accepted}_i(a, b')$ for all $b' \prec b$.
P4) If $\mathbf{promised}_i(a, b; b')$ then $e(b) \leq e(\min(i, i_{\max}))$, $b' \prec b$, $\mathbf{accepted}_i(a, b')$, and b' is the greatest such ballot in the sense that $\neg\mathbf{accepted}_i(a, b'')$ for all b'' having $b' \prec b'' \prec b$.
P5) If $\mathbf{proposed}_i(b)$ then there is a quorum $q \in Q^{\mathrm{I}}_{e(b)}$ such that for every $a \in q$ one of the following holds:
- $\mathbf{promised}_{\geq j}(a, b)$ for some $j \leq i$, or
- $\mathbf{promised}_i(a, b)$, or
- $\mathbf{promised}_i(a, b; b')$ for some b'.

Furthermore if $P \triangleq \{b' \mid \exists a \in q.\mathbf{promised}_i(a, b; b')\} \neq \varnothing$ then $v_i(b) = v_i\big(\max(P)\big)$.
P6) If $\mathbf{accepted}_i(a, b)$ then $\mathbf{proposed}_i(b)$.
P7) If $\mathbf{chosen}_i(b)$ then $i \leq i_{\max}$, $e(i) \leq e(b) + 1$, and there is a quorum $q \in Q^{\mathrm{II}}_{e(i)}$ with $\mathbf{accepted}_i(a, b)$ for every $a \in q$.

guarantee liveness in a deterministic asynchronous system [21] but as with earlier presentations of Paxos [7] here liveness can be shown under the assumption that a distinguished node ℓ is eventually selected as the only one that may emit $\mathbf{prepare}(b)$ messages.

The original liveness proof then proceeded by having ℓ emit $\mathbf{prepare}(b)$ for some b that is chosen to be large enough that a quorum of nodes may respond with promises. In contrast, here there is an upper bound on suitable ballots since a proposed ballot must not belong to an era which is too large, because if $e(b) > e(i)$ then no promise for ballot b at instance i may be made. Therefore here ℓ must be able to choose a ballot that is large enough to be accepted but which still belongs to the correct era, or, more precisely, for each ballot b, each era $e \geq e(b)$ and each node a there must be a ballot $b' \succ b$ having $e(b') = e$ and $\mathrm{owner}(b') = a$.

This means that ballot numbers cannot be simple integers because this would imply that there exist eras containing only finitely many ballots. Instead an implementation could, for example, let $\mathbb{B} = \mathbb{N} \times \mathbb{N} \times \mathbb{A}$ ordered lexicographically, where $e(\langle e, n, a\rangle) \triangleq e$ and $\mathrm{owner}(\langle e, n, a\rangle) \triangleq a$. This is the approach used in Egalitarian Paxos [11] in which eras are known as *epochs* but this terminology is avoided here to prevent confusion with the epochs (views, terms, ...) of leader-election protocols (e.g. [22]) which track the current leader rather than the current configuration.

With this in mind the proof of liveness runs much as in the original presentation:

Theorem 1 (Liveness)**.** *Given that there is eventually a nonfaulty distinguished node ℓ which is the only node that may emit prepare messages, and sufficiently many other nonfaulty nodes, and given that for every instance there is eventually at least one value to propose, then eventually a value is chosen for every instance.*

Proof. The proof proceeds by induction over the instances, so suppose that $\mathbf{chosen}_j(b_j)$ for all $j < i$ and show that eventually $\mathbf{chosen}_i(b_i)$ as follows.

Firstly recall that the sequences of eras and configurations are long enough, in the sense that $i \leq i_{\max}$ and $e(i_{\max}) \leq e_{\max}$, so that the values of $e(i)$, $Q^{\mathrm{I}}_{e(i)-1}$, $Q^{\mathrm{I}}_{e(i)}$ and $Q^{\mathrm{II}}_{e(i)}$ are known to ℓ.

The distinguished node ℓ first chooses a ballot b_i having $e(b_i) \in \{e(i) - 1, e(i)\}$ and $\mathrm{owner}(b_i) = \ell$ and such that enough nodes can emit $\mathbf{accepted}_i(a, b_i)$ without breaking any of their previously-made promises. By invariants P2, P3 and P4, all promises for such a ballot b' at instance i must have $e(b') \leq e(i)$ so that such a b_i does exist.

If ℓ has not yet received enough promises for b_i at instance i then it broadcasts $\mathbf{prepare}(b_i)$ and waits to receive promises from a quorum of nodes in $Q^{\mathrm{I}}_{e(b_i)}$.

Then, if ℓ has not yet emitted $\mathbf{proposed}_i(b_i)$ it selects one of the values for instance i (which eventually exists), sets $v_i(b_i)$ as appropriate, broadcasts $\mathbf{proposed}_i(b_i)$, and waits to receive acceptances in response. When acceptances have been received from a quorum of nodes in $Q^{\mathrm{II}}_{e(i)}$ it follows that $\mathbf{chosen}_i(b_i)$ as required. □

D. Fully concurrent configuration changes

The discussion so far shows that, like other Paxos variants, this algorithm satisfies consistency and liveness properties. The benefit of this scheme compared with other variants is that, under normal running conditions, it is possible to perform a reconfiguration without needing to impose a limit on the number of concurrently-running instances. This section describes the details of this procedure.

