Virtually Synchronous Methodology for Dynamic Service Replication

Ken Birman[†] Dahlia Malkhi[*] Robbert van Renesse[†]

November 18, 2010

Abstract

In designing and building distributed systems, it is common engineering practice to separate steady-state ("normal") operation from abnormal events such as recovery from failure. This way the normal case can be optimized extensively while recovery can be amortized. However, integrating the recovery procedure with the steady-state protocol is often far from obvious, and can present subtle difficulties. This issue comes to the forefront in modern data centers, where applications are often implemented as elastic sets of replicas that must reconfigure while continuing to provide service, and where it may be necessary to install new versions of active services as bugs are fixed or new functionality is introduced. Our paper explores this topic in the context of a dynamic reconfiguration model of our own design that unifies two widely popular prior approaches to the problem: *virtual synchrony*, a model and associated protocols for reliable group communication, and *state machine replication* (in particular, Paxos), a model and protocol for replicating some form of deterministic functionality specified as an event-driven state machine.

[*]Microsoft Research Silicon Valley, dalia@microsoft.com
[†]Cornell University, {ken,rvr}@cs.cornell.edu

1 Introduction

Our work deals with a style of distributed services that uses elastic sets of replicas to achieve availability, scalability, and long term durability. These replica sets can vary widely over time: expanding as the service confronts load surges or replaces failed server instances, and contracting as load drops or members crash. Thus, the contemporary distributed systems developer, or service developer, is unavoidably faced with issues of dynamic service replication.

Service replication and reconfiguration confront the developer with a series of tough choices. One can evade these choices by abandoning consistency, but there are many kinds of services for which consistency guarantees are important and must not be violated simply because the service replica set is elastic. For services of this type, reconfiguration remains a murky black art, error prone and associated with complex, subtle bugs. Moreover, among the correct, rigorously analyzed solutions there are all sorts of hidden tradeoffs, representing deliberate compromises between performance and complexity during steady-state operation of the service (when reconfiguration isn't happening) and the complexity of the reconfiguration event itself, which may also entail some period of service unavailability.

Our treatment of this problem is tutorial in style, but aimed at a sophisticated audience. We suspect that any developer confronting this topic will have had a considerable amount of experience with distributed systems. Accordingly, we target a reader who knows how one creates distributed services, has some familiarity with the classical papers on asynchronous consensus and communication protocols and impossibility results such as the FLP theorem, and who may even have worked with the virtual synchrony [8] or the state machine replication methodologies [19, 32, 20]. Our premise is that even knowledgeable readers be surprised to learn of some of the pitfalls, anomalies and tradeoffs that the prevailing methodologies tacitly accept. Our work unifies what had previously been different models using a single overarching methodology and formalism which we name Dynamic Service Replication (DSR).

Our goal will be to design services that can be reconfigured without disrupting correctness, up to some maximum tolerable rate of service membership changes, beyond which it becomes unavailable. We assume that reconfiguration is triggered by a *reconfiguration command,* issued by the system management layer, and either removing some members from a service, or adding some, or perhaps doing both at once. Reconfiguration could also change application parameters, or even be used to upgrade to a new version of an application or a protocol while keeping the service as available as possible.

Building a reliable service using the DSR approach entails three fundamental steps.

Safety. The first is to provide a service-oriented *safety* definition. A good specification must abstract away from implementation details, and describe the service using the methods clients can access, and expressing properties in terms of guarantees that those methods will offer. Two example services are interwoven throughout the body of our paper. The first example is a Reliable Multicast service, for which we give a service-oriented description. Our definition exposes a **Get**() API call that provide clients with a full, consistent history of messages that clients **Add**(). This definition lets us focus on the matter of preserving service state consistently while transitioning through configuration changes. Conversely, it is stripped of internal details such as clients joining and departing the service, delivery duplication, etc.; those can be added as various filters and add-ons, as discussed below. We use the well known State Machine Replication (SMR) problem [19, 32] as our second primary example. For completeness, in Section 8 we also briefly flesh out an Atomic Read/Write Storage service.

Liveness. The second ingredient of our treatment is an appropriate failure model. In the distributed computing arena, reliability is often expressed as the requirement to tolerate a threshold t failures out of an initial system of n processes. This classical fault model ignores the ability of a dynamic system to out-live such an initial setting via administrative decrees, e.g., to deploy new processes, or to remove faulty ones from consideration. For example, an initial system configuration of four processes, $\{A, B, C, D\}$, may tolerate a single failure. However, through

an administrative decree to reconfigure, say removing A after a failure, the remaining set $\{B, C, D\}$ can tolerate an additional single failure. But as this example suggests, although we have increased the overall fault tolerance through dynamism, we can't simply say that we have a system of four processes, any two of which may crash at any time. We need a more subtle condition that gives the system "sufficient time" to reconfigure. In our example, two crashes can be tolerated over the total period the system is running, but during the early part, before the system had been reconfigured, only one of $\{A, B, C, D\}$ may fail. This notion of a dynamically-defined majority appeared implicitly in various prior works, including [30, 31, 37, 24, 20, 26], but the conditions always involved solution-specific details. Here, we adopt the principles of the DynaStore liveness model [1], which gives an opaque **Reconfig**() handle for (administrative) clients. We then enhance this model to suit our more general framework.

Reconfiguration recipe. The third component is an algorithmic foundation for solution. Briefly, a DSR epoch-by-epoch reconfiguration starts with a consensus decision *by the current configuration* (say, C_1) on a new configuration (C_2). The reconfiguration procedure proceeds with a transfer of control from C_1 to C_2, which entails (a) a decision to suspend C_1; (b) a snapshot of completed and potentially-completed operations in C_1; (c) a state transfer of the snapshot to C_2. The reconfiguration completes with a decision to enable C_2 for processing new operations, with the initial state of C_2 determined by step b.

In our methodology, the developer deals with one epoch at a time. An epoch terminates with a configuration change decision. The next configuration is uniquely determined in the current epoch, and transition to it is irreversible. When an epoch ends, a new epoch starts a new incarnation of the same algorithm (albeit with a non-empty initial service state), whose participants and messages do not mix with the current epoch. If an epoch starts in a configuration that already includes some failed nodes, it might make progress in that state, or it might initiate a further round of reconfiguration to remove those nodes; indeed, it can initiate reconfiguration as its first action. The new configuration operates completely separately from the current one; the messages in it are not confused with messages in the current configuration; and it can consist of an entirely new set of machine. Any number of reconfigurations can be chained in this manner.

Solutions. While the general framework – decide, suspend, state transfer, resume – may appear obvious, there are numerous design choices and potential pitfalls when realizing any specific service. Underlying any choice of reconfiguration solution are inherent tradeoffs. One wants to maximize availability and performance, but without violating safety, and it isn't easy to accomplish all of these simultaneously. This does not mean that protocols need to be overly cumbersome. To the contrary, in six Figures, 4-10, we give succinct pseudo-code solutions for six service variants, each frame containing a entire solution including precise safety and liveness definitions.

We'll see that we can trade steady-state simplicity with continuous availability: A *fault-recovery* approach utilizes servers in steady-state in an uncomplicated manner, disregarding the possibility of failure. It can be highly optimized, but requires reconfiguration to unblock the service in case of a failure. Figure 4 illustrates this methodology within the context of Reliable Multicast and Figure 8 does so for SMR. The alternative is a *fault-masking* methodology, which crafts steady-state protocols with built-in redundancy for high availability; it adds reconfiguration functionality for even greater agility in a long-lived system. Figures 5, 6, 9 and 10 demonstrate fault-masking for multicast, SMR and read/write storage, respectively.

Reconfiguration itself presents another interesting design tradeoff. The reconfiguration procedure entails forming two consensus decisions – one on the next configuration and another on the closing state of the current. These may be obtained either among the group of servers themselves, or using a separate consensus engine. Even the latter case requires careful integration with the steady-state protocol, and we give the recipes for doing so in Figures 4 and 8. Fully distributed reconfiguration protocols are detailed in Figures 6, 10 and 9.

In general, any new members will need to catch up: for this we use the term *state transfer*; it entails packaging the state of the service of the service into some sort of external representation, copying that data to the new member(s), and them loading the state before starting to process new requests. The benefit of forming agreement

on a closing state is explained below in Section 6, using a novel formulation of an old idea, virtual synchrony. Without agreement, we'll see that a service might exhibit various forms of unexpected behavior. For example, one standard approach leads to services that are correct, but in which unfinished operations can linger in a concealed form, suddenly becoming complete "retroactively" arbitrarily far in the future, as discussed in Section 9. An alternative approach to this problem was explored in systems like RAMBO [24] which seek to offer continuous availability, to do so they must keep both the old and the new configurations active for a potentially extended period until the new one can take over entirely. Our methodology explains such behaviors, clarifies the associated design choices, and offers simple tradeoffs that let the developer select the properties needed, and to understand the associated costs and implementation issues.

We also explore a number of misconceptions regarding reconfigurable SMR. For example, when using the reconfigurable Paxos protocol, developers find themselves forced to choose between a simple solution with poor performance and a far more complex one that runs at higher speed (corresponding to the choice of value for the Paxos concurrency window parameter). However, a higher concurrency window may result in undesirable behavior where the sequence of state machine commands contains a mix of decisions out of their intended order. In fact, our paper grew out of a project to create a new "Virtually Synchronous Paxos" protocol, and our dynamically reconfigurable version of state machine replication in Section 7.2 achieves this objective (in particular, Figure 9 can be recognized as a virtually synchronous version of the Paxos protocol).

More pitfalls and anomalies of existing approaches are discussed in Section 9. This section contrasts our DSR method with respect to the three most relevant methodologies, namely, implementations of *virtually synchronous* protocols, implementations of the Paxos protocol for state machine replication, and dynamic atomic storage protocols.

Finally, we briefly sketch a correctness argument for a sample of our solutions in the Appendix.

Contribution. While some methodologies lead the developer to some single best solution, that won't be the case here; not only will the solutions we develop be incomparable in some ways, they even include application-specific considerations: the best protocols for implementing a reconfigurable reliable multicast turn out not to be directly mappable to state machine replication solution, and this illustrates just one of many such examples. Thus, readers of this paper will draw different conclusions based on the utility they individually assign to the various tradeoffs involved. Our contribution isn't some single answer, but rather a more principled treatment of the question. The methodology we offer here offers confidence in correctness, and for any given set of application-specific goals, enables steady-state performance similar to the best known hand-crafted solutions to the same problems. Also, our solution assumes less than is assumed when creating state machine solutions, and for this reason may be applicable to problems for which state machine replication is inappropriate.

In summary, our paper offers an overarching and unified reconfiguration framework, which reveals relationships between a number of prior works that led to correct and yet seemingly incomparable reconfigurable solutions in this space. Doing so helps the developer understand reconfiguration against a broad spectrum of choices, to understand the implications of those choices, and also makes it possible to see protocols that might previously have been portrayed as competing options as different realizations of a single overall goal.

2 Liveness Model

Our aim is to provide services in asynchronous systems whose set of servers is changed through explicit reconfigurations. We assume an additional set, potentially overlapping, of clients. We do not assume any bounds on message latencies or message processing times (*i.e.*, the execution environment is asynchronous), and messages may get lost, re-ordered, or duplicated on the network. However, we assume that a message that is delivered was previously sent by some live member and that correct members can eventually communicate any message.

