Viewstamped Replication Revisited

Barbara Liskov and James Cowling
MIT Computer Science and Artificial Intelligence Laboratory
liskov@csail.mit.edu, cowling@csail.mit.edu

Abstract

This paper presents an updated version of Viewstamped Replication, a replication technique that handles failures in which nodes crash. It describes how client requests are handled, how the group reorganizes when a replica fails, and how a failed replica is able to rejoin the group. The paper also describes a number of important optimizations and presents a protocol for handling reconfigurations that can change both the group membership and the number of failures the group is able to handle.

1 Introduction

This paper presents an updated version of Viewstamped Replication [11, 10, 8] (referred to as VR from now on). VR works in an asynchronous network like the Internet, and handles failures in which nodes fail by crashing. It supports a replicated service that runs on a number of *replica* nodes. The service maintains a state, and makes that state accessible to a set of *client* machines. VR provides *state machine replication* [4, 13]: clients can run general operations to observe and modify the service state. Thus the approach is suitable for implementing replicated services such as a lock manager or a file system.

The presentation here differs from the earlier papers on VR in several ways:

- The protocol described here improves on the original: it is simpler and has better performance. Some improvements were inspired by later work on Byzantine fault tolerance [2, 1].

- The protocol does not require any use of disk; instead it uses replicated state to provide persistence.

- The paper presents a reconfiguration protocol that allows the membership of the replica group to change, e.g., to replace a faulty node with a different machine. A reconfiguration can also change the group size so that the number of failures the group can handle can increase or decrease.

- The paper presents the protocol independently of any applications that use VR. The original papers explained the protocol as part of a database [11, 10] or file system [8], which made it difficult to separate the protocol from other details having to do with its use in an application.

VR was originally developed in the 1980s, at about the same time as Paxos [5, 6], but without knowledge of that work. It differs from Paxos in that it is a replication protocol rather than a consensus protocol: it uses consensus, with a protocol very similar to Paxos, as part of supporting a replicated state machine. Another point is that unlike Paxos, VR did not require disk I/O during the consensus protocol used to execute state machine operations.

Some information about the history of VR can be found in [7]. That paper presents a similar but less complete description of VR. It also explains how later work on Byzantine fault tolerance was based on VR and how those protocols are related to the VR protocols.

The remainder of the paper is organized as follows. Section 2 provides some background material and Section 3 gives an overview of the approach. The VR protocol is described in Section 4. Section 5 describes a number of details of the implementation that ensure good performance, while Section 6 discusses a number of optimizations that can further improve performance. Section 7 describes our reconfiguration protocol. Section 8 provides a discussion of the correctness of VR and we conclude in Section 9.

2 Background

This section begins with a discussion of our assumptions about the environment in which VR runs. Section 2.2 discusses the number of replicas required to provide correct behavior, and Section 2.3 describes how to configure a system to use VR.

2.1 Assumptions

VR handles crash failures: we assume that the only way nodes fail is by crashing, so that a machine is either functioning correctly or completely stopped. VR does not handle Byzantine failures, in which nodes can fail arbitrarily, perhaps due to an attack by a malicious party.

VR is intended to work in an asynchronous network, like the Internet, in which the non-arrival of a message indicates nothing about the state of its sender. Messages might be lost, delivered late or out of order, and delivered more than once; however, we assume that if sent repeatedly a message will eventually be delivered. In this paper we assume the network is not under attack by a malicious party who might spoof messages. If such attacks are a concern, they can be withstood by using cryptography to obtain secure channels.