In normal running there is an instance i_0 and a ballot b with $e(b) = e(i_0) = e(i_{\max}) = e_{\max}$ and the distinguished node $\ell = \mathrm{owner}(b)$ has received a quorum of promises $q^{\mathrm{I}} \in Q^{\mathrm{I}}_{e(b)}$ for ballot b for all instances $i \geq i_0$. In this state, ℓ may emit $\mathbf{proposed}_i(b)$ for any $i \geq i_0$ as long as $v_i(b)$ is set appropriately. The node ℓ is known as the *leader* and its proposals are normally accepted without undue delay by all other nodes.

Suppose that, in normal running, an operator wishes to change the cluster configuration to $\langle Q^{\mathrm{I}}_{\text{new}}, Q^{\mathrm{II}}_{\text{new}}\rangle$ where $Q^{\mathrm{I}}_{e(b)} \frown Q^{\mathrm{II}}_{\text{new}} \frown Q^{\mathrm{I}}_{\text{new}}$. First she appends $\langle Q^{\mathrm{I}}_{\text{new}}, Q^{\mathrm{II}}_{\text{new}}\rangle$ to the sequence of configurations, setting $\langle Q^{\mathrm{I}}_{e(b)+1}, Q^{\mathrm{II}}_{e(b)+1}\rangle = \langle Q^{\mathrm{I}}_{\text{new}}, Q^{\mathrm{II}}_{\text{new}}\rangle$ and $e_{\max} = e(b) + 1$, then she picks a future instance $i_c > i_{\max}$ at which the change should take effect and appends values to the sequence of eras to set $e(i) = e(i_0)$ for $i_0 \leq i < i_c$ and $e(i_c) = e(i_0) + 1$. Since $e(i_c) = e(i_0) + 1 \leq$

$e(b)+1$, values for instance i_c and any future instances with the same era may be proposed and chosen even though phase I has not yet run for a ballot in this era, so this does not prevent any further instances from running concurrently. There is therefore no drawback to choosing as small a value for i_c as possible, so it makes sense to choose $i_c = i_{\max} + 1$.

At this point, the system is no longer in normal running as defined above because $e(b) = e(i_{\max}) - 1$. If the operator were to increase $e(i_{\max})$ any further then there would be an instance i with $e(i) > e(b) + 1$ and hence $\neg\mathbf{chosen}_i(b)$. A value can still eventually be chosen for instance i due to theorem 1, but not before the leader has selected a new ballot b' in an appropriate era, completed phase I for b', and then broadcast new proposals for b'. These steps may cause the pipeline to stall if not completed quickly enough.

The system must therefore be returned to normal running before any further reconfiguration can occur. To do this, the leader chooses a new ballot b' having $e(b') = e(b) + 1$ and $\mathrm{owner}(b') = \ell$ and runs phase I for b' in a way that does not prevent any progress in era $e(b)$ while it has not completed. This is possible if ℓ has a *casting vote* in the sense that there are quorums of nonfailed nodes $q \in Q^{\mathrm{II}}_{e(b)}$ and $q' \in Q^{\mathrm{I}}_{e(b)+1}$ having $q \cap q' = \{\ell\}$. With a casting vote, ℓ may broadcast $\mathbf{prepare}(b')$ just to the nodes in $q' \setminus \{\ell\}$ without preventing further progress in era $e(b)$ since the nodes in q can continue to accept proposals in this era throughout. When it has received promises from all the other nodes in q' it can send itself $\mathbf{promised}_{\geq i'}(\ell, b')$ for some sufficiently large i', which completes phase I at b' and restores the system to normal running in era $e(b') = e(b) + 1$. A message from a node to itself does not incur any network delays so the last step occurs essentially instantaneously.

If ℓ does not have a casting vote, but there is some other node ℓ' which does, then ℓ should first abdicate its leadership to ℓ' and then the new leader should perform the new phase I as described above. If there is no node with a casting vote at all then the broadcast of $\mathbf{prepare}(b')$ may prevent progress until phase I is complete at b'.

If any node fails during this process then it may be necessary to retry some of the steps or possibly even elect a new leader. Liveness and consistency continue to hold if nodes fail but performance may be affected, for instance by meaning that ℓ no longer has a casting vote. In general it is not possible to prevent node failures from having a performance impact.