In order to capture a formal execution model with changing memberships, we borrow concepts which express explicit dynamism from [1], but modify the treatment for weaker requirements. We proceed with a formal definition, and follow with examples.

A fixed system *membership* consists of some set of servers and liveness conditions. As usual, our goal is to build systems that are guaranteed to make progress as long as the conditions on the servers hold. If too many servers fail (violating the conditions for liveness), safety will not be impaired, but the system might need to stop responding to client requests until reconfiguration occurs. We further refine our liveness conditions by breaking them into two parts, one for performing read requests, and one for performing updates; by doing so, we can sometimes continue to support one kind of operation even if the other kind is temporarily unavailable. Example memberships are '$f + 1$ servers, f may crash on any read, no crash tolerance on update', and 'n servers, any minority of which may fail for either read or write tolerance'.

We assume an initial membership M_0. Clients are provided with a membership-changing API function **Reconfig**(M), where M is a new membership. A call to **Reconfig**(M) must eventually complete, and returns an ACK response some time after its invocation. Clients might issue concurrent **Reconfig**(M) calls, and when this occurs, the protocol orders the reconfigurations through the sequence of ACK responses. Thus, a system might perform $M_0, M_1, M_2, \ldots$ even though **Reconfig**(M_2) was invoked before **Reconfig**(M_1). We will view issues such as pre-conditions for invoking **Reconfig**, and any access restrictions (e.g., to designated administrative users) on using the **Reconfig** API as falling outside of the scope of our treament.

In our model, two execution events are associated with each **Reconfig** call, marking its invocation and its completion. Both events change the liveness conditions, so they could be thought of as 'model-changing' events in that they transform a fixed liveness-condition into another liveness-condition. For clarity, we will continue referring to them as **Reconfig** invocation and response events. The first event occurs upon invocation of **Reconfig**. It changes the current liveness condition to incorporate the requested new membership, including its set of servers and its corresponding liveness conditions. The second event in our model marks a completion of a **Reconfig** call. This event signifies that the system has re-organized to switch the service to the new set of servers and transferred all necessary information to it, so that the old membership may be garbage collected.

We define the *startup* of a membership M_k to begin with the **Reconfig**(M_k) invocation and end with its response event (for M_0, this is defined as the point the system starts). We define the *lifetime* of a membership M_k to begin with the **Reconfig**(M_k) invocation (or with the start time of the system, for M_0) and end when the succeeding reconfiguration **Reconfig**(M_{k+1}) startup is completed.

Liveness: For every membership M_k in the sequence of **Reconfig** calls, the following holds:

1. Throughout the *lifetime* of M_k, its **read**-resilience is not violated.
2. There exists a future membership M_ℓ, where $\ell > k$, such that the **update**-resilience of M_ℓ holds throughout the *startup* of **Reconfig**(M_ℓ).

Note that it follows inductively from the definitions above that memberships $M_{k+1}, ..., M_{\ell-1}$ maintain their read-resilience condition until the response event of **Reconfig**(M_ℓ).

To illustrate the features of our liveness model, let's apply it in three simple scenarios. In the first, we have a system implemented by a single server which may be replaced as needed. Each membership consists of a singleton set. The read and update resilience conditions are identical here: Zero failures. Plugging these bounds into our liveness condition implies that, not surprisingly, a server must remain alive until a new one is installed. Moreover, the new server must be alive during its startup procedure. For example, say that initially we have M_0, which contains a singleton set of servers $\{q_0\}$. A client wishes to replace q_0 with an upgraded server q_1 by invoking **Reconfig**(M_1), with M_1 containing the set of servers $\{q_1\}$. Internally, the startup procedure of **Reconfig**(M_1) suspends q_0, copies data from q_0 to q_1, and redirects clients to q_1. Note that this startup procedure will be successful

under the assumptions of our liveness model, because both q_0 and q_1 remain alive while **Reconfig**(M_1) is in progress. Once q_1 stores the system state, the **Reconfig**(M_1) procedure completes and we model this as an abstract response event. From here on, q_0 may safely shut down.

The second example is a service implemented by $N = F+1$ servers for F-tolerance, such as a primary-backup setup. Here, the read and the update thresholds are substantially different. The update-threshold is $F + 1$. That is, in order for the service to store updates durably, it requires participation of all servers. In case of a failure, updates become stalled, and we use the reconfiguration manager to facilitate progress. Such a service must include another system component in charge of reconfiguration, because it impossible to form a consensus decision on the next membership among the $F + 1$ processes alone. To reconfigure, we need both one server to be available, in order to persist the service state; and the reconfiguration manager must be available in order to initiate the **Reconfig** procedure.

For example, say that M_0 has a set of servers S_0 and M_1 has S_1, each consisting of $F + 1$ servers. Upon invocation of **Reconfig**(M_1), our liveness requires read availability in both S_0 and S_1 and that some later membership has update availability. More specifically, our model says that at most F out of each of the sets S_0, S_1 fail. This suffices to suspend M_0 and transfer the closing state of M_0 to at least one server in S_1. When the closing state of M_0 has been transferred to all $F + 1$ servers of M_1 our model schedules the **Reconfig**(M_1) response event. At that time, our liveness model changes: it drops any resilience assumption on M_0. Indeed, it would be safe for all servers of S_0 (that are not in S_1) to shut down.

To re-iterate a point made above, **Reconfig** calls may overlap. Hence, this scenario could develop quite differently. Say that there is a failure in S_1 before we complete the state transfer to it. Hence, a client may issue **Reconfig**(M_2) to remedy this, before **Reconfig**(M_1) completes. In our formal model, another event occurs, marking the invocation of **Reconfig**(M_2) and changing the liveness assumption to 'F out of each of the sets S_0, S_1 and S_2 may fail'. Now servers in S_2 suspend M_1 and obtain its state. Note that our read-resilience assumption implies that this is possible, despite having failures in M_1 already. When all $F + 1$ servers in S_2 obtained their state, **Reconfig**(M_2) becomes completed. In this case, the completion of **Reconfig**(M_2) also pertinently marks the completion of **Reconfig**(M_1). Consequently, both M_0 and M_1 will be retired by this completion.

In our last example, a membership consists of a set of $N = 2F + 1$ servers, F of which may crash for either read or update. Here, the read and update resilience thresholds are identical. The liveness condition is simple here: During the lifetime of a membership, at most F of its servers may crash. A membership M_k ends with the completion of **Reconfig**(M_{k+1}), when the closing state of M_k is stored on $F + 1$ members of M_{k+1}. At that time, M_k may be retired.

Finally, we note that in order to reach a unique **Reconfig** decision, we are obviously bound by the impossibility of consensus: in order to guarantee termination for **Reconfig** decisions, we require an eventual leader with timely communication to a majority of the membership.

3 The Dynamic Reliable Multicast Problem

Let's jump in by exploring the reconfiguration question in the context of a simple but general form of reliable multicast. A reliable multicast protocol is simply a service (perhaps implemented by a library that its clients employ), which allows clients to *send* new multicast messages to groups of receivers, and to *receive* messages within groups. Multicast is a popular technology, both in the explicit form just described, and also in implicit forms, such as publish-subscribe or content-based communication infrastructures, so called *enterprise message bus* technologies, and many kinds of data replication and caching technologies. In three accompanying break-out boxes (Figures 1, 2, and 3) we discuss the possible mappings of our simple multicast API to more standard ones that might be used in such services.

Reliable multicast is a good place to start because many distributed systems rely on some form of multicast-like mechanism at a basic level, perhaps presenting it as a multicast, or perhaps hiding it in a data or file replication

How can our service model be made available through a more standard multicast package? Our multicast API is pull-based, but one could change it into a notification API by supporting an explicit group join operation. The join itself would change the group membership using a reconfiguration command, and we'll discuss these below. The Send operation maps directly to our **Add**() interface. Receive would then be supported as a callback from the multicast library into the application. When a process joins a multicast group for the first time, our durability rule requires it to learn the state of the group. Thus the first event that occurs upon joining would be a **Get**() of the sort we've included in our model: the **Get**(), in effect, embodies the state transfer. Subsequent receive operations, in contrast, would be modeled as **Get**() operations that omit messages that were previously delivered to the caller. View notification, if desired, would be implemented as an upcall that occurs when a new membership is initiated.

A real implementation would also garbage collect durable multicast messages once they are processed by the group members and reflected into the group state. The needed mechanisms complicate the protocol, but have no fundamental bearing on the reconfiguration problem, hence we omit discussion of them here. See [10] for details on methods for detecting durability and using that information to drive garbage collection.

Figure 1: Our simple multicast service abstracts classical ones.

mechanism. But whatever form the end-user functionality takes, the multicast communication pattern arises. If we can understand reconfiguration in a multicast protocol, we'll have taken a big step towards understanding reconfiguration as a general computing paradigm.

The multicast API. Our proposed service offers two interfaces to its clients, **Add**() and **Get**(). The **Add**() primitive sends a new message and returns an ACK. A **Get**() call returns a set of messages. A message m of a completed **Add**() or **Get**() operation becomes *durable*. The main requirement we have of a **Get**() call is:

Definition 1 (Multicast Durability) *A **Get**() call returns a set of messages that contains all messages which have become durable when the call was invoked.*

In practice, **Get**() returns all durable messages of completed **Add**() operations and possibly also some additional messages associated with concurrently executing **Add**() operations. If a message *m* is returned by **Get**, it becomes durable; hence every subsequent call to **Get** will also return *m*.

We don't require any particular order on operations; in particular, two concurrent calls to **Get**() may return disjoint sets of durable messages, and might order messages differently. We discuss this in more detail in Figure 2.

Epoch-by-epoch Solution. Our approach for dynamically reconfigurable reliable multicast has two parts: A *steady-state* protocol for sending and delivering messages during normal, stable periods; and a *reconfiguration* protocol. Each of these protocols is defined relative to a single *epoch*, which begins when a configuration of the system becomes live and ends by running the epoch termination protocol given below. After the current epoch ends, a new epoch starts, and we can understand it as hosting a completely new incarnation of this algorithm, whose newly added messages do not mix with the previous epoch. Indeed, when changing epochs a system could modify protocol parameters or even switch to a new protocol stack entirely incompatible with the prior one, reinitialize data, agree upon new security keys (in a secured group [29]), and so forth.

A Single Server Solution. It may be helpful to begin with a degenerate solution that employs just a single server, because by doing so, we create a form of reference implementation. Later when we explore distributed solutions, we can reason about their correctness by asking ourselves what properties it shares with the single server solution.

Multicast protocols and systems come in many forms and flavors. Variants may offer any of a wide range of message ordering policies, such as FIFO, causal, atomic (total), etc. Several of these policies can be implemented as a *filtering policy* over **Get**(). For example, a FIFO primitive would simply add a header to each message to represent ordering information (namely, the sender's sequence number). When **Get**() reports a durable message history to the application, the filtering policy would suppress previously delivered messages, and then sort undelivered messages, delivering them in sequence order, delaying any messages that follow a gap. Causal ordering can be done in the same manner: here, the sender includes a vector timestamp or some other representation of the causal dependency information, and **Get**() delays a given message until all causally prior multicasts have already been delivered. Total ordering is more complex; we discuss the issues and how they can be solved below, when we look closely at support for dynamically reconfigurable state machine replication.

