Paxos Made Practical

David Mazières

1 Introduction

Paxos [3] is a simple protocol that a group of machines in a distributed system can use to agree on a value proposed by a member of the group. If it terminates, the protocol reaches consensus even if the network was unreliable and multiple machines simultaneously tried to propose different values. The basic idea is that each proposal has a unique number. Higher numbered proposals override lower-numbered ones. However, a "proposer" machine must notify the group of its proposal number before proposing a particular value. If, after hearing from a majority of the group, the proposer learns one or more values from previous proposals, it must re-use the same value as the highest-numbered previous proposal. Otherwise, the proposer can select any value to propose.

The protocol has three rounds. In the first round, the proposer selects a proposal number, $n > 0$. n's low-order bits should contain a unique identifier for the proposer machine, so that two different machines never select the same n. The proposer then broadcasts the message PREPARE(n). Each group member either rejects this message if it has already seen a PREPARE message greater than n, replies with PREPARE-RESULT(n', v') if the highest numbered proposal it has seen is $n' < n$ for value v', or replies with PREPARE-RESULT(0, nil) if it has not yet seen any value proposed.

If at least a majority of the group (including the proposer) accepts the PREPARE message, the proposer moves to the second round. It sets v to the value in the highest-numbered PREPARE-RESULT it received. If v is nil, it selects any value it wishes for v. The proposer then broad-

casts the message PROPOSE(n, v). Again, each group member rejects this message if it has seen a PREPARE(n'') message with $n'' > n$. Otherwise, it indicates acceptance in its reply to the proposer.

If at least a majority of the group (including the proposer) accepts the PROPOSE message, the proposer broadcasts DECIDE(n, v) to indicate that the group has agreed on value v.

A number of fault-tolerant distributed systems [1, 4, 8] have been published that claim to use Paxos for consensus. However, this is tantamount to saying they use sockets for consensus—it leaves many details unspecified. To begin with, systems must agree on more than one value. Moreover, in fault-tolerant systems, machines come and go. If one is using Paxos to agree on the set of machines replicating a service, does a majority of machines mean a majority of the old replica set, the new set, or both? How do you know it is safe to agree on a new set of replicas? Will the new set have all the state from the old set? What about operations in progress at the time of the change? What if machines fail and none of the new replicas receive the DECIDE message? Many such complicated questions are just not addressed in the literature.

The one paper that makes a comprehensive effort to explain how to use a Paxos-like protocol in a real system is Viewstamped Replication [6]. However, that paper has two shortcomings, the first cosmetic, the second substantive. First, Viewstamped Replication is described in terms of distributed transactions. As depicted in Figure 1, a system consists of *groups* of machines. Each group contains of one or more *cohorts*, which are machines that maintain replicas
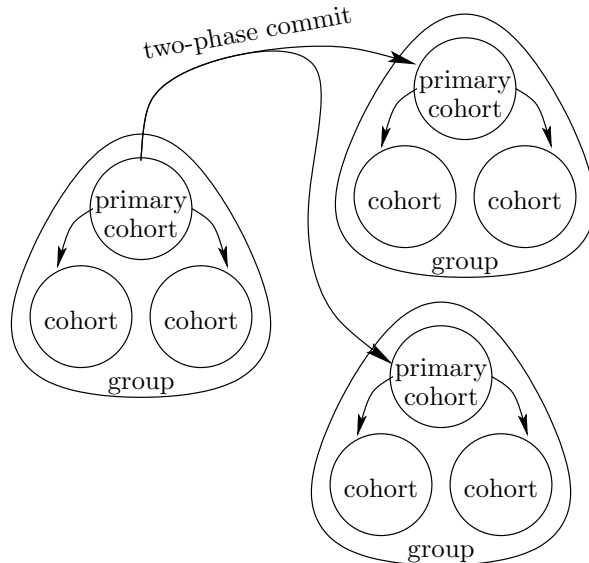
Figure 1: Overview of viewstamped replication

of the same objects. Different groups store different objects. Transactions span objects in multiple groups, with a two-phase commit used to commit transactions across groups. Distributed transactions add a great deal of complexity that not all applications need, making it harder to figure out how to replicate a simple system. Thus, in this paper, we concentrate on implementing a single group that replicates a state machine.