To give a concrete example of this process, fig. 3 shows the flow of messages during a reconfiguration involving the nodes a_1 and a_2 and the leader ℓ. The system starts in era e where $\{\ell, a_1\} \in Q^{\mathrm{I}}_e$ and $\{\ell, a_2\} \in Q^{\mathrm{II}}_e$, and moves to era $e+1$ where $\{\ell, a_1\} \in Q^{\mathrm{I}}_{e+1}$ and $\{\ell, a_2\} \in Q^{\mathrm{II}}_{e+1}$ too. Because of these quorums, the leader only needs a response from node a_1 to complete phase I, and similarly only needs a response from node a_2 to complete phase II and choose a value. Notice that this means the leader has a casting vote. The messages sent from ℓ to the other nodes are labelled on the diagram, but the successful responses (promises and acceptances from a_1 and a_2 respectively) are left unlabelled for clarity. Initially, $i_{\max} = i + 3$ and $e(i) = e(i+1) = e(i+2) = e(i+3) = e = e_{\max}$, and all instances before i have already been chosen.

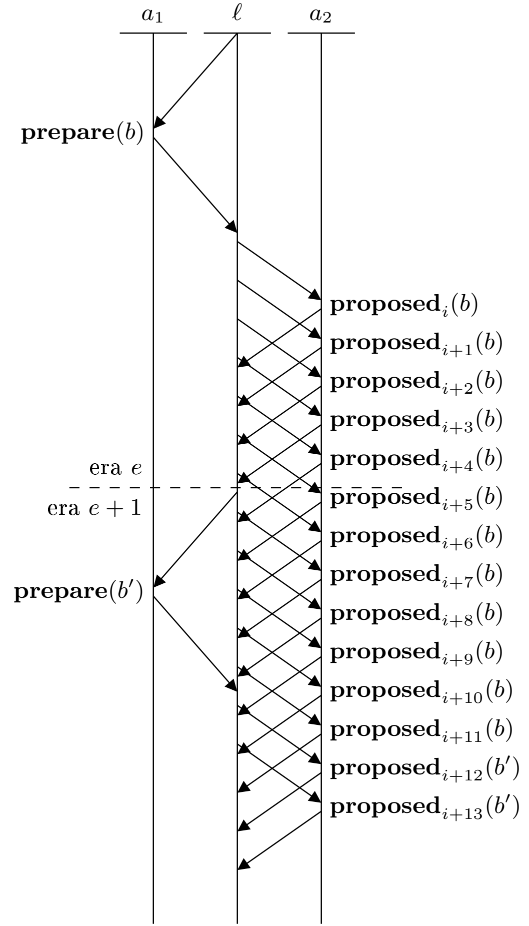

Fig. 3. Message flow during a reconfiguration.

The leader starts by completing phase I at a sufficiently large ballot b, where $e(b) = e$, which starts a period of normal running in era e and means that ℓ may emit $\mathbf{proposed}_j(b)$ for any $j \geq i$. It receives three client requests, causing it to propose values for instances i, $i+1$ and $i+2$ in turn. On the receipt of each proposal the node a_2 responds with an acceptance, which when ultimately received by ℓ allows it to decide that each value is chosen because $\{\ell, a_2\} \in Q^{\mathrm{II}}_e$.

A configuration change is proposed at instance $i+3$ which, when chosen, appends the next configuration $\langle Q^{\mathrm{I}}_{e+1}, Q^{\mathrm{II}}_{e+1} \rangle$ to the configuration sequence, sets $e(i+4) = e(i+5) = \ldots = e+1$, and increases $e_{\max}$ and $i_{\max}$ accordingly. This takes the system out of normal running because now $e(i_{\max}) = e+1 \neq e(b)$. In order to bring the system back into normal running, the leader must choose a ballot b' having $e(b') = e_{\max}$ and complete a phase I at b', so it sends $\mathbf{prepare}(b')$ to a_1.

While instance $i+3$ was being chosen, the leader continued to service client requests by sending out $\mathbf{proposed}_{i+4}(b)$, $\mathbf{proposed}_{i+5}(b)$ and $\mathbf{proposed}_{i+6}(b)$, which a_2 accepts in due course. Although these instances come after the configuration change at instance $i+3$, when the leader receives $\mathbf{accepted}_{i+4}(a_2, b)$ it may still safely deduce $\mathbf{chosen}_{i+4}(b)$ since $e(i+4) = e+1 \leq e(b)+1$ and $\{\ell, a_2\} \in Q^{\mathrm{II}}_{e+1}$, and similarly for instances $i+5$ and $i+6$. It is important to notice that the leader is now using Q^{II}_{e+1} and not Q^{II}_e to determine