Other multicast delivery primitives could implement delivery semantics beyond the model considered here. For our work here, we require durability, but many multicast systems support "early" delivery of non-durable messages, to reduce latency in situations where the application itself doesn't need durability. Failure could cause such messages to be lost, but there are applications that replicate various forms of soft or transient state and in which durability, for those updates, is unimportant, but minimizing latency is paramount (see the discussion of this topic in [11]). It is also possible to modify **Get**() to report non-durable messages, and the application could then manipulate its filtering policy to control which ones it will see. Doing this typically entails offering two versions of **Get**(), one with durability and one without. To give the application even more control, one can offer a **Flush**() primitive, which pauses until every multicast that was pending when it was invoked has become durable. None of these variants impose fundamentally different requirements on the underlying reconfiguration protocols, and hence for brevity, we will not explore them in detail.

Figure 2: Alternative delivery functionalities.

To implement **Add**(), the client in the single-server case locates the server and sends it a **store** request for whatever message m is being sent. The server adds m to the message history and acknowledges, and the client considers the **Add**() complete when the acknowledgment is received.

To implement **Get**(), the client contacts the server and the server sends back the entire current message history. Notice that this history will contain every completed **Add**(), and perhaps also a few more messages that were recently stored but for which the corresponding client hasn't yet received the acknowledgment. These requests are, in a technical sense, incomplete, although the associated messages will in fact be durable. In particular, notice that once a **Get**() includes m into a response, every future **Get**() will also include m.

Finally, how might we reconfigure the single-server implementation? The simplest solution would be as follows. A membership tracking module would receive a reconfigure command, specifying that henceforth, server s' will run the service. The membership service determines that s is currently running the service, and sends a message to s that causes it to enter what we will call a *wedged* state, meaning that no further operations will be accepted. Server s now transmits its final state to the membership service, which now sends s' a message that includes the final state. Having initialized itself from this transferred state, s' becomes operational, accepting new store operations. Meanwhile, s can terminate, discarding its state. The protocol, of course, isn't tolerant of failures: if s fails before the state is transferred, s' will be unable to enter the normal operational state.

Notice that we passed the new epoch state via the membership service. If this state is large, doing so might be undesirable, because during the period between when s transmits the state and when s' loads it, the service will be unavailable. However, there are ways to reduce this gap. For example, we might have s' speculatively copy the state from s using **Get**() operations, before the reconfiguration is even initiated. Now only the recent delta of messages that reached s subsequent to that preliminary transfer will be needed to bring s' into sync with s. Indeed, one could iterate, such that the membership service forms the new epoch, in which s' is the new server, only when

Readers familiar with protocols such as IP multicast, or gossip-based multicast, might be surprised to see durability even considered as a property that a multicast protocol should offer. In many settings, durability is viewed strictly as an end-to-end question, not something that belongs in a multicast layer. Moreover, there are many ways for an application to satisfy objectives such as durability. For example, while our discussion looks at multicasts that only deliver messages when they are safe in the sense of durable, there are some kinds of applications that operate optimistically, accepting messages instantly upon reception, but then rolling back if necessary to back out of unsafe states [28]. Why, then, are we treating durability as a multicast property?

Recall, however, that our overarching goal is to drill down on the question of *reconfiguring a durable service*, with the hope of teasing out essential aspects of the required solution. We're looking at multicast simply because multicast is the communication pattern underlying data replication, hence applications that perform durable data replication can be understood as if they were using a multicast for the data updates. By modeling the multicast protocol as the source of durability, we avoid needing to explicitly model the application, and can isolate the interplay between durability within the multicast protocol and reconfiguration of the multicast group membership.

Figure 3: Why should a multicast primitive offer durability?

the remaining delta is small enough; this way, if s' tries to join during a burst of activity, it is delayed slightly (during which it will continue to transfer chunks of state), until finally a brief moment of reduced load occurs, when s' can finally catch up relatively rapidly.

To use the terminology introduced above, we now have a fault-intolerant solution in which each epoch is associated with some single server, begins with the initialization of that server, runs in a steady state by storing messages, and ends when the next server takes over, wedging the previous one. The previous server can shut down entirely once the state transfer has been carried out.

Fault-Recovery versus Fault-Masking Steady State. We are now in a position to replicate our service to achieve such benefits as higher availability, load balancing of **Get**() operations over its members, etc. In what follows, we start by designing the steady-state protocols and only then consider the protocol needed to reconfigure. Two principal strategies suggest themselves:

1. The first is a *fault-recovery* approach, in which we store messages at *all* of the servers. If some server is unresponsive, this version will become blocked until a reconfiguration occurs.

2. The second is a *fault-masking* algorithm, in which we store messages at a majority of the servers. We deliver messages by reading from a majority and storing back messages at a majority. This version can continue to complete operations of both types as long as no more than a minority of servers become unresponsive. We need a majority of servers to remain available in order to transfer the state of the current configuration to any future one, but after reconfiguration, the old configuration may be retired completely.

In the coming two sections, we first flesh out a fault-recovery solution (Section 4), then a fault-masking one (Section 5).

4 Fault-Recovery Multicast

We start with the fault-recovery solution. The advantage of the a fault-recovery approach to the reliable multicast problem is that it requires just $N = F + 1$ servers and maintains data durability in the face of up to F failures. In case of a failure, we employ an auxiliary *consensus engine* to facilitate reconfiguration.

API:

Add(m): return(ACK)
Get$()$: return(S), such that:
if **Add**(m) completed before **Get**() was invoked, $m \in S$
if $S' =$ **Get**$()$ completed before invocation, $S' \subseteq S$
Reconfig(M): return(ACK)

Liveness condition:

throughout the *lifetime* of M, at least one server is correct; and
there exists a future membership M' in which all servers are correct throughout the *startup* of **Reconfig**(M')

Operation **Add**(m) at client:
send($\langle$**store**$, m\rangle$) to servers
wait for replies from all servers
return(ACK)

Operation **Get**$()$ at client:
send($\langle$**collect**$\rangle$) to servers
wait for reply S_q from each server q
return($\cap_q S_q$)

Upon $\langle$**store**$, m\rangle$ request at server and not wedged:
save m to local store and return ACK

Upon $\langle$**collect**$\rangle$ request at server and not wedged:
return all locally stored messages

Operation **Reconfig**(M):
Send($\langle$**wedge**$\rangle$) request to servers
Wait for reply $\langle$**suspended**$, S_q\rangle_q$
from any server q
Invoke consensus engine $decide(M, S_q)$
When all servers of new epoch have started
return(ACK)

Upon $\langle$**wedge**$\rangle$ request at server q:
stop serving **store/collect** commands
return $\langle$**suspended**$, S_q\rangle$ where
S_q contains all locally stored messages

At any server of new membership M'
Upon learning $(M', S) \leftarrow decide()$:
store S locally and start service

Figure 4: Single Epoch Fault-Recovery Reliable Multicast Solution

Figure 4 gives a succinct summary of the entire problem definition and its fault-recovery solutions. We elaborate further on them below.

4.1 Fault-Recovery Add/Get Implementation

Add: A client that wants to send a message m sends a **store** message containing m to all servers in a configuration. A server that receives the update inserts m to its local messages set (unless the server is wedged—see Reconfiguration below). Each server acknowledges the **store** message to the client, and the **Add** call completes when acknowledgments have been received from all servers.

Get: When a client invokes **Get**(), it sends a **collect** message to all servers in a configuration. A server that receives a collect command returns its locally stored messages to the client (again, unless it is wedged). The client waits to receive responses from all servers, and computes an intersection set S of message which appear in all of of these histories (message order is unimportant). The **Get** call returns the set S.

4.2 Reconfiguration Protocol

Recall that participants initiate reconfiguration by issuing a **Reconfig** command. Reconfiguration entails these steps:

1. The issuing client sends a *wedge* request to servers in the current epoch. As in our single-server solution, such a request causes a receiving server to become wedged (it may already be wedged from another **Reconfig** command), and to return a representation of its state.

2. Since as many as F servers could be faulty, the client waits for just one response, but this response will contain all durable messages (and possibly more). These messages are used for the initial state of each server in the next epoch.

3. **Reconfig** employs some kind of a consensus engine to form a decision both on the next membership M and on the set S of durable messages. For example, the consensus engine could be implemented by a centralized authority, which itself could be made reliable by running Paxos among replicated state machines [20].

4. Servers s' in a new membership M' learn the reconfiguration decision either directly from the auxiliary authority, or indirectly from other members. Either way, they learn both the membership M' of the new configuration and its initial message store S'. The initial state could also include application-specific information, or even specify an upgrade to a new version of the application or the protocols it uses. As in the single-server case, note that there are many ways to optimize state transfer so that the amount of information actually passed through the membership service could be quite small. What matters here is that a server s' in the new epoch should only process new requests after it has initialized itself appropriately.

   After completing state transfer, the server enables itself for handling normal client requests in the new epoch. Of course, if the server was also present in an earlier epoch, it remains wedged with respect to that epoch.

5. The **Reconfig** command is considered completed when all servers in the new configuration have enabled themselves. Note that **Reconfig** may never complete. In that case, yet a further reconfiguration command would be needed before system availability is restored.

The reader may wonder about reconfigurations that occur in response to a failure. In such cases, one or more servers in the current epoch will be unresponsive. However, any durable message will have been stored in all histories, and hence will be included in the initial state of all servers in the new epoch. On the other hand, consider an uncompleted operation, corresponding to a message stored in just a subset of the histories. At this stage, depending on the pattern of failures, that message could be missing from some of the surviving histories and dropped. But it could also turn out to be included in the history of the server whose state is used for the next configuration, in which case it would become durable even though the associated **Add**() operation may not have completed.

It isn't difficult to see that such problems are unavoidable. In effect, the outcome of a **Add**() that was pending at the time of a reconfiguration is determined by the membership service: if the message is included into the new epoch state, the **Add**() should be construed as successful, and if the message is not included, the **Add**() has failed and should be reissued in the new epoch. The client can learn this outcome from any server in the new epoch, or from the membership service itself. We leave the details of returning these out-of-band responses to clients out of the discussion here.

5 Fault-Masking Multicast

We continue with a fault-masking, majorities-based solution. The majorities-based fault-masking approach to the reliable multicast problem deploys $N = 2F + 1$ servers to maintain both data durability and non-disrupted

API:

Add(m): return(ACK)

Get$()$: return(S), such that:

if **Add**(m) completed before **Get**() was invoked, $m \in S$

if $S' =$ **Get**$()$ completed before invocation, $S' \subseteq S$

Reconfig(M): return(ACK)

Liveness condition:

throughout the lifetime of a membership M, a majority of servers are correct

---

Operation **Add**(m) at client:
send($\langle$**store**$, m\rangle$) to servers
wait for replies from a majority of servers
return(ACK)

Operation **Get**$()$ at client:
send($\langle$**collect**$\rangle$) to servers
wait for replies S_q from a majority of servers q
send($\langle$**store**$, \cup_q S_q\rangle$) to servers
wait for replies from a majority of servers
return($\cup_q S_q$)

Upon $\langle$**store**$, m\rangle$ request at server and not wedged:
save m to local store and return ACK

Upon $\langle$**collect**$\rangle$ request at server and not wedged:
return all locally stored messages

Operation **Reconfig**(M):
Send($\langle$**wedge**$\rangle$) request to servers
Wait for replies $\langle$**suspended**$, S_q\rangle_q$
from a majority of servers q
Invoke consensus engine $decide(M, \cup_q S_q)$
When a majority of servers in M have started
return(ACK)

Upon $\langle$**wedge**$\rangle$ request at server q:
stop serving **store/collect** commands
return $\langle$**suspended**$, S_q\rangle$ where
S_q contains all locally stored messages

At any server of new membership M'
Upon learning $(M', S) \leftarrow decide()$:
store S locally and start service

Figure 5: Majority-based Reliable Multicast Solution

operation in face of up to F failures. Figure 5 gives a succinct summary. For convenience, the differences from the Fault-Recovery approach of Figure 4 are highlighted.