The second limitation of Viewstamped Replication is an assumption that the set of possible cohorts is fixed over time. The system requires participation of a majority of all *possible* cohorts, when it would be far better to require only a majority of all *active* cohorts. For example, if a group has five cohorts and two of them fail, the group will reconfigure itself to have three cohorts and continue to operate. However, if one more cohort fails before first two can be fixed, the group will also fail. This is unfortunate, since a majority of the three active cohorts is still functioning. More generally, for many applications it would be desirable to be able to add and remove cohorts from a group dynamically, for instance to facilitate migrating cohorts for maintenance or load-balancing purposes.

The remainder of this paper describes a practical protocol one can use to replicate systems for fault tolerance. Unlike other papers on replicated systems, it doesn't gloss over the details of how to use Paxos. It also overcomes a significant limitation of Viewstamped Replication and likely other Paxos-based systems.

2 State machine replication

We describe the protocol in terms of state-machine replication. A state machine is a deterministic service that accepts requests and produces replies. Because the service is deterministic, if two instances of the same state machine start in the same initial state and receive identical sequences of requests, they will also produce identical replies.

To make this more concrete, let us design a C++ interface that can be used between a service to be replicated and an implementation of the replication protocol described in this paper. A replication system provides two libraries, a server-side library, against which one links the service-specific state machine implementation, and a client-side library, which allows clients

to send requests to the state machine and get replies. For the interface, let `buf` be a C++ data structure implementing a variable-size array of bytes that can be grown or shrunk as needed.

The server-side replication library provides three functions:

```
id_t newgroup (char *path);
id_t joingroup (id_t group,
                char *path);
int run (char *path,
         sockaddr *other_cohort,
         buf (*execute) (buf));
```

The `newgroup` function initializes a new replicated state machine. The function creates a directory called `path`, and uses it to store persistent state for the group. `newgroup` must only be called once for each state machine you create. When `newgroup` returns, a new group exists with a single cohort, namely the machine on which `newgroup` was invoked. We assume that `id_t`, the type used for group names, is large enough that the probability of two `newgroup` invocations selecting the same group name is negligible.

When another cohort wishes to join a particular group, it must initialize its own state directory by calling the `joingroup` function. This creates the necessary files for the cohort to try to join `group`, but the cohort will not actually become a member of `group` until existing cohorts accept it into the group, as described later.

Finally, the bulk of the work takes place in the `run` function, which takes three arguments. `path` is the directory containing the state machine's files. `other_cohort` is an optional argument. If it is NULL, `run` will attempt to join the group by contacting the cohorts that existed the last time this cohort was running. In some cases, such as right after joining a group, the replication system won't know of other cohorts, in which case it is necessary to tell the library how to contact at least one cohort currently active in the group.

The final argument is a function pointer to a single function that implements the state machine being replicated. Each state machine must supply a function

```
buf execute (buf request);
```

that takes requests for the state machine and returns replies. Note that unless there is an error, `run` never returns; it just loops forever calling `execute`.

On the client side, the replication library provides a matching function to `execute`:

```
buf invoke (id_t group,
            sockaddr *cohort,
            buf request);
```

When a client calls `invoke`, the library, using the protocol described in this paper, attempts to call `execute` with the exact same `request` argument on every cohort in the group. If multiple clients call `invoke` concurrently, the replication system chooses the same execution order for requests on all cohorts. Because the service is deterministic, `execute` returns the same result on every cohort. `invoke` returns a copy of this result on the client. When invoking an operation in a group for the first time, it may be necessary to tell the client library how to contact members of the group. The `cohort` argument, if not NULL, can tell the library how to contact a member of the group.

One particularly easy way to implement an `execute` function is to implement a server using a remote procedure call (RPC) interface. An RPC library typically waits for a message from the network, decodes the procedure number and marshaled arguments, calls the appropriate C++ function for the procedure, and then marshals the results of the called function to be shipped back to the client. One can implement `execute` by just relaying the bytes for RPC requests and responses to and from an instance of the server (provided the server does not receive requests from any other source). `execute` can be in a small stand-alone program that talks to an unmodified server over local stream sockets. For efficiency, however, `execute` may be implemented as a special type of RPC transport in the same address space as the server.