2.2 Replica Groups

VR ensures reliability and availability when no more than a *threshold* of f replicas are faulty. It does this by using replica groups of size $2f + 1$; this is the minimal number of replicas in an asynchronous network under the crash failure model. The rationale for needing this many replicas is as follows:

We have to be able to carry out a request without waiting for f replicas to participate, since these replicas may be crashed and unable to reply. However, the f replicas we didn't hear from might merely be slow to reply, e.g., because of network congestion. f of the replicas we *did* hear from may thus subsequently fail. Therefore we need to run the protocol with enough replicas to ensure that even if these f fail, there is a least one replica that knows about the request. This implies that each step of the protocol must be processed by $f + 1$ replicas. These $f + 1$ together with the f that may not respond give us the smallest group size of $2f + 1$.

A group of $f + 1$ replicas is often referred to as a *quorum* and correctness of the protocol depends on the *quorum intersection property*: the quorum of replicas that processes a particular step of the protocol must have a non-empty intersection with the group of replicas available to handle the next step, since this way we can ensure that at each next step at least one participant knows what happened in the previous step. In a group of $2f + 1$ replicas, $f + 1$ is the smallest quorum size that will work.

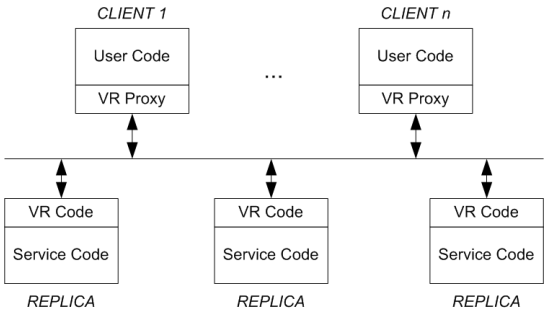

Figure 1: VR Architecture; the figure shows the configuration when $f = 1$.

In general a group need not be exactly of size $2f + 1$; if it isn't, the threshold is the largest f such that $2f + 1$ is less than or equal to the group size, K, and a quorum is of size $K - f$. However, for a particular threshold f there is no benefit in having a group of size larger than $2f + 1$: a larger group requires larger quorums to ensure intersection, but does not tolerate more failures. Therefore in the protocol descriptions in this paper we assume the group size is exactly $2f + 1$.

2.3 Architecture

The architecture for running VR is presented in Figure 1. The figure shows some client machines that are using VR, which is running on 3 replicas; thus $f = 1$ in this example. Client machines run the user code on top of the VR *proxy*. The user code communicates with VR by making operation calls to the proxy. The proxy then communicates with the replicas to cause the operation to be carried out and returns the result to the client when the operation has completed.

The replicas run code for the service that is being replicated using VR, e.g., a file system. The replicas also run the VR code. The VR code accepts requests from client proxies, carries out the protocol, and when the request is ready to be executed, causes this to happen by making an up-call to the service code at the replica. The service code executes the call and returns the result to the VR code, which sends it in a message to the client proxy that made the request.

Of the $2f + 1$ replicas, only $f + 1$ need to run the service code. This point is discussed further in Section 6.1.

3 Overview

State machine replication requires that replicas start in the same initial state, and that operations be deterministic. Given these assumptions, it is easy to see that replicas will end up in the same state if they execute the same sequence of operations. The challenge for the replication protocol is to ensure that operations execute in the same order at all replicas in spite of concurrent requests from clients and in spite of failures.

VR uses a *primary* replica to order client requests; the other replicas are *backups* that simply accept the order selected by the primary. Using a primary provides an easy solution to the ordering requirement, but it also introduces a problem: what happens if the primary fails? VR's solution to this problem is to allow different replicas to assume the role of primary over time. The system moves through a sequence of *views*. In each view one of the replicas is selected to be the primary. The backups monitor the primary, and if it appears to be faulty, they carry out a *view change* protocol to select a new primary.

To work correctly across a view change the state of the system in the next view must reflect all client operations that were executed in earlier views, in the previously selected order. We support this requirement by having the primary wait until at least $f + 1$ replicas (including itself) know about a client request before executing it, and by initializing the state of a new view by consulting at least $f + 1$ replicas. Thus each request is known to a quorum and the new view starts from a quorum.