5.1 Majorities-Based Tolerant Add/Get Implementation

The steady-state fault-masking solution for Reliable Multicast works as follows.

Add: A client that wants to send a message m sends a **store** message containing m to all servers in a configuration. A server that receives the update and is not wedged inserts m to its local messages set. Each server acknowledges the **store** message to the client, and the **Add** call completes when a majority of acknowledgments have been received.

Get: When a client invokes **Get**(), it sends a **collect** message to all servers in a configuration. A server that receives a collect command and is not wedged returns its locally stored messages to the client. The client waits to receive responses from a majority, and computes a union set S of message which appear in any of these histories. The client then stores back the set S at a majority by issuing a **store** message and waiting for acknowledgement from a

majority. [1] The **Get** call returns the set S.

5.2 Reconfiguration Protocol for Majorities-Based Multicast

The same reconfiguration protocol as the fault-recovery one can be used in this case, but with a minor modification to the rule used to compute the initial state of the new epoch.

Now, the issuing client must contact a majority of servers to discover all successful multicasts. Such a server enters the wedged state, and then sends the message history to the client, as above. This was why we required that, for the fault-masking case, the service include at least $2F + 1$ servers: if F fail, $F + 1$ will still be operational, and for any durable message, at least one of them will know of it. This, then, permits the client to include all durable messages in the initial state of the new epoch.

Again, after completing state transfer, a server of the new membership enables itself for handling normal client requests in the new epoch. However, it suffices for a majority of servers of the new configuration to become enabled for the **Reconfig** operation to be considered completed. And just as we saw above, state transfer can be optimized to transfer much of the data through an out-of-band channel, directly from the servers in the current epoch to the ones that will be members of the next epoch.

5.3 Reconfiguration Agreement Protocol

We now "open" the consensus engine and flesh out a procedure that unifies forming agreement on the next epoch with state transfer. Figure 6 summarizes a multicast solution that uses $N = 2F + 1$ servers and contains a detailed reconfiguration decision protocol combined with state transfer. (Only the **Reconfig** part is changed from Figure 5, but for completeness, Figure 6 gives a full solution.)

The combined reconfiguration and state transfer protocol is based on the well-known Synod algorithm [20]. It is triggered by a client **Reconfig** command and uses a set of $N = 2F + 1$ servers of the current epoch. The **Reconfig** procedure makes use of a uniquely chosen *stake* (for example, an integer or some other ordered type). Clients invoking reconfiguration may repeatedly try increasingly higher stakes until a decision is reached. The protocol of a particular stake has two phases. Although we could wait for the consensus decision and then perform state transfer, it turns out that we can make efficient use of message exchanges inside the protocol to accomplish state transfer at the same time.

More specifically, in **Phase 1**, a client performs one exchange with a majority of servers. When the client hears back from a majority, it learns:

(1) Either a reconfiguration command *RC*, which might have been chosen. In case of multiple possibly chosen *RC*'s, the one whose stake is highest is selected.

(2) Or that no reconfiguration command was chosen.

This exchange also tells the servers to ignore future proposals from any client that precedes it in the stake-order.

Coupled into this (standard) Phase 1 of consensus protocol is the collect phase of our state transfer. Namely, the same exchange marks the servers wedged, which is done by obtaining a commitment from the servers not to respond to any **store** or **collect** request from processes. In their responses in Phase 1, the client collects from each server the set of messages it stores.

In **Phase 2**, the client performs another single exchange with a majority of servers. If case (1) applies, then it tells servers to choose *RC*. Otherwise, in case (2), it proposes a new reconfiguration decision RC. The new RC contains the configuration that the client requested, as well as a a union of the messages it collected in the first phase.

[1] Clearly, we can optimize to store S only at sufficiently many servers to guarantee that S is stored at a majority.

The server's protocol is to respond to client's messages, unless it was contacted by a higher-stake client already:

- In phase 1, it responds with the value of a reconfiguration proposal *RC* of the highest-stake it knows of, or an empty *RC*. It also incorporates into the response the set of messages it stores locally, and commits to ignore future client **store/collect** requests.
- In phase 2, it acknowledges a client's proposal and stores it.

Each server in the new membership waits to collect **start** messages with the same stake from a majority of servers of the previous epoch. In this way, it learns about the next epoch decision, which includes the new configuration and the set of messages that are now durable. It stores these messages in its message history and becomes enabled for serving clients in the new epoch. Once a majority of servers in the new epoch are enabled, the **Reconfig**() operation completes.

As an example scenario, consider a system with three servers $\{1, 2, 3\}$. Assume message a reaches $\{1, 2\}$, b reaches $\{2, 3\}$ and c reaches server 3. During Phase 1 of reconfiguration, a client sends **wedge** messages to the servers. In response, server 2 suspends itself and sends $\{a, b\}$, and server 3 sends $\{b, c\}$. The client collects these responses, and enters phase 2, proposing a new epoch configuration consisting of server-set $\{4, 5, 6\}$ and message set $\{a, b, c\}$. Although this set contains more than the set of completed messages (c's **Add**() not having completed), those extra messages pose no problem; in effect, they complete and become durable as part of reconfiguration. Again, here we ignore the matter of sending out of band responses to the client which invoked **Add**(c).

Servers $\{4, 5\}$ in the new epoch each learns the decision, stores $\{a, b, c\}$ locally and become enabled. At this point, it is possible that all the servers in the previous epoch are shut down. This poses no risk, as no information is lost. In particular, every **Get**() request in the new epoch returns $\{a, b, c\}$.

Another possibility is that the client collects information from servers $\{1, 2\}$, and the reconfiguration decision includes only messages $\{a, b\}$. In this case, message c disappears. This is legitimate, as no **Add**() or **Get**() with c ever completes, in the past or in the future.

6 Coordinated State Transfer: The Virtual Synchrony Property

Consider what would happen in the Reliable Multicast service if we did not include in the reconfiguration decision the set S of messages ever completed in the current configuration. Instead, imagine a protocol in which every server in the new membership independently obtains the state from a read-set of the current configuration. For those messages M whose **Add** or **Get** have completed, there will be no difference. That is, every server in the new membership will obtain M before starting the new epoch. However, messages with partially completed **Add**/**Get** may or may not be obtained by the new servers, depending on which member(s) of the current configuration they query. Such messages could later become durable, by being transferred to all servers some epoch. Let's study this case in more detail by revising our fault-recovery approach.

For example, suppose that **Add**(a) has arrived at both servers $\{1, 2\}$ of some initial epoch. Say that **Add**(b) has reached $\{1\}$ so far, and **Add**(c) has reached $\{2\}$. Let the reconfiguration manager establish a decision on a new epoch set $\{3, 4, 5\}$. We may have server 3 suspend and pull messages $\{a, b\}$ from $\{1\}$. Note that message a was completed, and b may yet complete in the future; however, we cannot distinguish between these two situations. If servers $4, 5$ were to do the same state transfer, it would then be possible for a client to perform a **Get**() in the new epoch, and return $\{a, b\}$. This is the only safe response, as **Add**(b) could complete meanwhile.

Alternatively, servers $4, 5$ might pull $\{a, c\}$ from $\{2\}$. At this point, because both 1 and 2 are wedged and will not acknowledge further **store** requests, neither b nor c may ever complete in the current configuration. However, this situation cannot be distinguished by the servers. Hence, when a client requests **Get**() in the new epoch, server 3 must responds with $\{a, b\}$, while $4, 5$ must respond with $\{a, c\}$. This is fine, since the client will return the intersection $\{a\}$ in response to **Get**().

API:

Add(m): return(ACK)
Get$()$: return(S), such that:
if **Add**(m) completed before **Get**() was invoked, $m \in S$
if $S' =$ **Get**$()$ completed before invocation, $S' \subseteq S$
Reconfig(M): return(ACK)

Liveness condition:

throughout the lifetime of a membership M, a majority of servers are correct

---

Operation **Add**(m) at client:
send($\langle$**store**, $m\rangle$) to servers
wait for replies from a majority of servers
return(ACK)

Operation **Get**$()$ at client:
send($\langle$**collect**$\rangle$) to servers
wait for replies S_q from a majority of servers q
send($\langle$**store**, $\cup_q S_q\rangle$) to servers
wait for replies from a majority of servers
return($\cup_q S_q$)

Upon $\langle$**store**, $m\rangle$ request at server and not wedged:
save m to local store and return ACK

Upon $\langle$**collect**$\rangle$ request at server and not wedged:
return all locally stored messages

Operation **Reconfig**(M):
Choose unique $stake$
Send($\langle$**wedge**, $stake\rangle$) request to servers
Wait for replies $\langle$**suspended**, $stake, \langle st, RC\rangle, S_q\rangle_q$
from a majority of servers q
if any (st, RC) is non-empty
choose RC of highest (st, RC) pair;
else
let $RC \leftarrow (M, \cup_q S_q)$
Send($\langle$**accept**, $stake, RC\rangle$)
When a majority of servers in RC have started
return(ACK)

Upon $\langle$**wedge**, $st\rangle$ request at server q:
stop serving **store/collect** commands
unless accessed by higher-stake leader already
return $\langle$**suspended**, $st, \langle highst, highRC\rangle, S_q\rangle$

Upon $\langle$**accept**, $st, RC\rangle$) request at server q:
unless accessed by higher-stake leader already
store $highst \leftarrow st$, $highRC \leftarrow RC$
send $\langle$**start**, $st, RC\rangle$ to servers in RC

At any server in RC
Upon obtaining $\langle$**start**, $st, RC = (M, S)\rangle$
from a majority of previous epoch
store S locally and start service

Figure 6: Majority-based Reliable Multicast with $2F + 1$ Servers: Full Solution

The service remains correct with such 'uncoordinated state transfer'. However, in some future epoch, events might transpire in a way that causes all the servers to pull b and c from the preceding configuration. Suddenly, b and c will become durable, and all future **Get**() requests will include them. We believe that this behavior is undesirable. Indeed, in DSR we prevented such outcome by including in the consensus reconfiguration decision a set of messages from the current configuration, such that no other messages may appear later. The connection of the DRS approach to reconfiguration with the virtual synchrony approach to group communication may now become apparent to readers familiar with that literature: DRS guarantees to terminate operations within the lifetime of their invoking configuration. We now give a new formal definition of this old idea.