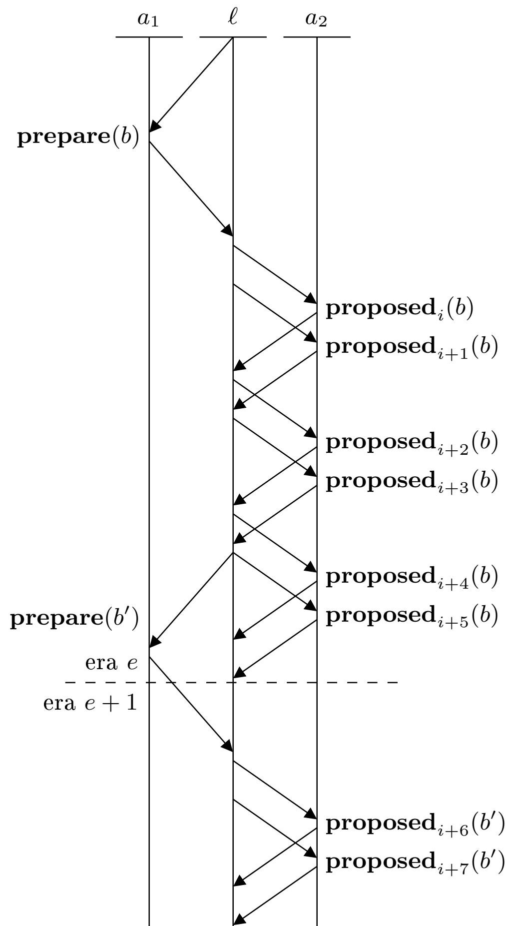

Fig. 4. Message flow for Dynamic Paxos with $\alpha = 2$.

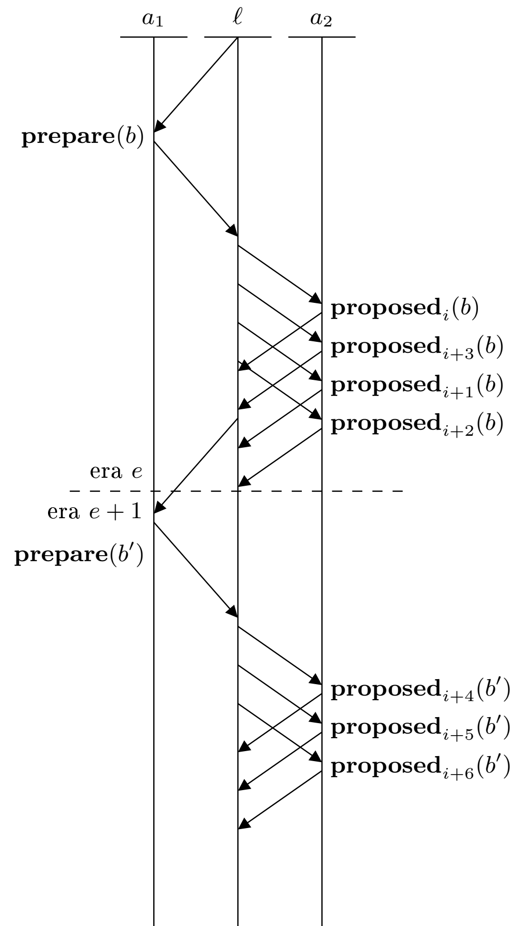

Fig. 5. Message flow for Stoppable Paxos.

when ballots are chosen. The horizontal dashed line shows the point in time at which the leader moves from era e to era $e+1$.

While the leader is waiting for the response from a_1, client requests continue to arrive, yielding proposals for subsequent instances $i+7$, $i+8$, ... Notice that the leader is still using ballot b for these requests as it has not yet completed phase I at b', but that it is still safe to deduce $\mathbf{chosen}_{i+7}(b)$, $\mathbf{chosen}_{i+8}(b)$, ... because $e(i+7) = e(i+8) = e(b)+1$. Importantly, there is no limit to how many requests the leader can handle in this way, so the system will continue to be able to process client requests even if the phase I messages are arbitrarily delayed.

At last the response from a_1 is received just after the sending of $\mathbf{proposed}_{i+11}(b)$ and before a proposal has been made for instance $i+12$. The leader can then send itself the message $\mathbf{promised}_{\geq i+12}(\ell, b')$ which completes phase I at ballot b' since $\{\ell, a_1\} \in Q^{\mathrm{I}}_{e+1}$. This restores the system to normal running, and allows ℓ to make proposals $\mathbf{proposed}_j(b')$ for all $j \geq i+12$. It is imporant to notice that the leader did not need to predict how long it would take to complete phase I at b' in advance, nor how many requests it might have to handle during this time, in order to ensure that it can continue to process these requests without delay.

Fig. 4 shows an equivalent reconfiguration performed in a Dynamic Paxos cluster with the pipeline length $\alpha = 2$. As there are frequently two proposals being processed concurrently it seems likely that better performance could be achieved by selecting a higher value. A higher value still could have avoided the pause between instances $i+5$ and $i+6$ caused by the extra delay in completing phase I at ballot b'. On the other hand if the pipeline is too long then configuration changes can be expensive to complete. It is, in general, difficult to select an appropriate value for α up-front as the best choice may depend on changeable system conditions, and no matter what value is selected it is possible that a configuration change may cause a pause if a phase-I message is unexpectedly delayed or a burst of client requests are received.

In contrast, fig. 5 shows an equivalent reconfiguration performed in Stoppable Paxos. This variant of Paxos allows for an unlimited number of proposals to run in parallel within each configuration, and permits out-of-order execution, so in this illustration the stopping command is proposed at instance $i+3$ before proposals are made for the two preceding instances. By selecting instance $i+3$ for the reconfiguration, the operator is limiting the system to service at most two more client requests before the reconfiguration completes. As in the illustration of Dynamic Paxos above, the operator's choice is too conservative so the remaining two free instances are used up before phase I is completed at ballot b', which causes the system to temporarily suspend its processing of client requests.