Unfortunately, not all RPC servers are deterministic. For example, a file server may set the

modification time in a file's inode when it receives a write request. Even if two cohorts running the same file server code execute identical write requests, they will likely use different time values for the same write operation, thereby entering divergent states. This problem can be fixed, at the cost of some transparency, by having one cohort choose all the non-deterministic values for an operation and having all cohorts use those values. To do this we change `run`'s interface to take two function arguments:

```
buf choose (buf request);
buf execute (buf request,
             buf extra);
```

The fist function, `choose`, selects any non-deterministic values (such as the file modification time) that must be used to execute `request`, marshals them into a buffer, and returns the buffer. The second function, `execute`, now has a second argument, `extra`, that is the result of calling `choose`. For any given request, the library calls `choose` on one cohort and supplies its result as the `extra` argument to `execute` on all cohorts.

3   The setting

The rest of this paper describes a protocol that can be used to implement the replication system whose interface was presented in the last section. To be concrete about the protocol, we will describe the messages using Sun RPC's XDR (external data representation) language [7]. XDR types are similar to C data structures with a few exceptions. Angle-brackets designate variable-length arrays. For example,

```
cohort_t backups<>;
```

declares `backups` to be a variable length array of objects of type `cohort_t`. The special type `opaque` is used to designate bytes (which can only be declared in arrays), so that

```
opaque message<>;
```

declares `message` to be a variable-length array of bytes, which might be represented as the C++ `buf` type from the previous section. Type `hyper` designates a 64-bit integer. Structure declarations are like C. Unions are different, but we explain when they arise.

We define a few data types without specifying them, because the particular implementation does not really matter. For example, we assume every machine has a unique identifier of type `cid_t`, which can stand for cohort-id or client-id depending on the context. `cid_t` might be some centrally assigned value, or just a large enough array of bytes that if machines chose their identifiers randomly, the probability of collision is negligible. We also represent a machine's network address as `net_address_t`, which might, for instance, be an IP address and UDP port number.

There are two standard models for dealing with persistence in a replicated system. One is to treat reboots just like any other form of failure. Since the system's goal is to survive as long as a majority of cohorts do not fail simultaneously, we do not need to keep any persistent state when viewing reboots as failures. We are assuming a majority will always be up. In this model, a cohort gets a fresh `cid_t` after each reboot; we assume it knows nothing about past operations, except possibly as a bandwidth-saving optimization for re-synchronizing state. Note that view-stamped replication uses this model, but also requires a few forced disk writes because cohorts cannot choose fresh `cid_t`s upon reboot.

The other model is to assume that any machine may reboot at any time, and that as long as it doesn't lose its disk, this doesn't count as a failure. Given three replicas in this model, if all three are power-cycled simultaneously and one loses a disk in the process, the system can continue working as soon as the other two machines reboot. We will design our protocol to use this second model. When we say that a cohort logs information, we mean it does a forced write to disk before proceeding. It is easy to convert our protocol to the reboot-as-failure model by mak-
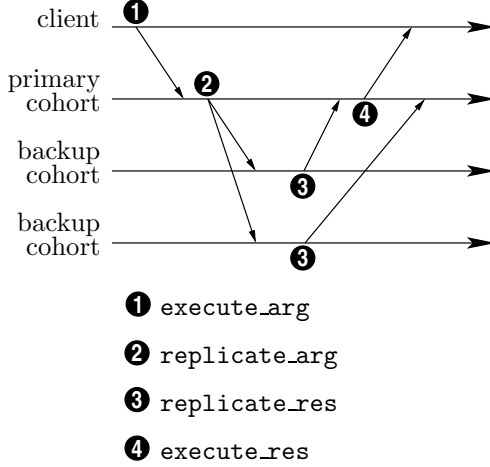
1 execute_arg

2 replicate_arg

3 replicate_res

4 execute_res

Figure 2: Messages sent during normal-case operation, when no cohorts fail or join the system.

ing all disk writes asynchronous.

4 Normal-case operation

In normal-case operation, when no cohorts fail or join the system, one cohort in a group is designated the primary while the others are backups, and all cohorts agree on this configuration of the group. We use the term *view* to denote a set of active cohorts with a designated primary. The system also assigns each view a unique *view-id*. As described in the next section, view-ids increase monotonically each time the group's view changes.

When the system executes a request, four types of message must be sent, as depicted in Figure 2. At a very high level:

1. The client sends its request to the primary cohort.

2. The primary cohort logs the request and forwards it to all other cohorts.

3. Cohorts log the operation and send an acknowledgment back to the primary.

4. Once the primary knows that a majority of cohorts (including itself) have logged the op-

eration, it executes the operation and sends the result to the client.