We'll start by explicitly modelling the configuration visible to a client when it invokes an operation. Notice that this is a reasonable addition to our model, since any client-side library must locate and interact with servers of the configuration, and hence it makes sense to rigorously specify the behavior of the associated interface. In addition to **Reconfig** invoke and response events, we model **Reconfig** *notification* events, which arrive at clients individually. Clients may be notified at different times about the same configuration.

Definition 2 *We say that a client invokes operation o* in configuration C *if C is the latest **Reconfig**-notification event at this client preceding o's invocation, or, in case no **Reconfig**-notification event has occurred, if C is the initial configuration C_0.*

For example, for some client, we may have the following sequence of events: invoke **Add**(m_1), response from **Add**(m_1), invoke **Add**(m_2), **Reconfig**(C_1) notification, response from **Add**(m_2), invoke **Add**(m_3), response from **Add**(m_3). In this sequence, the **Add**() operations of m_1 and m_2 are invoked in the initial configuration (C_0), and that of m_3 is invoked in C_1.

Now for Reliable Multicast, virtual synchrony requires the following:

Definition 3 (Virtually Synchronous Multicast) *If **Add**(m) was invoked in configuration C_k, then if ever **Get**() returns m, then for all configurations C_ℓ, where $\ell > k$, a **Get**() invoked in C_ℓ returns m.*

It is not hard to see that our Multicast reconfiguration strategy satisfies this condition, because whether or not m is stored in future configuration is determined by the reconfiguration decision itself.

For arbitrary services, we would consider operations which *have an effect* on others. In the Multicast case, **Add** operations change the behavior of future **Get** requests: **Get**() must return all previously sent messages. More generally, an operation o of type O which has an effect on operations o' of type O' changes the outcome of o' if the response event for o occurs before o' is invoked. Conversely, we say that o' *reflects* o.

The Virtual Synchrony guarantee implies the following:

Definition 4 (Virtual Synchrony) *Consider an operation o of type O which has an effect on operations of type O'. If o was invoked in configuration C_k, then if any operation o' of type O' reflects o, then for all configurations C_ℓ, where $\ell > k$, operations w' of type O' invoked in C_ℓ reflect o.*

7 Dynamic State Machine Replication and Virtually Synchronous Paxos

In this section, we take our Dynamic Service Replication (DSR) epoch-changing approach into the generic realm of State Machine Replication (SMR). In the state machine replication approach, replicas are implemented as deterministic state machines. The state machines start with a common initial state and apply commands in sequence order. This approach yields a protocol we refer to as *Virtually Synchronous Paxos*[2]. Paxos-users will recognize the close similarity of the protocol to the standard Paxos, yet in contrast with Reconfigurable Paxos and other reconfigurable SMR implementations, this solution achieves higher steady state performance with less complexity around the handling of reconfiguration.

7.1 On Paxos Anomalies

SMR operates on an infinite sequence of commands. Even though the sequence of commands is logically active all the time, in reality, systems progress in epochs: During each epoch, a fixed leader chooses a dense sequence of commands. Then, the leader is replaced, and a new epoch starts. The new leader starts where the previous leader stopped, and passes another subsequence of commands. And so on. If we envision the sequence of commands as a horizontal axis, then each leader-epoch lasts a contiguous segment along the horizontal axis. Nevertheless, abstractly, the same consensus protocol runs in the entire infinite sequence of commands, ignoring epoch and leader changes. Each instance vertically runs a succession of leaders, though each leader takes actual actions only in some instances, and none in others. The consensus-based point of view makes arguing correctness easy.

Unfortunately, the seemingly obvious intuition about epochs is wrong, and may lead to undesirable effects. Consider the scenario depicted in Figure 7. We have two Leaders U and V contending for the same pair of offsets, k and $k+1$. U proposes u_k at k and u_{k+1} and $k+1$, intending that u_k executes before u_{k+1}. V proposes v_k and v_{k+1} for these offsets. Consider the case where neither Leader U nor V obtains acceptance of a majority to their proposals. Hence, a third Leader W, starts after U and V are both retired. W finds traces of the command u_{k+1}, proposed by U, at offset $k+1$, and of the command v_k, proposed by V, at offset k. Paxos mandates W to re-propose v_k and u_{k+1} at k and $k+1$, respectively. If W is a stable leader, then indeed v_k and u_{k+1} will be chosen. But this violates the intended ordering constraint of the proposers!

In the above situation, Paxos permits the intended ordering of commands to be violated. As noted by the engineers who designed and built the Yahoo! ZooKeeper service [17], there are real-life situations in which leaders intend for their proposed $k+1$ command to be executed only if their proposed k command is done first. To achieve high performance many such systems are forced to pipeline their proposals, so a leader might propose command $k+1$ without waiting for a decision on command k. Paxos is quite capable of choosing $k+1$ but not k, and (as just noted) there are conditions under which it might choose both, but each one from a different leader. This particular case was sufficiently troublesome to convince the ZooKeeper engineers to move away from Paxos and to implement a virtually-synchronous, epoch-by-epoch type of leader election in their system.

So far, we have only discussed simple leader-changes. Our scenario turns out to have even more serious ramifications for previous reconfigurable SMR solutions. Recall that vanilla Paxos allows reconfiguration of the system that implements the state machine by injecting configuration-changing commands into the sequence of state machine commands. This is a natural use of the power of consensus inherent in the implementation of SMR. It entails a concurrency barrier on the steady-state command path: Suppose we form a reconfiguration command, for example, at index y in the sequence of commands. The command determines how the system will form the agreement decision on subsequent commands, for example, at index $y+1$ onward. More generally, the configuration-changing command at y may determine the consensus algorithm for commands starting at index

[2]This is a good juncture at which to note that Leslie Lamport was involved in an earlier stage of this research. Although he is not a co-author on this paper, his help in the initial formulation of our problem was invaluable, and this choice of protocol name is intended to acknowledge the strong roots of our protocol in his earlier work.

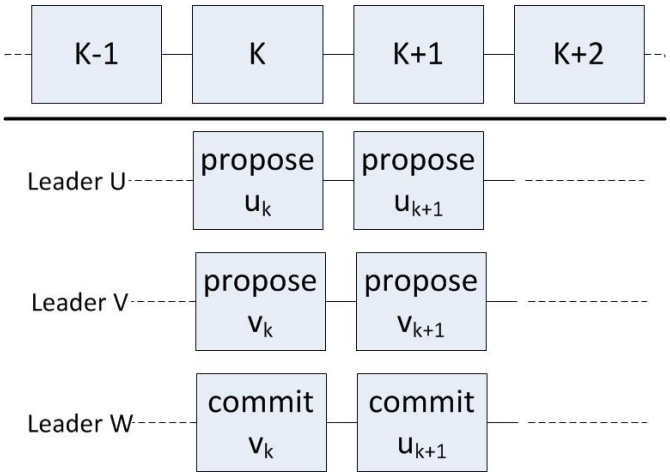

Figure 7: Anomalous Paxos behavior.

$y + \alpha$, for some pre-determined concurrency parameter α. We must wait for a reconfiguration decision at y to complete before we inject additional command requests to the system, at index $y + 1$ onward (or $y + \alpha$ in the general case). This barrier effectively reduces the system to one-by-one execution of commands, hence if reconfigurable Paxos is used, it would normally be deployed with α greater than 1 [23].

But, now consider the behavior of a Paxos-based SMR solution with $\alpha > 1$. Recall the scenario of contending leaders in Figure 7 above, and suppose that commands v_k and u_{k+1} are reconfiguration commands, which are mutually incompatible. This may cause the current configuration to issue two (or more) Reconfig commands, up to the maximum of α commands. Suppose that all of these were intended to apply to the configuration in they were issued. The first to be chosen will update the configuration as usual. But what are we to do when a second or subsequent command is chosen? These commands may no longer make sense. For example, a pending request to remove a faulty server from the configuration might be chosen after one that switches to a configuration in which that server is no longer a member. Executing these reconfigurations one after another is nonsensical. Likewise, a command to change a protocol parameter might be executed in a context where the system has reconfigured and is now using some other protocol within which that parameter has a different meaning, or no meaning at all.

If we use a window α larger than 1, then such events will be possible. An approach that seeks to prevent these commands from being chosen once they are no longer meaningful would require us to implement complex semantic rules. Allowing them to be chosen forces the application designer to understand that a seemingly "buggy" behavior is actually permitted by the protocol. In practice, many Paxos implementations (indeed, all that we know of) either do not support reconfiguration at all, or set α to 1, thus serializing command processing: only one command can be performed at a time. Batching commands can alleviate this cost: rather than one command at a time, the system might perform β at a time, for some parameter β. But this can help only to a limited degree.

Our remarks may come as a surprise to readers familiar with SMR and Paxos, because many published presentations of the model and protocols omit any discussion of the complexities introduced by reconfiguration. These remarks are not entirely new, and summarize several recent works which allude to these difficulties: The causality-violating scenario has been pointed out in [17], and Paxos reconfiguration idiosyncracies were discussed in [22].

7.2 Virtually Synchronous SMR

We now give a full solution to the dynamic SMR problem which avoids the above undesired behavior. As for the multicast service, we flesh out two approaches, a fault-recovery solution and a fault-masking one. The latter we

also call "Virtually Synchronous Paxos" because of its resemblance to the Paxos protocol.

We model an SMR service as providing clients with a **Submit**(op) API to atomically perform op on a shared object obj. Below, we denote by obj^k the object state after k operations. For durability, we store obj at a group of servers: $F+1$ in the fault-recovery approach, and $2F+1$ for fault-masking. We denote by obj_q the copy stored by server q. As usual, we provide a **Reconfig** API in order to reconfigure the service.

It is worth noting that our model is slightly different from the standard SMR model, in which there is a distributed engine for forming total order on commands and a separate one for execution. In most deployments, the same set of servers is used for both roles, and this simplifies our exposition.

Fault-Recovery Virtually Synchronous SMR

We begin with a fault-recovery approach, which utilizes only $F+1$ state machine servers for F-tolerance. As usual, this means that an auxiliary configuration engine is responsible for forming a consensus decision on the next configuration. This captures the classical primary-backup approach for replication, and extends it with a precise treatment of reconfiguration. Figure 9 describes the problem definition and gives a full solution in pseudo-code. We give one reconfiguration procedure for all cases, though specific scenarios may be further optimized, e.g., single backup failure, deploying a new secondary without failures, etc. Our formulation follows precisely the *Vertical Paxos* protocol [21], and we repeat it here for completeness.

The steady-state solution designates one server as *primary*. The primary obtains command requests from clients. For each request, it picks the next unused index k in the sequence of commands and requests the servers to accept the new command as the k'th. (If computing an operation is a heavy burden, the primary may opt to precompute the next state obj^k and send it to servers for efficiency, instead of sending the operation itself. This trick may also be used in case non-deterministic operations are to be supported.) A command is completed when the client received a (deterministic) response from all servers.

We put our DSR methodology to action in order to reconfigure state machines by having a reconfiguration decision that stops the sequence of state machine commands. The reconfiguration procedure first wedges (at least) one server. Here, the server's protocol is to respond to client's **wedge** messages with the latest object state and its index obj_q, k_q. The initiating client then lets the consensus engine form a decision on the next configuration. Crucially, the consensus decision contains the new configuration as well as the closing state of the current configuration. The decision value determines the configuration used in the next SMR instance and its initial object state. This decision effectively completes operations up to the k'th (and as usual, results of uncompleted operations are returned to their clients).