VR also provides a way for nodes that have failed to recover and then continue processing. This is important since otherwise the number of failed nodes would eventually exceed the threshold. Correct recovery requires that the recovering replica rejoin the protocol only after it knows a state at least as recent as its state when it failed, so that it can respond correctly if it is needed for a quorum. Clearly this requirement could be satisfied by having each replica record what it knows on disk prior to each communication. However we do not require the use of disk for this purpose (and neither did the original version of VR).

Thus, VR uses three sub-protocols that work together to ensure correctness:

- Normal case processing of user requests.
- View changes to select a new primary.
- Recovery of a failed replica so that it can rejoin the group.

These sub-protocols are described in detail in the next section.

4 The VR Protocol

This section describes how VR works under the assumption that the group of replicas is fixed. We discuss some ways to improve performance of the protocols in Section 5 and optimizations in Section 6. The reconfiguration protocol, which allows the group of replicas to change, is described in Section 7.

Figure 2 shows the state of the VR layer at a replica. The identity of the primary isn't recorded in the state but rather is computed from the *view-number* and the *configuration*. The replicas are numbered based on their IP addresses: the replica with the smallest IP address is replica 1. The primary is chosen round-robin, starting with replica 1, as the system moves to new views. The *status* indicates what sub-protocol the replica is engaged in.

The client-side proxy also has state. It records the *configuration* and what it believes is the current *view-number*, which allows it to know which replica is currently the primary. Each message sent to the client informs it of the current *view-number*; this allows the client to track the primary.

In addition the client records its own *client-id* and a current *request-number*. A client is allowed to have just one outstanding request at a time. Each request is given a number by the client and later requests must have larger numbers than earlier ones; we discuss how clients ensure this if they fail and recover in Section 4.5. The request number is used by the replicas to avoid running requests more than once; it is also used by the client to discard duplicate responses to its requests.

4.1 Normal Operation

This section describes how VR works when the primary isn't faulty. Replicas participate in processing of client requests only when their *status* is *normal*. This constraint is critical for correctness as discussed in Section 8.

The protocol description assumes all participating replicas are in the same view. Every message sent from one replica to another contains the sender's current *view-number*. Replicas only process normal protocol messages containing a *view-number* that matches the *view-number* they know. If the sender is behind, the receiver drops the message. If the sender is ahead, the replica performs a *state transfer*: it requests information it is missing from the other replicas and uses this information to bring itself up to date before processing the message. State transfer is discussed further in Section 5.2.

The *request processing protocol* works as follows. The description ignores a number of minor details, such as re-sending protocol messages that haven't received responses.

- The *configuration*. This is a sorted array containing the IP addresses of each of the $2f + 1$ replicas.
- The *replica number*. This is the index into the configuration where this replica's IP address is stored.
- The current *view-number*, initially 0.
- The current *status*, either *normal*, *view-change*, or *recovering*.
- The *op-number* assigned to the most recently received request, initially 0.
- The *log*. This is an array containing *op-number* entries. The entries contain the requests that have been received so far in their assigned order.
- The *commit-number* is the *op-number* of the most recently committed operation.
- The *client-table*. This records for each client the number of its most recent request, plus, if the request has been executed, the result sent for that request.

Figure 2: VR state at a replica.

1. The client sends a ⟨REQUEST *op, c, s*⟩ message to the primary, where *op* is the operation (with its arguments) the client wants to run, c is the *client-id*, and s is the *request-number* assigned to the request.

2. When the primary receives the request, it compares the *request-number* in the request with the information in the client table. If the *request-number* s isn't bigger than the information in the table it drops the request, but it will re-send the response if the request is the most recent one from this client and it has already been executed.

3. The primary advances *op-number*, adds the request to the end of the *log*, and updates the information for this client in the *client-table* to contain the new request number, s. Then it sends a ⟨PREPARE *v, m, n, k*⟩ message to the other replicas, where v is the current *view-number*, *m* is the message it received from the client, n is the *op-number* it assigned to the request, and k is the *commit-number*.