In the remainder of this section, we describe the contents of these messages in more detail.

Before initiating a request, the client must learn the group's current primary server and view-id. How it learns these is outside the scope of the protocol. One possibility is to rely on some external directory service; another is to discover the identity of a cohort on the local network through broadcast RPC. Once the client knows the identity of any cohort in the group, it can learn the group's primary and view-id from that cohort.

The client sends its request to the primary in the following message:

```
struct execute_arg {
  cid_t cid;
  uid_t rid;
  viewid_t vid;
  opaque request<>;
};
```

Here `cid` is the client's unique identifier. `rid` ("request id") is a unique identifier for this request. Its purpose is to allow the primary cohort to recognize duplicate requests and avoid re-executing them. `vid` is the current view-id. Its purpose is to prevent future primary cohorts, after view changes, from re-executing previous requests they may not have known about. `request` specifies the argument of the `execute` function.

The primary cohort numbers all requests it receives in a given view with consecutive *timestamp* values, starting at 1. Timestamps specify the order in which cohorts must execute requests within a view. Since view numbers are monotonically increasing, the combination of view-id and timestamp, which we call a *viewstamp*, determines the execution order of all requests over time. The primary server includes the viewstamp it assigns an operation when forwarding the operation to backups:

```
struct viewstamp_t {
```

```
    viewid_t vid;
    unsigned ts;
};

struct replicate_arg {
  viewstamp_t vs;
  execute_arg arg;
  opaque extra<>;
  viewstamp_t committed;
}
```

Here `extra` is any nondeterministic state to pass as the second argument of `execute`. The field `committed` specifies a viewstamp below which the server has executed all requests and sent their results back to clients. These committed operations never need to be rolled back and can therefore be executed at backups.

When executing a request, a cohort must persistently record the result of the last operation it has executed for each client, in case the primary's reply to the client gets lost and the primary subsequently fails. This does not require a forced disk write as long as the old state and log are kept persistently, so that the cohort can re-execute the request after a reboot. Cohorts can safely delete log entries before `committed`, although keeping old entries for a while makes it more efficient to synchronize state with another cohort that may have been partitioned from the network and missed a few operations.

As previously described, the primary assigns sequential timestamp (`ts`) values to viewstamps within a view. Backups use this field to ensure they have not missed any operations. A backup only logs a request after logging all previous operations, and only acknowledges a request once it has been logged. The backup acknowledges the request by viewstamp:

```
struct replicate_res {
  viewstamp_t vs;
};
```

Finally, after receiving acknowledgments from a majority of cohorts (including itself), the pri-mary calls the `execute` function on `request` and sends the reply back to the client:

```
struct execute_viewinfo {
  viewid_t vid;
  net_address_t primary;
};
union execute_res switch (bool ok) {
  case TRUE:
    opaque reply<>;
  case FALSE:
    execute_viewinfo viewinfo;
};
```

Here the RPC union syntax simply says that the result can be one of two types. If `ok` is true, all went well and the `reply` field contains the result of the `execute` function. If `ok` is false, the client either got the view-id wrong or sent the request to a cohort other than the primary; in this case the `viewinfo` field contains the current view-id and primary.

5 View-change protocol

At some point, one of the cohorts may suspect that another cohort has crashed because it fails to respond to messages. Alternatively, a new cohort may wish to join the system, possibly to replace a previously failed cohort. Either scenario calls for the group's membership to change. Since the trigger for a such a change may be the primary failing, any cohort that is part of the current view may decide to try to change the group's configuration by initiating a *view change*. This is a multi-step process like Paxos that involves first proposing a new view-id, then proposing the new view.

Figure 3 shows the state maintained by each cohort for keeping track of view changes. When a cohort updates this view-change state, it always record the new state to disk with a forced write before sending its next network message. We will discuss the meaning of the fields as we develop the protocol. However, we note that a

6

```
struct vc_state {
  enum {
    /* active in a formed view: */
    VC_ACTIVE,
    /* proposing a new view: */
    VC_MANAGER,
    /* invited to join a new view: */
    VC_UNDERLING
  } mode;

  /* last (or current) formed view: */
  view_t view;

  /* last committed op at any cohort*/
  viewstamp_t latest_seen;

  /* highest proposed new view-id: */
  viewid_t proposed_vid;

  /* accepted new view (if any): */
  view_t *accepted_view;
};
```

Figure 3: View-change state maintained by each cohort.

cohort only participates in the normal-case protocol of the last section when `mode` is VC_ACTIVE. If an execute or replicate RPC arrives when the cohort is in one of the other states, it ignores the request.