Once a decision is formed on both the new configuration and on its closing state, we can start the new state machine. However, in order for the new configuration to uphold linearizability [16], the sequence of commands in the new configuration must follow those of the current configuration. We achieve this by initializing the servers of the new configuration with the closing state of the object from the current configuration. Any server in the new membership M' that learns the decision (M', k_q, obj_q) can start the new configuration by (i) initializing its local state to k_q, obj_q and becoming enabled for normal command processing. Our liveness condition guarantees that the transfer can complete before reconfiguration is done to at least one server in the next configuration, and that some future configuration can become completely enabled. Accordingly, only upon receiving acknowledgement from all servers of that configuration, the **Reconfig** response event occurs.

Fault-Masking Virtually Synchronous SMR

We continue in Figure 9 with a fault-masking virtually synchronous SMR solution, which we call *Virtually Synchronous Paxos*. Although we could use the Paxos Synod protocol even in steady-state mode, we present a simpler version that uses a fixed primary. When the primary fails, we simply reconfigure to facilitate progress. Reconfiguring upon primary replacement in this manner alleviates the anomalous behavior related with primary transition in

API:

Submit(op): execute op atomically on object and return result $r = apply(op, obj)$
Reconfig(M): return(ACK)

Liveness condition:

throughout the *lifetime* of a membership M, one server is correct and
there exists a future membership M' in which all servers are correct throughout the *startup* of **Reconfig**(M')

Operation **Submit**(op) at client:
send $\langle op \rangle$ to designated primary
wait for responses r from all servers
return(r)

Upon $\langle op \rangle$ request at primary:
send $\langle$**submit**$, k, op\rangle$ to servers
increment k
possibly optimize:
$r^k \leftarrow apply(op, obj^{k-1})$
send $\langle$**submit**$, k, obj^k, r^k\rangle$

Upon $\langle$**submit**$, k, op\rangle$ request at server and not wedged:
let $r^k \leftarrow apply(op, obj^{k-1})$
store k, obj^k
send r^k to client

Operation **Reconfig**(M):
Send $\langle$**wedge**$\rangle$ request to servers
Wait for reply $\langle$**suspended**$, k_q, obj_q\rangle_q$
from any server q
Invoke consensus engine $decide(M, k_q, obj_q)$
When all servers of new epoch have started
return(ACK)

Upon $\langle$**wedge**$\rangle$ request at server q:
stop serving **submit** commands
return $\langle$**suspended**$, k_q, obj_q\rangle$

At any server in M'
Upon learning $(M', k_q, obj_q) \leftarrow decide()$:
initialize local k, obj from k_q, obj_q and start service

Figure 8: Fault-Recovery Virtually Synchronous SMR with $F + 1$ servers (Vertical Paxos)

Paxos pointed above in Figure 7. Leader election of our Virtually Synchronous Paxos within a fixed configuration is similar to the leader change protocol of ZooKeeper [17].

Steady state operation is done as follows. The primary obtains command requests from clients. For each request, it picks the next unused index k in the sequence of commands and requests the servers to accept the new command as the k'th. (The same optimization of pre-computing the resulting state is possible in case computing is expensive or non-deterministic, but not shown in the code.) A command is completed when the client received a (deterministic) response from majority of servers.

Recall that when reconfiguration is desired, we need to form agreement both on the next configuration and on the closing state of the current configuration. A client that wishes to reconfigure chooses a unique stake and performs the following two phases.

Phase 1: The client performs one exchange with a majority of servers. When it hears back from servers, it learns the latest computed state of the object obj^k known to servers. This reflects a prefix of the commands sequence proposed by the primary, whose tail may not have been completed yet. It also learns with respect to the configuration-changing decision either: (1) A reconfiguration command *RC* that might have been chosen, or (2) That no command was chosen. In this exchange, the client also obtains a commitment from the servers to ignore future messages from any lower stake client.

Phase 2: The client performs another single exchange with a majority of servers. If case (1) applies, then it tells

API:

Submit(op): execute op atomically on object and return result $r = apply(op, obj)$
Reconfig(M): return(ACK)

Liveness condition:

throughout the lifetime of a membership M, a majority of servers are correct

---

Operation **Submit**(op) at client:
send $\langle op \rangle$ to designated primary
wait for responses r from a majority of servers
return(r)

Upon $\langle op \rangle$ request at primary:
send $\langle$**submit**$, k, op \rangle$ to servers
increment k

Upon $\langle$**submit**$, k, op \rangle$ request at server and not wedged:
let $r^k \leftarrow apply(op, obj^{k-1})$
store k, obj^k
send r^k to client

Operation **Reconfig**(M):
choose unique $stake$
send $\langle$**wedge**$, stake \rangle$ request to servers
wait for replies $\langle$**wedged**$, (st, RC), k_q, obj_q \rangle$
from each server q in a majority
if any (st, RC) is non-empty
choose RC of highest (st, RC) pair;
else
let $RC \leftarrow (M, k_q, obj_q)$ of highest k_q
send $\langle$**accept**$, stake, RC \rangle$

when a majority of servers in RC have started
return(ACK)

Upon $\langle$**wedge**$, st \rangle$ request at server q:
stop serving **submit** commands
unless accessed by higher-stake leader already
return $\langle$**wedged**$, st_q, RC_q, k_q, obj_q \rangle$

Upon $\langle$**accept**$, st, RC \rangle$ request at server q:
unless accessed by higher-stake leader already
store st, RC
send $\langle$**start**$, st, RC \rangle$ to servers in RC

At any server in RC
Upon obtaining $\langle$**start**$, st, RC = (M, k_q, obj_q) \rangle$
from a majority of previous epoch
initialize obj state from RC and start service

Figure 9: Fault-Masking Virtually Synchronous SMR with $2F + 1$ servers (Virtually Synchronous Paxos)

servers to choose *RC*. Otherwise, in case (2), it proposes a new RC which contains the new membership and the closing state (k, obj^k). In either case, a client may propose only one reconfiguration for a particular stake.

The server's protocol is to respond to client's messages, unless it was instructed by a higher-stake client to ignore this client: In phase 1, server q responds with the latest object state obj_q; and with the value of a reconfiguration proposal *RC* of the highest-ranking client it knows of, or empty if none. In phase 2, it stores a client's proposal and acknowledges it in the form of a **start** message to the servers of the new configuration.

Any server in the new membership that learns the decision RC from a majority of servers in the current configuration (directly or indirectly) can start the new configuration by (i) initializing its local object to $RC.obj$ and becoming enabled for normal command processing. Our liveness condition guarantees that the transfer can complete before reconfiguration is done. Accordingly, only upon receiving acknowledgement from a majority of servers in *RC.M*, the **Reconfig** response event occurs.

8 Dynamic Read/Write Storage

In this section, we complement our arsenal of dynamic services with a solution to the dynamic Read/Write storage problem. A Read/Write Storage service provides two API methods, **Read** and **Write**, which execute atomically. This classical problem received much attention in the literature, starting with the seminal fault tolerant solution for static environments by Attiya et al. [5], and continuing with several recent storage systems for dynamic settings [24, 14, 12, 1, 33]. The full problem description and pseudo-code solution are given in Figure 10 below. The solution resembles the RDS protocol of Chockler et al. [12], but differs in that it maintains the virtual synchrony property and uses fewer phases. Although the solution approach is very similar to the Reliable Multicast one, we discuss it here for completeness. However, we only describe one flavor, a fault-masking protocol; the reader can complete the other variants based on the Multicast example. The fault-masking solution employs $2F + 1$ servers, each of which stores a copy of a shared object obj along with a logical update timestamp t. We denote by obj_q, t_q, the local copies stored at server q.

API:

Write(v): execute write(v) atomically on obj and return(ACK)
Read$()$: execute read() atomically on obj and return(u)
Reconfig(M): return(ACK)

Liveness condition:

throughout the lifetime of a membership M, a majority of servers are correct

---

Operation **Write**(v) at client:
send($\langle$**writequery**$\rangle$) to servers
wait for replies $\langle t_q \rangle$ from a majority of servers q
choose t greater than all t_q
send($\langle$**store**$, v, t\rangle$) to servers
wait for replies from a majority of servers
return(ACK)

Operation **Read**$()$ at client:
send($\langle$**collect**$\rangle$) to servers
wait for replies $\langle v_q, t_q \rangle$ from a majority of servers q
let $\langle v, t \rangle$ be the pair of highest timestamp
send($\langle$**store**$, v, t\rangle$) to servers
wait for replies from a majority of servers
return(v)

Upon $\langle$**writequery**$\rangle$ request at server q and not wedged:
return $\langle t_q \rangle$

Upon $\langle$**store**$, v', t'\rangle$ request at server q and not wedged:
if $t' > t_q$ save v', t' to obj_q, t_q
return ACK

Upon $\langle$**collect**$\rangle$ request at server q and not wedged:
return $\langle obj_q, t_q \rangle$

Operation **Reconfig**(M):
Choose unique $stake$
Send($\langle$**wedge**$, stake\rangle$) request to servers
Wait for replies $\langle$**suspended**$, stake, \langle st, RC\rangle, obj_q, t_q\rangle_q$
from a majority of servers q
if any (st, RC) is non-empty
choose RC of highest (st, RC) pair;
else
let $RC \leftarrow (M, obj_q, t_q)$ of highest t_q
Send($\langle$**accept**$, stake, RC\rangle$)
When a majority of servers in RC have started
return(ACK)

Upon $\langle$**wedge**$, st\rangle$ request at server q:
stop serving **store/collect/writequery** commands
unless accessed by higher-stake leader already
return $\langle$**suspended**$, st, \langle highst, highRC\rangle, obj_q, t_q\rangle$

Upon $\langle$**accept**$, st, RC\rangle$ request at server q:
unless accessed by higher-stake leader already
store $highst \leftarrow st$, $highRC \leftarrow RC$
send $\langle$**start**$, st, RC\rangle$ to servers in RC

At any server in RC
Upon obtaining $\langle$**start**$, st, RC = (M, obj_q, t_q)\rangle$
from a majority of previous epoch
store v, t, locally and start service

Figure 10: Majority-based Atomic Read/Write Storage with $2F + 1$ Servers: Full Solution

9 DSR in Perspective

Although our development portrays reconfiguration as a relatively linear task that entails making a series of seemingly straightforward decisions, there exist many incorrect or inefficient reconfiguration solutions in the published literature. In this section, we discuss the reasoning that can lead to complexity in dynamic membership protocols, and because of that complexity, expose the solution to potential bugs. As a first step it may be helpful to express our solution as a high-level recipe. Abstractly, the DSR epoch-by-epoch reconfiguration strategy entails the steps listed below. We have numbered the steps as either A.n or B.n to indicate that steps A/B may intermix in any order:

A.1 In the current configuration, form a **consensus** decision on the next configuration.

B.1 Suspend the current configuration from serving new requests (in progress requests may continue to completion, or die).

B.2 Take a snapshot of the current configuration's closing state. A legitimate (non-unique) snapshot must contain all operations *ever to complete*.