V. EXAMPLES

This section contains some examples of configuration changes that satisfy the conditions described above. All the configurations described here have equal sets of quorums in phase I and phase II of each era, so for the sake of simplicity throughout this section define $Q_e \triangleq Q_e^{\mathrm{I}} = Q_e^{\mathrm{II}}$. Implementations can ensure $Q_e \frown Q_e$ for each e by, for instance, arranging for each quorum in Q_e to comprise a majority subset of some finite set of nodes. Slightly more generally, let a *weight function* be a function $w : \mathbb{A} \to \mathbb{N}$ that only takes finitely many nonzero values. This can be used to define a configuration $\langle M(w), M(w)\rangle$ by *weighted majority*:

$$M(w) \triangleq \left\{ q \;\middle|\; \sum_{a\in q} 2w(a) > \sum_{a\in\mathbb{A}} w(a) \right\}.$$

Corollary 5 in appendix A demonstrates the well-known result that $M(w) \frown M(w)$ for any weight function w. Indeed, if w and w' are weight functions that differ by a constant factor in the sense that that there are positive integers k and k' with $kw(a) = k'w'(a)$ for all a, then clearly $M(w) = M(w')$ and hence $M(w) \frown M(w')$.

Raft's quorums are simple unweighted majorities of a finite set of nodes, which can be emulated with weight functions that only take values in $\{0, 1\}$. It only supports adding or removing a single node from this set which amounts to changing the weight of a single node by ± 1. For instance, if $\mathbb{A} = \{a_1, a_2, \ldots\}$ then define weight functions

$$w^{1\ldots n}(a) \triangleq \begin{cases} 1 & \text{if} a \in \{a_1, \ldots, a_n\} \\ 0 & \text{otherwise} \end{cases}$$

and observe that $M(w^{1\ldots3}) \frown M(w^{1\ldots4})$ and $M(w^{1\ldots4}) \frown M(w^{1\ldots5})$ but $M(w^{1\ldots3}) \not\frown M(w^{1\ldots5})$ because $\{a_1, a_2\} \in M(w^{1\ldots3})$ and $\{a_3, a_4, a_5\} \in M(w^{1\ldots5})$ do not intersect. This justifies the restriction against adding or removing more than one node at once.

In fact there is no need to restrict attention just to weight functions taking values in $\{0, 1\}$ as shown by lemma 3 which is reminiscent of the amoeba analogy in [8]: any two integer-valued weight functions whose total absolute difference is at most one can be used to define consecutive configurations. This extra generality is important for deployments where nodes may share infrastructure (e.g. power distribution or network connectivity) because such nodes may suffer correlated failures, and reducing this correlation by adding more independent infrastructure may be costly. In more detail, a naïve approach to swapping a node a_{old} for a replacement a_{new} in a three-node cluster using unweighted majorities would be to perform the configuration changes given by this sequence of weight functions starting at era e:

node	a_{old}	a_{new}	a_1	a_2
w_e	1	0	1	1
w_{e+1}	1	1	1	1
w_{e+2}	0	1	1	1

However to be resilient to infrastructure failures this requires all four nodes and their underlying infrastructures to be completely independent since a correlated failure of any two nodes would prevent further progress. It also has no node with a casting vote in era $e + 1$. On the other hand the following sequence achieves the same overall change but allows a_{old} and a_{new} to share infrastructure without extra risk, and both a_1 and a_2 have casting votes throughout:

node	a_{old}	a_{new}	a_1	a_2
w_e	1	0	1	1
w_{e+1}	2	0	2	2
w_{e+2}	2	1	2	2
w_{e+3}	1	1	2	2
w_{e+4}	0	1	2	2
w_{e+5}	0	2	2	2
w_{e+6}	0	1	1	1

This is important as in many operating environments it may be too expensive or complicated to arrange for four independent infrastructures particularly if the fourth is only required to ensure consistency in relatively rare periods of maintenance. For instance at time of writing only one Amazon Web Services region (`us-east-1`) has four independent zones, whereas four of them have three: `ap-southeast-2`, `eu-west-1`, `sa-east-1` and `us-west-2`. Similarly, only one Google Cloud Platform region (`us-central1`) has four zones and all the others have three zones.

Early versions of Raft supported more general reconfigurations using a technique known as *joint configurations*. To change from configuration Q_e to an unrelated configuration Q' (i.e. $Q_e \not\frown Q' \frown Q'$) it is possible to set $Q_{e+2} = Q'$ and set Q_{e+1} to be the *joint configuration* of Q_e and Q':

$$Q_{e+1} = \{q \cup q' \mid q \in Q_e, q' \in Q'\}$$

as this satisfies that $Q_e \frown Q_{e+1} \frown Q_{e+1} \frown Q'$.