4. Backups process PREPARE messages in order: a backup won't accept a prepare with *op-number n* until it has entries for all earlier requests in its *log*. When a backup i receives a PREPARE message, it waits until it has entries in its *log* for all earlier requests (doing state transfer if necessary to get the missing information). Then it increments its *op-number*, adds the request to the end of its *log*, updates the client's information in the *client-table*, and sends a ⟨PREPAREOK *v, n, i*⟩ message to the primary to indicate that this operation and all earlier ones have prepared locally.

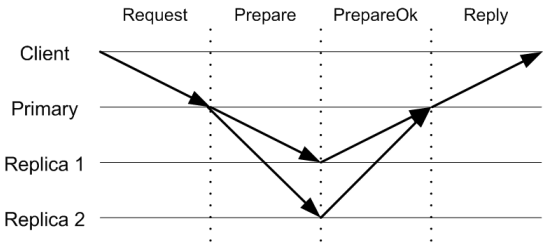

Figure 3: Normal case processing in VR for a configuration with $f = 1$.

5. The primary waits for f PREPAREOK messages from different backups; at this point it considers the operation (and all earlier ones) to be *committed*. Then, after it has executed all earlier operations (those assigned smaller *op-numbers*), the primary executes the operation by making an up-call to the service code, and increments its *commit-number*. Then it sends a ⟨REPLY *v, s, x*⟩ message to the client; here v is the *view-number*, s is the number the client provided in the request, and x is the result of the up-call. The primary also updates the client's entry in the *client-table* to contain the result.

6. Normally the primary informs backups about the commit when it sends the next PREPARE message; this is the purpose of the *commit-number* in the PREPARE message. However, if the primary does not receive a new client request in a timely way, it instead informs the backups of the latest commit by sending them a ⟨COMMIT *v, k*⟩ message, where k is *commit-number* (note that in this case *commit-number* = *op-number*).

7. When a backup learns of a commit, it waits until it has the request in its *log* (which may require state transfer) and until it has executed all earlier operations. Then it executes the operation by performing the up-call to the service code, increments its *commit-number*, updates the client's entry in the *client-table*, but does not send the reply to the client.

Figure 3 shows the phases of the normal processing protocol.

If a client doesn't receive a timely response to a request, it re-sends the request to all replicas. This way if the group has moved to a later view, its message will

reach the new primary. Backups ignore client requests; only the primary processes them.

The protocol could be modified to allow backups to process PREPARE messages out of order in Step 3. However there is no great benefit in doing things this way, and it complicates the view change protocol. Therefore backups process PREPARE messages in *op-number* order.

The protocol does not require any writing to disk. For example, replicas do not need to write the log to disk when they add the operation to the log. This point is discussed further in Section 4.3.

The protocol as described above has backups executing operations quickly: information about commits propagates rapidly, and backups execute operations as soon as they can. A somewhat lazier approach could be used, but it is important that backups not lag very far behind. The reason is that when there is a view change, the replica that becomes the new primary will be unable to execute new client requests until it is up to date. By executing operations speedily, we ensure that when a replica takes over as primary it is able to respond to new client requests with low delay.

4.2 View Changes

View changes are used to mask failures of the primary.

Backups monitor the primary: they expect to hear from it regularly. Normally the primary is sending PREPARE messages, but if it is idle (due to no requests) it sends COMMIT messages instead. If a timeout expires without a communication from the primary, the replicas carry out a view change to switch to a new primary.

The correctness condition for view changes is that every operation that has been executed by means of an up-call to the service code at one of the replicas must survive into the new view in the same order selected for it at the time it was executed. This up-call is performed at the old primary first, and therefore the replicas carrying out the view change may not know whether the up-call occurred. However, up-calls occur only for committed operations. This means that the old primary must have received at least f PREPAREOK messages from other replicas, and this in turn implies that the operation is recorded in the logs of at least $f + 1$ replicas (the old primary and the f backups that sent the PREPAREOK messages).