B.3 Form **agreement** on a legitimate closing state. In fault-recovery solutions, this agreement must be carried by the configuration manger, or in the next configuration (after the configuration manager has designated one). In a fault-masking solution, it may be carried by either the current configuration or the next one, but we must a priori designate which one has the responsibility.

A.2/B.4 Enable servers in the next configuration initialized with the consensus snapshot state formed in step B.3.

With this in mind, we pause to highlight some of the more common pitfalls that have traditionally led researchers (including the authors of the current paper) to propose less than ideal solutions.

9.1 Speculative-views

One design decision that we recommend against arises when a service is optimized for high availability by adding a mechanism that will copy state and enable activity in a new epoch that has been proposed, but not finalized, for example during intermediate steps of the agreement protocol. Much prior work in the group communication literature falls in this category [9, 3, 4, 35, 34, 6]; a significant body of PhD dissertations and published papers on *transient/ambiguous configurations*, *extended/weak virtual synchrony*, and others, were devoted to this strategy [2, 13, 18, 27]. Many other works in the group-communication area are covered in the survey by Chockler et al. [15].

The group communication approach essentially performs the task of steps A.1 and B.1-B.3 *inside a speculative next configuration*, in an attempt to optimize and transfer-state while forming agreement on it. This works roughly as follows.

A.1 In the current configuration, form a **proposal** on the next configuration.

A.2(B.1) Suspend those members of the next configuration which persist from the current one from serving new requests.

A.3(B.2-3) Among the members of the proposed next configuration, form agreement on both the transition itself, and on a legitimate snapshot of the current configuration's closing state. The snapshot decision incorporates input from those members which persist from the current configuration and from previously attempted *proposed* configurations.

A.4 If the next configuration fails to reach consensus, go back to step A.1.

A.5(B.4) Enable servers in the next configuration (their state is already initialized with the consensus snapshot state formed in step A.3.)

While the resulting solutions can certainly be implemented and proved correct, they require protocol steps in which the members of a new epoch collect information from every view in which operations might have been performed, so that the view that will ultimately be used has full knowledge of the service state. Such solutions can embody substantial complexity, and this of course means that implementations will often be quite hard to debug.

As an example scenario, let's revisit the scenario explored above, with an initial membership $\{1, 2, 3\}$. During reconfiguration, a client could propose a new epoch configuration consisting of server-set $\{2, 3, 4\}$. A typical complication here was for servers $\{2, 3, 4\}$ to form a "transient" configuration without waiting for a majority of the previous epoch to acknowledge it. They perform a state transfer and start serving client requests immediately. Thus, servers $\{3, 4\}$ might respond to a **Get**() request with a message-set containing $\{b, c\}$. In the end, we may form consensus on a different configuration, say $\{1, 2, 5\}$. If we only collect information from a majority of the previous epoch, say $\{1, 2\}$, we might "forget" message c. Note that we intentionally chose a new server-set that intersects with a majority of the members of the previous epoch. We did so to illustrate that there are situations in which a set large enough to reach agreement could nonetheless not suffice if a system needs to discover every durable message.

We therefore need to collect information from the transient configuration, so as not to lose message c. This is not only complex, but there is no guarantee that we will find traces of the transient epoch, unless we enforce additional constraints, the dynamic-quorum rule, as we now show.

9.2 Dynamic-quorums and Cascading changes.

The above scenario leads us to another common pitfall. In essence, it requires that cascading epoch changes jointly have a non-empty intersection. This guarantees that speculative epochs all intersect and become aware of one another. In the example above, server 2 is in the intersection of all attempted changes, and so we rely on information collected from it to discover the transient epoch, and collect information from its majority as well. More generally, a line of related works emanating from those with speculative configurations was devoted to the issue of handling "cascading reconfigurations", and to quorum intersection rules that guarantee to maintain a unique primary quorum through such cascading changes, e.g., [30, 37]. This constraint is unnecessary.

In our method, consensus is reached by a majority in the current epoch; the next epoch need not share any servers with the previous one. More generally, our epoch-by-epoch approach emphasizes that one only needs to deal with one epoch at a time, and that cascading or chained epochs just aren't necessary (nor do they have any performance or code complexity benefit; indeed, quite the opposite). The correct algorithmic foundation we establish terminates an epoch with a configuration-changing decision. It does not matter if we have a centralized configuration manager, or run a distributed consensus protocol; both are part of the current epoch, and it is well defined who determines the next configuration. The next epoch is uniquely determined in the current epoch, and transition to it is irreversible. When an epoch ends, a new epoch starts a new incarnation of the same algorithm (albeit with a non-empty initial service state), whose participants and messages do not mix with the current epoch. The new epoch may itself decide to reconfigure, and we iterate through epoch changes again. Any number of reconfigurations can be chained in this manner, with no difficulty at all.

In the example above, either we decide to transition to $\{2, 3, 4\}$ or not. If a decision is made, it is irreversible. A future decision to switch to $\{1, 2, 5\}$ must be taken by the new epoch, after it is enabled. It is achieved with the normal decision and state transfer mechanism. Conversely, if there is no decision on $\{2, 3, 4\}$, then no client operation executes in that epoch. Later, when a different decision is reached the new epoch starts with servers $\{1, 2, 5\}$.

9.3 Off-line Versus on-line reconfiguration

We need to comment briefly on our decision to use an off-line reconfiguration strategy. A recent line of works, pioneered by the RAMBO project [24] and continued in Rambo II [14], RDS [12], DynaStore [1] and DynaDisk [33], emphasize the importance of non-blocking reconfiguration. These systems tackle dynamic **Read/Write** storage services, but the same design issues occur in Reliable Multicast and other, similar services.

In the "RAMBO" approach, the solution is further optimized by mechanisms that prevent a client from ever encountering a suspended service. This *on-line* transition comes, however, at the cost of greater system complexity. RAMBO clients continue accessing the **Read/Write** objects by interacting both with the current and the next epoch. Every **Read** operation copies the object it accesses from one epoch to the next. Every **Write** operation also accesses a majority in the previous epoch, but no copying of data is necessary. Even if clients access objects which have already been copied to the new epoch (which would be the common case for "hot objects"), they still need to access majorities in *all* active configurations. The advantage is that RAMBO never blocks any client request, even momentarily, but the disadvantage is the complication of sorting through responses from two epochs and figuring out which one is more current.

Even more importantly, the on-line reconfiguration approach prevents a client from ever "sealing" partially-completed operations which were initiated in the current epoch. Traces of those may transfer to later epochs, and become durable arbitrarily far in the future. Such behavior may not be desirable for applications engineered with a Reliable Multicast (or Read/Write) infrastructure, as we saw earlier in our discussion of Virtual Synchrony (Section 6).

By comparison, the advantage of off-line reconfiguration is simplicity: clients only deal with a single system view at a time. We therefore see the off-line reconfiguration strategy above as occupying an attractive sweet-spot, despite the need to briefly block the service while switching from the old to the new configuration. On the other hand, off-line reconfiguration comes with its own form of complexity: the need to think hard about state transfer. If the service state might become large (for example, if the server is an entire file system or database) state transfer may entail moving a huge amount of data from the old to the new servers. Blocking the service while this occurs can be prohibitive, and even doing it rapidly may degrade the responsiveness of a service to an unacceptable degree. Thus our suggestion that state be transferred in advance of the reconfiguration, as much as possible, is important and may be key to using this approach in settings where service unavailability must be minimized.

By pre-transferring server state, the amount of state actually moved in the last steps of the procedure can be minimized and clients prevented from seeing much of a disruption. Moreover, the actual mechanism needed for this sort of anticipatory state transfer is fairly simple; the new server needs to fetch and load the state, and also needs some way to compute the delta between the loaded state and the initial state assigned to it in the new epoch, which will reflect operations that completed after it fetched that preloaded state and before the consensus decision was made.

9.4 Paxos Anomaly

The anomalous behavior we encountered when discussing Paxos reconfiguration illustrated a different kind of problem. Recall that vanilla Paxos is quite simple but runs in a fixed configuration. In contrast, reconfigurable Paxos introduced not just a leader-changing procedure, but also a set of secondary issues that represent side-effects of the way in which reconfiguration is supported. Although the per-command leader transition in Paxos entails a virtually-synchronous reconfiguration, the actions of a new leader are made independently for each command. In this way, leader W in our example above could not know that leader V's proposal to command $k+1$ implies that no command was committed by its predecessor, leader U, at command k.

Reconfigurable Paxos with $\alpha > 1$ indeed forms an orderly succession of configurations, but suffers from the same leader-change anomaly above within the window of α commands.

10 Related Work

Our primary contribution in this paper is to offer a unifying framework for the problem of dynamic service reconfiguration. We are not aware of any prior work on this question. Closest was the survey of Chockler, et al. [15], which offers a very interesting review of group communication systems in a single model. However, the main objective in that work was to contrast the guarantees offered by each category of solution and to categorize the weakest progress (liveness) assumptions made in each. In particular, the model used didn't formalize the reconfiguration topic as a separate mechanism, and hence doesn't get at the kinds of fine-grained choices, and their consequences, explored here.

Our epoch-by-epoch approach draws heavily on prior work on group communication. As mentioned, a good survey of group communication specifications appears in [15]. The approach resembles Virtual Synchrony [7, 11], but unlike our approach Virtual Synchrony does not provide linearizability. The approach also resembles Reconfigurable Paxos (as worked out in SMART [23]), but as pointed out in Section 7 this approach allows anomalous behavior. Approaches like Boxwood [25] and Chain Replication [36] allow reconfiguration with the help of an external configuration service.

There has also been considerable work on reconfigurable read/write stores, such as RAMBO [24], Dynamic Byzantine Storage [26], and DynaStore [1]. Our approach allows more general operations. Our liveness model is entirely based on that of DynaStore, however.

This review of prior work could cite a huge number of papers on specific multicast protocols, message-queuing middleware products, publish-subscribe products, enterprise service buses ands related technologies. However, we concluded that to do so would be tangential and perhaps even confusing to the reader. First, only some of these prior technologies offered durability, and among the ones that did (such as message-queuing middleware), some accomplished that goal using non-reconfigurable databases or logging servers. Thus, while the area is rich in prior work, the prior work on durability across reconfigurations is much scantier, and we believe the key works on that specific topic are precisely the ones we looked at closely in the body of this paper.

References

[1] M. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2009.

[2] Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Hebrew University of Jerusalem, September 1995.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malkhi. Transis: A communication subsystem for high availability. In *Proc. of the Twenty-2nd Int. Symp. on Fault-Tolerant Computing*, pages 76–84, Boston, MA, July 1992. IEEE.

[4] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *Trans. on Computer Systems*, 13(4):311–342, November 1995.

[5] H. Attiya, A. Bar Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121–132, 1995.

[6] Ö. Babaoğlu, R. Davoli, L. A. Giachini, and M. G. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proc. of the the 28th Hawaii Int. Conf. on System Sciences*, pages 612–621. IEEE, January 1995.

[7] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM Symposium on Operating Systems Principles*. ACM, 1987.

[8] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987. ACM Press.

[9] K.P. Birman. Replication and availability in the ISIS system. In *Proc. of the 10th ACM Symp. on Operating Systems Principles*, pages 79–86, Orcas Island, WA, December 1985.

[10] K.P. Birman. *Reliable Distributed Systems: Technologies, Web Services and Applications*. Springer-Verlag, 2005.