5.1 Proposing a new view-id

To initiate a view change, a cohort starts by proposing a new view-id. The cohort proposing the view-id is called the *view manager* for the new view. (The view manager should not be confused with the primary; if the view manager succeeds in forming a new view, it may or may not become the primary in that view.) The first step is to select a view-id greater than the highest one the view manager has ever seen (which it stores in `proposed_vid`).

A view-id consists of two fields:

```
struct viewid_t {
  unsigned hyper counter;
  cid_t manager;
};
```

Given two view-ids a and b, we say $a < b$ iff (a.counter $< b$.counter or (a.counter $= b$.counter and a.manager $< b$.manager)). The view manager selects a new view-id by incrementing `proposed_vid.counter` and setting `proposed_vid.manager` to its own cohort-id. By including the manager's cohort-id in the view-id, we ensure no two view managers ever propose the same view-id.

Figure 4 shows a typical instance of the view-change protocol. Once the view manager has selected a new view-id, it sends a *view change* RPC to all the other cohorts that are either in the current view or should join the new view. The arguments are:

```
struct cohort_t {
  cid_t id;
  net_address_t addr;
};
struct view_t {
  viewid_t vid;
  cohort_t primary;
  cohort_t backups<>;
};

struct view_change_arg {
  view_t oldview;
  viewid_t newvid;
};
```

Here `newvid` is the newly selected view-id, while `oldview` is the most recent successfully formed view that the view manager knows about. In `oldview`, `vid` is the view-id of the view, `primary` is the cohort-id and network address of the primary, while `backups` contains the cohort-ids and network addresses of the backups.

The old view-id in the view change RPC argument, `oldview.vid`, plays an important role. It identifies the *instance* of the Paxos protocol being run. Paxos is invoked once for each view

view-
manager
cohort

❶ ❸ ❺

underling
cohort

❷ ❹

underling
cohort

❷ ❹

underling/
new primary
cohort

❷ ❹

❶ `view_change_arg`

❷ `view_change_res`

❸ `new_view_arg`
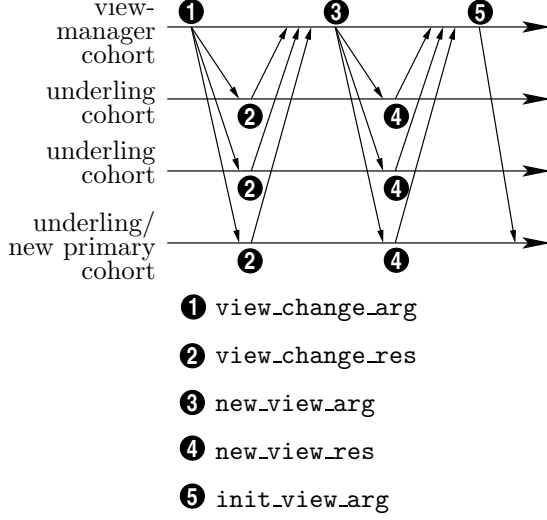
❹ `new_view_res`

❺ `init_view_arg`

Figure 4: Messages sent during a view change.

to agree on the configuration of the next view and ensure at most one such configuration can proceed. The view-id plays a dual role. It is the Paxos proposal number, but then, once a view is formed, becomes the identifier that names the next instance of Paxos.

We call cohorts that receive a view change request *underlings*, to distinguish them from the view manager that sent the RPC. When an underling receives a view change RPC, there are four possible cases to consider:

1. `oldview.vid` in the view change request is less than `view.vid` in the underling's state. Thus, at least one subsequent view has already successfully formed since the one the manager wants to change. The underling therefore rejects the view change RPC.

2. `oldview.vid` $\geq$ `view.vid`, but `newvid` in the request is less than `proposed_vid` in the underling's state. Thus, another manager has already proposed a higher new view-id. Again, the underling rejects the proposed view change, but updates `view` $\leftarrow$ `oldview` in its state if `view.vid` < `oldview.vid`.