Therefore the view change protocol obtains information from the logs of at least $f + 1$ replicas. This is sufficient to ensure that all committed operations will be known, since each must be recorded in at least one of these logs; here we are relying on the quorum intersection property. Operations that had not committed might also survive, but this is not a problem: it is beneficial to have as many operations survive as possible.

However, it's impossible to guarantee that every client request that was preparing when the view change occurred makes it into the new view. For example, operation 25 might have been preparing when the view change happened, but none of the replicas that knew about it participated in the view change protocol and as a result the new primary knows nothing about operation 25. In this case, the new primary might assign this number to a different operation.

If two operations are assigned the same *op-number*, how can we ensure that the right one is executed at that point in the order? The solution to this dilemma is to use the *view-number*: two operations can be assigned the same number only when there has been a view change and in this case the one assigned a number in the later view prevails.

The *view change protocol* works as follows. Again the presentation ignores minor details having to do with filtering of duplicate messages and with re-sending of messages that appear to have been lost.

1. A replica i that notices the need for a view change advances its *view-number*, sets its status to *view-change*, and sends a ⟨STARTVIEWCHANGE *v, i*⟩ message to the all other replicas, where v identifies the new view. A replica notices the need for a view change either based on its own timer, or because it receives a STARTVIEWCHANGE or DOVIEWCHANGE message for a view with a larger number than its own *view-number*.

2. When replica i receives STARTVIEWCHANGE messages for its *view-number* from f other replicas, it sends a ⟨DOVIEWCHANGE *v, l, v', n, k, i*⟩ message to the node that will be the primary in the new view. Here v is its *view-number*, l is its log, v' is the view number of the latest view in which its status was *normal*, n is the *op-number*, and k is the *commit-number*.

3. When the new primary receives $f + 1$ DOVIEWCHANGE messages from different replicas (including itself), it sets its *view-number* to that in the messages and selects as the new *log* the one contained in the message with the largest v'; if several messages have the same v' it selects the one among them with the largest n. It sets its *op-number* to that of the topmost entry in the new *log*, sets its *commit-number* to the largest such number it received in the DOVIEWCHANGE messages, changes its *status* to *normal*, and informs the other replicas of the completion of the view change by sending ⟨STARTVIEW *v, l, n, k*⟩ messages to the other replicas, where l is the new log, n is the *op-number*, and k is the *commit-number*.

4. The new primary starts accepting client requests. It also executes (in order) any committed operations that it hadn't executed previously, updates its client table, and sends the replies to the clients.

5. When other replicas receive the STARTVIEW message, they replace their *log* with the one in the message, set their *op-number* to that of the latest entry in the log, set their *view-number* to the view number in the message, change their *status* to *normal*, and update the information in their *client-table*. If there are non-committed operations in the log, they send a ⟨PREPAREOK *v, n, i*⟩ message to the primary; here n is the *op-number*. Then they execute all operations known to be committed that they haven't executed previously, advance their *commit-number*, and update the information in their *client-table*.

In this protocol we solve the problem of more than one request being assigned the same *op-number* by taking the log for the next view from latest previous active view and ignoring logs from earlier view. VR as originally defined used a slightly different approach: it assigned each operation a *viewstamp*. A viewstamp is a pair ⟨*view-number, op-number*⟩, with the natural order: the *view-number* is considered first, and then the *op-number* for two viewstamps with the same *view-number*. At any *op-number*, VR retained the request with the higher viewstamp. VR got its name from these viewstamps.

A view change may not succeed, e.g., because the new primary fails. In this case the replicas will start a further view change, with yet another primary.