[11] K.P. Birman. A history of the Virtual Synchrony replication model. In B. Charron-Bost, F. Pedone, and A. Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, chapter 6, pages 91–120. Springer-Verlag, 2010.

[12] G. Chockler, S. Gilbert, V.C. Gramoli, P.M. Musial, and AA.A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *9th International Conference on Principles of Distributed Systems (OPODIS'05)*, December 2005.

[13] R. Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, Technion, 1994.

[14] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. of the 17th Intl. Symp. on Distributed Computing (DISC)*, pages 259–268, June 2003.

[15] R. Vitenberg G.V. Chockler, I. Keidar. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), December 2001.

[16] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463 – 492, 1990.

[17] F. Junqueira, P. Hunt, M. Konar, and B. Reed. The ZooKeeper Coordination Service (poster). In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[18] I. Keidar. *Consistency and High Availability of Information Dissemination in Multi-Processor Networks*. PhD thesis, Hebrew University of Jerusalem, October 1998.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[21] L. Lamport, D. Malkhi, and L. Zhou. Brief announcement: Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of Distributed Computing (PODC'09)*, pages 312–313, 2009.

[22] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. Technical report, Microsoft Research, 2009.

[23] 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 *Proc. of the 1st Eurosys Conference*, pages 103–115, Leuven, Belgium, April 2006. ACM.

[24] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of the 16th International Symposium on Distributed Computing*, pages 173–190, Toulouse, France, October 2002.

[25] J. MacCormick, N. Murphy, M. Najork, C.A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 105–120. USENIX, December 2004.

[26] J.-P. Martin and L Alvisi. A framework for dynamic byzantine storage. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pages 325–334, 2004.

[27] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, University of Bologna, Italy, 2000.

[28] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *Trans. on Computer Systems*, 23(4), November 2005.

[29] M. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *Trans. on Computer Systems*, 12(4):340–371, November 1994.

[30] A. Ricciardi and K.P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pages 341–351, Montreal, Quebec, August 1991. ACM SIGOPS-SIGACT.

[31] A.M. Ricciardi. *The Asynchronous Membership Problem in Asynchronous Systems*. PhD thesis, Cornell University, November 1992.

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

[33] A. Shraer, J.-P. Martin, D. Malkhi, and I. Keidar. Data-centric reconfiguration with network-attached disks. In *Large-Scale Distributed Systems and Middleware (LADIS 2010)*, July 2010.

[34] R. van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software—Practice and Experience*, August 1998.

[35] R. van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *CACM*, 39(4):76–83, April 1996.

[36] R. van Renesse and F.B. Schneider. Chain replication for supporting high throughput and availability. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

[37] E. Yeger-Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, pages 63–71, 1997.

A Correctness

Our goal in this section is to give a flavor of the correctness argument regarding our DSR epoch-by-epoch reconfiguration methodology. We view a formal specification and correctness proofs as being outside the scope of this paper. In this section, we provide sketch arguments on two sample solutions, the fault-recovery Reliable Multicast solution (Figure 4) and the fault-masking Reliable Multicast solution (Figure 6). The proof arguments for our other protocols, for SMR and for Read/Write storage, are similar in their core arguments concerning reconfiguration. They differ mostly in details regarding the specific problem models, namely, consensus and atomic Read/Write storage, which are well studied in the literature.

A.1 Correctness of Fault-Recovery Reliable Multicast Solution

In this section, we show that our fault-recovery protocol of Figure 4 maintains *Multicast Durability* (see Definition 1) and *Virtual Synchrony* (see Definition 3). Our proof shows that these conditions hold from one configuration to the next; proving that it holds for a sequence of reconfigurations follows by induction.

Refer to a current, initial configuration as C_0 and a new configuration as C_1. Recall that a message is durable if it belongs to a completed **Add**() or **Get**() operation.

Claim 1 *If a message m ever becomes durable by **Add**(m) or **Get**() operations in C_0, and a server q that learns about C_1 transfers to C_1 a set S_0 of messages, then $m \in S_0$.*

Proof: This claim is the core of the correctness claim. Although it is simple to prove given our algorithmic foundation, it is crucial to note that the set of messages sent/delivered in C_0 may continue changing after a decision on C_1 was made, and also subsequent to a state transfer. Nevertheless, the precise statement of this claim is in fact *stable*: Any message m which is **ever** sent/delivered in C_0 is included in S_0. We now prove it.

Since m is in a **Add**(m) or **Get**() which completes in C_0, every process in C_0 performs the $\langle$**store**$, m\rangle$ request and acknowledges it. Since wedged servers do not respond to **store** requests, q has responded to the **store** request before it became wedged for state transfer. Therefore, at step 2 of the state transfer, q already stores m, and includes it in the set S_0 which it transfers to C_1.

Claim 2 *If a message m becomes durable by **Add**(m) or **Get**() operations in C_0 and a subsequent **Get**() command completes at some process p, then p's response contains m.*

Proof: A **Get**() request which is subsequent to the completion of the **Add**(**Get**) command containing m may either occur in C_0 or in C_1. In C_0, every process already acknowledged $\langle$**store**$, m\rangle$. Hence, every server contacted by the requesting client will include m in the delivery response. In C_1, by Claim 1 above, every server also stores m by the time state transfer has completed. Therefore, **Get**() in C_1 also includes m.

Claim 3 *If a message m becomes durable by **Add**(m) or **Get**() operations in C_1 and a subsequent **Get**() command completes at some process p, then p's delivery response contains m.*

Proof: We first note that a **Get**() request that arrives at C_0 after m was sent/delivered in C_1 is not served by any server that participated in reconfiguration, because they are wedged; hence, the deliver request does not complete in C_0, and is deferred to C_1. In C_1, every process stores already acknowledged $\langle$**store**$, m\rangle$. Hence, every server contacted by the requesting client will include m in the delivery response.

Claim 4 *The fault-recovery protocol maintains Multicast Durability (Definition 1); that is, if a message m becomes durable by **Add**(m) or **Get**() operations, and a subsequent **Get**() command completes at some process p, then p's delivery response contains m.*

Proof: By Claims 2 and 3, after m is completely sent/delivered, a subsequent **Get**() request invoked at any process p includes m in the delivery response.

Claim 5 *The fault-recovery protocol maintains Multicast Virtual Synchrony (Definition 3); that is, If **Add**(m) was invoked in configuration C_k, then if ever **Get**() returns m, then for all configurations C_ℓ, where $\ell > k$, **Get**() returns m.*

Proof: By Claim 1, if **Get**() ever returns m in C_0, then m is included in the set S_0 of messages transferred to C_1. Hence, any **Get**() request in C_1 will return m.

Otherwise, suppose that no **Get**() in C_0 returns m, but there exists a **Get**() call in C_1 which returns m. Since m was sent to C_0, there exists a server q in C_1 which obtained m in a state transfer. But since state transfer passes a message-set determined by a consensus decision, every server in C_1 stores m before it becomes enabled for service in C_1. Hence, every **Get**() call in C_1 must return m.

We have shown that the claim holds for C_1. Using C_1 as the initial view, we obtain the result by induction for any subsequent view following C_1.

A.2 Correctness of Fault-Masking Reliable Multicast Solution

In this section, we show that our fault-masking protocol of Figure 6 maintains Multicast Durability and Virtual Synchrony. Our proof shows that these conditions hold from one configuration to the next; proving that it holds for a sequence of reconfigurations follows by induction.

Refer to a current, initial configuration as C_0 and a new configuration as C_1. Recall that a message is durable if it belongs to a completed **Add**() or **Get**() operation.

Claim 6 *If a message m ever becomes durable by **Add**(m) or **Get**() operations in C_0, and a client collects **suspended** responses to a* wedge *request from a majority of servers in C_0 into a set S_0 of messages, then $m \in S_0$.*

Proof: Since m is in a **Add**(m) or **Get**() which completes in C_0, a majority of servers in C_0 perform the $\langle$**store**, $m\rangle$ request and acknowledge it. Since wedged servers do not respond to **store** requests, every server in this majority has responded to the **store** request before it became wedged for state transfer. There exists a server q in the intersection of this majority and the majority of servers responding to the client *wedge* request. Therefore, q already stores m when it sends its **suspended** response to the client, and m is included in S_q. Since the client takes a union of the sets S_q contained in all the server responses, $m \in S_0$.

Claim 7 *If a message m ever becomes durable by **Add**(m) or **Get**() operations in C_0, then the set S_0 of the consensus decision on the next configuration contains m, i.e., $m \in S_0$.*

Proof: By Claim 6 above, any client which proposes input to the consensus engine regarding reconfiguration includes m in its proposed message-set. Hence, any decision S_0 contains m.

Claim 8 *If a message m becomes durable by **Add**(m) or **Get**() operations in C_0 and a subsequent **Get**() command completes at some process p, then p's response contains m.*

Proof: A **Get**() request which is subsequent to the completion of the **Add**(**Get**) command containing m may either occur in C_0 or in C_1.

In C_0, a majority of servers already acknowledged $\langle$**store**, $m\rangle$. There exists a server q in the intersection of this majority and the majority of servers responding to the client's *collect* request. Therefore, q already stores m when it sends its response to the client's collect request, and m is included in the response S_q. Since the client takes a union of the sets S_q contained in all the server responses, m is returned in the return-set of the **Get**() call.

In C_1, by Claim 7 above, a majority of servers also stores m by the time state transfer has completed. Therefore, **Get**() in C_1 also includes m by the same argument as in C_0.

Claim 9 *If a message m becomes durable by **Add**(m) or **Get**() operations in C_1 and a subsequent **Get**() command completes at some process p, then p's delivery response contains m.*

Proof: We first note that a **Get**() request that arrives at C_0 after m was sent/delivered in C_1 is not served by any server that participated in reconfiguration, because they are wedged; hence, the deliver request does not complete in C_0, and is deferred to C_1. In C_1, a majority of servers already acknowledged $\langle \textbf{store}, m \rangle$. Hence, by the same argument as in Claim 7 above, a **Get**() call in C_1 returns m.

Claim 10 *If a message m becomes durable by **Add**(m) or **Get**() operations, and a subsequent **Get**() command completes at some process p, then p's delivery response contains m.*

Proof: By Claims 8 and 9, after m is completely sent/delivered, a subsequent **Get**() request invoked at any process p includes m in the delivery response.

Claim 11 *The fault-masking protocol maintains Multicast Virtual Synchrony (Definition 3); that is, If **Add**(m) was invoked in configuration C_k, then if ever **Get**() returns m, then for all configurations C_ℓ, where $\ell > k$, **Get**() returns m.*

Proof: By Claim 6, if **Get**() ever returns m in C_0, then m is included in the set S_0 of messages transferred to C_1. Hence, any **Get**() request in C_1 will return m.

Otherwise, suppose that no **Get**() in C_0 returns m, but there exists a **Get**() call in C_1 which returns m. Since m was sent to C_0, there exists a server q in C_1 which obtained m in a state transfer. But since state transfer passes a message-set determined by a consensus decision, every server in C_1 stores m before it becomes enabled for service in C_1. Hence, every **Get**() call in C_1 must return m.

We have shown that the claim holds for C_1. Using C_1 as the initial view, we obtain the result by induction for any subsequent view following C_1.