3. `oldview.vid` = `view.vid` and `newvid` $\geq$ `proposed_vid`, so this is the highest new

view-id the underling has seen proposed. However, `accepted_view` is non-NULL, meaning the underling has already agreed to a particular configuration for the view following `oldview.vid`. (We note, however, that the underling can change its mind about this configuration if a majority of cohorts from `oldview` didn't agree on it.)

4. Either `oldview.vid` > `view.vid` (in which case the underling was not even aware of the last view), or else `oldview.vid` = `view.id` and `accepted_view` = NULL. This is like the previous case, except the underling never agreed to any particular configuration for the view after `oldview`.

The underling sends one of two reply types to a view change RPC, depending on whether it rejects or accepts the view change request:

```
struct view_change_reject {
  view_t oldview;
  viewid_t newvid;
};
struct view_change_accept {
  cit_t myid;
  bool include_me;
  viewstamp_t latest;
  view_t *newview;
};
union view_change_res
    switch (bool accepted) {
  case FALSE:
    view_change_reject reject;
  case TRUE:
    view_change_accept accept;
};
```

In cases 1 and 2, the underling rejects the view change RPC. It refuses to continue as an underling to this view manger, and sends back the latest successful view and proposed view-id that it knows about so that the failed view manager can update its stale state.

In cases 3 and 4, the underling accepts the view change RPC. It stops being an underling

8

to any other view manager, and it stops being a view manager itself if it had previously initiated a view change. The underling sends back its cohort-id in `accept.myid` and indicates if it wants to be included in the next view by setting `accept.include_me`. (For load-balancing purposes, a functioning cohort might want to leave a group.) The underling also sends back the viewstamp on the last entry in its log in `accept.latest` (not `latest_seen` in the state, which is only used by the view manager). In case 3, the underling includes the configuration it has agreed to for the next view by setting `*accept.newview` to `*accepted_view`. In case 4, the underling sets `accept.newview` to NULL, and updates its state so `view ← oldview` and `accepted_view ← NULL`.

5.2 Proposing a new view

The view manager collects replies to view change RPCs until it either aborts the view change, all cohorts reply, or, after a timeout, once it can form a new view. It aborts the view change if any of the underlings reject the view change request, or if it accepts a view change from a different view manager, thereby becoming an underling.

The view manager can form a new view V if 1) at least one cohort in V knows all past committed operations, and 2) no other view manager can form a view that executes operations concurrently with V. The first condition will be met if either a majority of cohorts in the previous view, V_{old}, are also in V, or if the primary from V_{old} is in V. The second condition can be met on two more conditions: 1) a majority of cohorts in V_{old} agree on the configuration of V, and 2) if a majority of V_{old} previously tried unsuccessfully to form a new view V', then V contains the same cohorts as V' and a majority of them agree V' never executed and never will execute a request.

Let us first consider the view manager's behavior in the simpler case that this is the first attempt to form a successor to view V_{old}. The view manager has sent a view change RPC to every other member of V_{old} and to any new cohorts

it wants to include in the next view. The RPC arguments have `oldview` set to V_{old}. Because the cohorts have not yet agreed on a successor view, all replies to the RPC will have a NULL `accept.newview`. Once all cohorts have replied, or a timeout has expired and the cohorts that did reply, together with the view manager, constitute a majority of V_{old}, the view manager goes about selecting the configuration of the new view.

The view manager first determines what cohorts new view V will include. It starts by including itself (if it wants to stay in the group) and all cohorts that replied with `accept.include_me` set to true. If this set of cohorts is not guaranteed to contain a member that knows all past operations, the view manager starts additionally conscripting cohorts that set `accept.include_me` to false. If the primary from V_{old} replied with `accept.include_me` false, the view manager includes it in the new view, which is sufficient. Otherwise, the view manager starts including cohorts in order of decreasing `accept.latest` fields until V contains a majority of V_{old}. (It can break ties arbitrarily.)