If $w(a) = 0$ for all a then w is said to be *weightless* and $M(w) = \varnothing$. Clearly if there is an instance i such that $Q_{e(i)} = \varnothing$ then no value can ever be chosen for i. On the other hand $Q \frown \varnothing$ for all Q so changing to or from a weightless configuration is always permitted.

There is no requirement for the eras of consecutive instances to differ by at most one so an era may be skipped at instance i by setting $e(i) = e(i - 1) + 2$. This recovers the ability to perform arbitrary configuration changes in a single step as in Stoppable Paxos. In more detail, if the system is currently using configuration Q_e and an operator wishes to change to an unrelated configuration Q' then she can set $Q_{e+2} = Q'$ and pick an appropriate Q_{e+1} (such as $\varnothing$) which satisfies that $Q_e \frown Q_{e+1} \frown Q_{e+1} \frown Q'$. However, re-running phase I can only be delayed as described in section IV-D above if the era increases by 1 and in this situation the era increases by 2, so a new phase I must be completed before phase II of any new instances can be started. To prevent this causing the pipeline to stall, the operator chooses a sufficiently large $\alpha > 0$ and sets $e(i + \alpha) = e(i) + 2$ and $e(j) = e(i)$ for $i < j \leq i + \alpha$, effectively delaying the configuration change for α instances in the hope that this is long enough to have completed the new phase I. The pattern of communication in this situation is very similar to that shown in fig. 5 where α was chosen to be 3.

VI. Conclusion

The approach described here improves on Dynamic Paxos [8] by supporting changing the pipeline length parameter α and running with an unlimited-length pipeline while a configuration change is not in progress. In that sense, it can be compared to that of Stoppable Paxos [9] which allows for an unlimited number of proposals to run in parallel within each configuration, but requires a temporary arbitrary limit on concurrency while a reconfiguration takes place.

In both Dynamic and Stoppable Paxos, if the selected limit is either too small or too large then it may affect the system's performance, and the best choice of limit depends on a prediction of the system's future performance. The approach described here avoids the need to make any such prediction or select any such limit and responds to changing system conditions without needing further tuning. Once a reconfiguration is chosen, it can complete after a single round-trip to a quorum of nodes and the system can continue to serve all clients while this round-trip is in progress, no matter how long it takes.

This is achieved by using Raft-style reconfigurations [3] which can be performed with an unlimited pipeline throughout. Unlike in Raft, here a configuration change only takes effect once it is chosen, which avoids the need to back-track to an earlier state if a leader fails during reconfiguration. It generalises the simple majorities used in Raft to integer-weighted majorities which can reduce the costs of dealing with correlated failures during maintenance.

It achieves equivalent goals to those of Vertical Paxos [10] except that here there is no requirement for a separate oracle to manage the configuration of the system.

It is also noted that there is no need for every message to include the corresponding value, or even a hash of the value, which may allow for even cheaper implementations of Cheap Paxos [8] and can simplify the transfer of state [10] required when new nodes are commissioned.

Appendix A

Lemma 2. *If $w, w' : \mathbb{A} \to \mathbb{N}$ are weight functions and k, k' are positive integers such that $\sum_{a\in\mathbb{A}} |k'w'(a) - kw(a)| \leq 1$ then $M(w) \frown M(w')$.*

Proof. Since $kw(a)$ and $k'w'(a)$ are integers for all a, there must be a node a_0 such that $kw(a) = k'w'(a)$ for all $a \neq a_0$. Let $q \in M(w)$ and $q' \in M(w')$. By the definition of M, and since w and w' take only integer values, $\sum_{a\in q} 2w(a) \geq \sum_{a\in\mathbb{A}} w(a) + 1$ and $\sum_{a\in q'} 2w'(a) \geq \sum_{a\in\mathbb{A}} w'(a) + 1$. Let $d_\mathbb{A} \triangleq k'w'(a_0) - kw(a_0)$ so that $\sum_{a\in\mathbb{A}} k'w'(a) = \sum_{a\in\mathbb{A}} kw(a) + d_\mathbb{A}$ and $|d_\mathbb{A}| \leq 1$. Also let

$$d_{q'} \triangleq \begin{cases} d_\mathbb{A} & a_0 \in q' \\ 0 & \text{otherwise,} \end{cases}$$

so that $\sum_{a\in q'} k'w'(a) = \sum_{a\in q'} kw(a) + d_{q'}$. Then

$$\begin{aligned}
&\sum_{a\in\mathbb{A}} 2kw(a) + d_\mathbb{A} + k + k' \\
&\qquad = k\left(\sum_{a\in\mathbb{A}} w(a) + 1\right) + k'\left(\sum_{a\in\mathbb{A}} w'(a) + 1\right) \\
&\qquad \leq \sum_{a\in q} 2kw(a) + \sum_{a\in q'} 2k'w'(a) \\
&\qquad = \sum_{a\in q} 2kw(a) + \sum_{a\in q'} 2kw(a) + 2d_{q'} \\
&\qquad = \sum_{a\in q\cup q'} 2kw(a) + \sum_{a\in q\cap q'} 2kw(a) + 2d_{q'} \\
&\qquad \leq \sum_{a\in\mathbb{A}} 2kw(a) + \sum_{a\in q\cap q'} 2kw(a) + 2d_{q'}
\end{aligned}$$

so that $\sum_{a\in q\cap q'} 2kw(a) \geq d_\mathbb{A} + k + k' - 2d_{q'} = k + k' \pm d_\mathbb{A} \geq 1$ and hence $q \cap q' \neq \varnothing$ as desired. ☐

Lemma 3. *If $w, w' : \mathbb{A} \to \mathbb{N}$ are weight functions such that $\sum_{a\in\mathbb{A}} |w'(a) - w(a)| \leq 1$ then $M(w) \frown M(w')$.*

Proof. By lemma 2 with $k = k' = 1$. ☐

Lemma 4. *If $w, w' : \mathbb{A} \to \mathbb{N}$ are weight functions and k, k' are positive integers such that $k'w'(a) = kw(a)$ for all a then then $M(w) \frown M(w')$.*

Proof. By lemma 2, since $\sum_{a\in\mathbb{A}} |k'w'(a) - kw(a)| = 0$. ☐

Corollary 5. *If $w : \mathbb{A} \to \mathbb{N}$ is a weight function then*

$$M(w) \frown M(w).$$

Proof. By lemma 4 with $w' = w$ and $k' = k$. ☐

Appendix B
Consistency of the Synod algorithm

Lemma 6. *If $\mathbf{accepted}(a, b_2)$, $\mathbf{promised}(a, b_1; b_3)$ and $b_2 \prec b_1$ then $b_2 \preceq b_3$.*

Proof. From invariant S3 it follows that b_3 is the largest ballot such that $b_3 \prec b_1$ and $\mathbf{accepted}(a, b_3)$, but b_2 is also such a ballot and therefore $b_2 \preceq b_3$ as required. ☐

Lemma 7. *If $\mathbf{chosen}(b_2)$, $\mathbf{proposed}(b_1)$ and $b_2 \prec b_1$ then $v(b_1) = v(b_2)$.*

Proof. Suppose for a contradiction that $v(b_1) \neq v(b_2)$ and since $\prec$ is wellfounded suppose without loss of generality that b_1 is the minimal such ballot. Since $\mathbf{chosen}(b_2)$ by invariant S6 there is a quorum $q^{\mathrm{II}} \in Q^{\mathrm{II}}(b_2)$ such that $\mathbf{accepted}(a, b_2)$ for every $a \in q^{\mathrm{II}}$. By invariant S2 it cannot be that $\mathbf{promised}(a, b_1)$ for any $a \in q^{\mathrm{II}}$. Also since $\mathbf{proposed}(b_1)$ by invariant S4 there is a quorum $q^{\mathrm{I}} \in Q^{\mathrm{I}}(b_1)$ such that either $\mathbf{promised}(a, b_1)$ or $\exists b'.\mathbf{promised}(a, b_1; b')$ for all $a \in q^{\mathrm{I}}$. Let $P \triangleq \{b' \mid \exists a \in q^{\mathrm{I}}.\mathbf{promised}(a, b_1; b')\}$. By invariant S1, $Q^{\mathrm{I}}(b_1) \frown Q^{\mathrm{II}}(b_2)$ and hence $q^{\mathrm{I}} \cap q^{\mathrm{II}} \neq \varnothing$ so it follows that $P \neq \varnothing$, which means that $v(b_1) = v(\max(P))$ by invariant S4. Let $a_{\max} \in q^{\mathrm{I}}$ be such that $\mathbf{promised}(a_{\max}, b_1; \max(P))$. By invariant S3 it follows that $\max(P) \prec b_1$ and also that $\mathbf{accepted}(a_{\max}, \max(P))$ and hence $\mathbf{proposed}(\max(P))$

by invariant S5. Furthermore by lemma 6 it follows that $b_2 \preceq \max(P)$ and since b_1 was assumed to be the smallest counterexample it must be that $v(\max(P)) = v(b_2)$. Hence $v(b_1) = v(b_2)$ which is a contradiction as required. □

Theorem 8. *If* $\mathbf{chosen}(b_1)$ *and* $\mathbf{chosen}(b_2)$ *then* $v(b_1) = v(b_2)$.

Proof. Without loss of generality assume that $b_2 \prec b_1$. By invariant S6 there is a quorum $q \in Q^{\text{II}}(b_1)$ such that $\mathbf{accepted}(a, b_1)$ for every node $a \in q$ and therefore $\mathbf{proposed}(b_1)$ by invariant S5. Therefore by lemma 7 it follows that $v(b_1) = v(b_2)$ as required. □

APPENDIX C
CONSISTENCY OF THE PAXOS ALGORITHM

Lemma 9. *For* $b_1 \succ b_2 \in \mathbb{B}$*, if* $\mathbf{proposed}_i(b_1)$ *and* $\mathbf{chosen}_i(b_2)$ *then* $Q^{I}_{e(b_1)} \frown Q^{II}_{e(i)}$.