Next, the view manager selects a primary. If the primary from V_{old} is also in V, the view manager keeps the same primary. Otherwise, it selects the cohort in V with the highest `accept.latest` field, breaking ties first by preferentially selecting itself, then arbitrarily. The view manager sends the proposed new view V to all cohorts in V_{old} and V with a *new view* RPC. The arguments also include the highest `accept.latest` viewstamp reported by any cohort:

```
struct new_view_arg {
  viewstamp_t latest;
  view_t view;
};
```

If an underling has switched to a different view manager with a higher proposed `newvid`, it rejects the new view RPC. Otherwise, it compares `latest` to the viewstamp at the end of its own log, and if any operations are missing brings itself up to date by transferring the missing log entries

(or entire current state) from the proposed new primary. Finally, it replies:

```
struct new_view_res {
  bool accepted;
};
```

If the set of cohorts that has accepting the new view RPC grows to include both a majority of V_{old} and a majority of V, it is then safe to begin executing operations in view V. If the manager is not the primary, it notifies the new primary with an init view RPC:

```
struct init_view_arg {
  view_t view;
};
```

At this point, the new primary enters the VC_ACTIVE mode. It logs and broadcasts a special replicate_arg message to the backups of the new view, setting vs.ts to 0 and arg .request to the following structure:

```
struct init_view_request {
  view_t newview;
  viewstamp_t prev;
};
```

newview is the composition of the new view. prev is the viewstamp of the last committed operation in the previous view (to facilitate walking the log backwards). Recall that regular operations start with timestamp 1; thus backups recognize timestamp 0 as a special operation initializing the view. The backups log the operation, update view in their state, enter VC_ACTIVE mode, and reply. Once a majority of backups has replied, the new view has formed and it is safe for any cohorts that have dropped out of the group to erase their state (though they must poll the new primary to discover this).

If a view manager is not the first cohort to attempt to form a new group, it is possible that a previous view manager issued a new view RPC proposing view V', and that one or more of the cohorts accepted the new view. If a majority of cohorts accepted V', the view may even have formed. It is quite possible for the view manager not to know that V' has formed, particularly if there is little membership overlap between the old and new views. However, if a majority of V_{old} accepted V', and a majority of V_{old} also accepts the new view manager's view change RPC, then at least one view_change_res will contain a non-NULL accept.newview.

If the view manager receives a non-NULL accept.newview, this specifies a new view V' that could potentially have formed. If there are multiple non-NULL accept.newviews, the view manager considers only the one with the highest accept.newview->vid; let this potential view be V'. The view manager then send additional view change RPCs (with the same view_change_arg) to any cohorts in V' to which it didn't already send the RPC. The view manager must wait for both a majority of V_{old} and a majority of V' to accept this view change RPC.

When it comes time to form the new view, there are two possibilities to consider. Either the primary in V' has replied to the view change RPC, or it hasn't. If it has, the view manager chooses a new view V with exactly the same primary and backup cohorts as V'; only the view-id is different. It then proceeds with a new view RPC as before. If the primary in V' does not reply to the view change RPC, it may have failed, either before or after forming a new view. The view manager therefore constructs a new view V that has the same set of cohorts as V' (including the unresponsive primary), but for a primary it chooses the responsive node with the highest viewstamp in its latest field, then proceeds as before.

6   Optimizations

A number of optimizations can make the replication protocol described more efficient. As described, the protocol requires all requests to be broadcast to backup cohorts, including read-only operations. Otherwise, if the primary did not wait to hear from a majority of backups before

replying to read-only requests, it might be unaware that the remaining replicas had formed a new view (for instance because of a network partition), and therefore return stale data. This broadcast can be avoided through leases [2]. For example, if a majority of backups promise not to form a new view for 60 seconds, the primary can temporarily respond to read-only requests without involving backups.

Another optimization concerns hardware cost. As described, the system requires at least three full replicas to survive a failure, so that the remaining two constitute a majority. With only two replicas, neither can form a view on its own without running the risk that the other replica also concurrently formed a view. A solution is to employ a third machine, called a *witness* [5], that ordinarily sits idle without executing requests, but can participate in the consensus protocol to allow one of the other two replicas to form a view after a failure or network partition.

Acknowledgment

The author thanks Nickolai Zeldovich for his detailed feedback on the paper and contributions to the protocol.

References

[1] Mike Burrows. Chubby distributed lock service. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[2] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210. ACM, 1989.

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

[4] Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92. ACM, October 1996.

[5] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the harp file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 226–238, Pacific Grove, CA, October 1991. ACM.

[6] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[7] R. Srinivasan. XDR: External data representation standard. RFC 1832, Network Working Group, August 1995.

[8] Chandramohan Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, Saint-Malo, France, October 1997. ACM.