Proof. $\mathbf{proposed}_i(b_1)$ implies that $\mathbf{promised}_{\geq i'}(a, b_1)$ or $\mathbf{promised}_{i'}(a, b_1)$ or $\mathbf{promised}_{i'}(a, b_1; b')$ for some node a and some $i' \leq i$ with $e(b_1) \leq e(\min(i', i_{\max}))$. Therefore $e(b_1) \leq e(i)$ since e is nondecreasing. It follows that $e(i) \leq e(b_2) + 1 \leq e(b_1) + 1 \leq e(i) + 1$ since $\mathbf{chosen}_i(b_2)$ so that $e(i) \in \{e(b_1), e(b_1)+1\}$ and hence $Q^{\text{I}}_{e(b_1)} \frown Q^{\text{II}}_{e(i)}$ by invariant P1. □

Theorem 10. *If* $\mathbf{chosen}_i(b_1)$ *and* $\mathbf{chosen}_i(b_2)$ *then* $v_i(b_1) = v_i(b_2)$.

Proof. If $\mathbf{chosen}_i(b_1)$ then the Paxos invariants imply the Synod invariants for instance i. In more detail, let

$$
\begin{aligned}
Q^{\text{I}}(b) &\triangleq Q^{\text{I}}_{e(b)} \\
Q^{\text{II}}(b) &\triangleq Q^{\text{II}}_{e(i)} \\
\mathbf{promised}(a, b) &\triangleq \mathbf{promised}_i(a, b) \\
&\quad \vee \exists i' \leq i . \mathbf{promised}_{\geq i'}(a, b) \\
\mathbf{promised}(a, b; b') &\triangleq \mathbf{promised}_i(a, b; b') \\
\mathbf{proposed}(b) &\triangleq \mathbf{proposed}_i(b) \\
\mathbf{accepted}(a, b) &\triangleq \mathbf{accepted}_i(a, b) \\
\mathbf{chosen}(b) &\triangleq \mathbf{chosen}_i(b) \text{ and} \\
v(b) &\triangleq v_i(b).
\end{aligned}
$$

Synod's invariant S1 follows from lemma 9 and the remaining invariants are simple to show so by theorem 8 it follows that $v_i(b_1) = v_i(b_2)$ as required. □

ACKNOWLEDGMENT

The author would like to thank Leslie Lamport, Dahlia Malkhi and Leander Nikolaus Jehl for their encouragement and comments on earlier drafts of this paper. The author is also very grateful to Tracsis plc for supporting this work.

REFERENCES

[1] B. W. Lampson, "How to build a highly available system using consensus," in *Proceedings of the 10th International Workshop on Distributed Algorithms*, ser. WDAG '96. London, UK, UK: Springer-Verlag, 1996, pp. 1–17.

[2] L. Lamport, "The part-time parliament," *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, vol. 16, no. 2, pp. 133–169, 1998.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320.

[4] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *Proceedings of the 2006 EuroSys Conference*. Leuven, Belgium: Association for Computing Machinery, Inc., April 2006, p. 103115.

[5] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," Microsoft Research, Tech. Rep. MSR-TR-2008-198, February 2008.

[6] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[7] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[8] L. Lamport and M. Massa, "Cheap paxos," 2004.

[9] L. Lamport, D. Malkhi, and L. Zhou, "Stoppable paxos," Microsoft Research, Tech. Rep. MSR-TR-2008-197, April 2008.

[10] ——, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 312–313.

[11] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 358–372.

[12] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 398–407.

[13] K. Birman, D. Malkhi, and R. van Renesse, "Virtually synchronous methodology for dynamic service replication," Tech. Rep., 2010.

[14] V. Bortnikov, G. Chockler, A. Roytman, S. Shachor, I. Shnayderman, and D. Perelman, *Reconfigurable state machine replication from non-reconfigurable building blocks*. ACM, 2012, pp. 93–94.

[15] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.

[16] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, ser. LADIS '08. New York, NY, USA: ACM, 2008, pp. 2:1–2:6.

[17] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, "Dynamic reconfiguration of primary/backup clusters," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 425–437.

[18] L. Jehl and H. Meling, *Distributed Computing and Networking: 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Asynchronous Reconfiguration for Paxos State Machines, pp. 119–133.

[19] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible Paxos: Quorum intersection revisited," *ArXiv e-prints*, Aug. 2016.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Conflict-Free Replicated Data Types, pp. 386–400.

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[22] D. Malkhi, F. Oprea, and L. Zhou, "Omega meets paxos: Leader election and stability without eventual timely links," in *19th Intl. Symposium on Distributed Computing (DISC 05)*, no. MSR-TR-2005-93. Cracow, Poland: European Association for Theoretical Computer Science, September 2005, p. 25.

David C. Turner received the MMath degree and the Ph.D. degree from the University of Cambridge in 2011 and 2009 respectively.

Since 2010 he has worked as a software engineer at Tracsis plc building systems to support and optimise operational planning processes in the transport industry.