The protocol as described is expensive because the *log* is big, and therefore messages can be large. The approach we use to reduce the expense of view changes is described in Section 5.

4.3 Recovery

When a replica recovers after a crash it cannot participate in request processing and view changes until it has a state at least as recent as when it failed. If it could participate sooner than this, the system can fail. For example, if it forgets that it prepared some operation, this operation might then be known to fewer than a quorum of replicas even though it committed, which could cause the operation to be forgotten in a view change.

If nodes record their state on disk before sending messages, a node will be able to rejoin the system as soon as it has reinitialized its state by reading from disk. The reason is that in this case a recovering node hasn't forgotten anything it did before the crash (assuming the disk is intact). Instead it is the same as a node that has been unable to communicate for some period of time: its state is old but it hasn't forgotten anything it did before.

However, running the protocol this way is unattractive since it adds a delay to normal case processing: the primary would need to write to disk before sending the PREPARE message, and the other replicas would need to write to disk before sending the PREPAREOK response. Furthermore, it is unnecessary to do the disk write because the state is also stored at the other replicas and can be retrieved from them, using a *recovery protocol*. Retrieving state will be successful provided replicas are *failure independent*, i.e., highly unlikely to fail at the same time. If all replicas were to fail simultaneously, state will be lost if the information on disk isn't up to date; with failure independence a simultaneous failure is unlikely. If nodes are all in the same data center, the use of UPS's (uninterruptible power supplies) or non-volatile memory can provide failure independence if the problem is a power failure. Placing replicas at different geographical locations can additionally avoid loss of information when there is a local problem like a fire.

This section describes a recovery protocol that doesn't require disk I/O during either normal processing or during a view change. The original VR specification used a protocol that wrote to disk during the view change but did not require writing to disk during normal case processing.

When a node comes back up after a crash it sets its *status* to *recovering* and carries out the recovery protocol. While a replica's status is *recovering* it does not participate in either the request processing protocol or the view change protocol. To carry out the recovery protocol, the node needs to know the configuration. It can learn this by waiting to receive messages from other group members and then fetching the configuration from one of them; alternatively this information could be stored on disk.

The *recovery protocol* is as follows:

1. The recovering replica, i, sends a ⟨RECOVERY *i, x*⟩ message to all other replicas, where x is a nonce.

2. A replica *j* replies to a RECOVERY message only when its status is *normal*. In this case the replica sends a ⟨RECOVERYRESPONSE *v, x, l, n, k, j*⟩ message to the recovering replica, where v is its *view-number* and x is the nonce in the RECOVERY message. If j is the primary of its view, *l* is its *log*, n is its *op-number*, and k is the *commit-number*; otherwise these values are *nil*.

3. The recovering replica waits to receive at least $f + 1$ RECOVERYRESPONSE messages from different replicas, all containing the nonce it sent in its RECOVERY message, including one from the primary of the latest view it learns of in these messages. Then it updates its state using the information from the primary, changes its status to *normal*, and the recovery protocol is complete.

The protocol is expensive because *logs* are big and therefore the messages are big. A way to reduce this expense is discussed in Section 5.

If the group is doing a view change at the time of recovery, and the recovering replica, i, would be the primary of the new view, that view change cannot complete, since i will not respond to the DOVIEWCHANGE messages. This will cause the group to do a further view change, and i will be able to recover once this view change occurs.

The protocol uses the nonce to ensure that the recovering replica accepts only RECOVERYRESPONSE messages that are for this recovery and not an earlier one. It can produce the nonce by reading its clock; this will produce a unique nonce assuming clocks always advance. Alternatively, it could maintain a counter on disk and advance this counter on each recovery.

4.4 Non-deterministic Operations

State machine replication requires that if replicas start in the same state and execute the same sequence of operations, they will end up with the same state. However, applications frequently have non-deterministic operations. For example, file reads and writes are non-deterministic if they require setting "time-last-read" and "time-last-modified". If these values are obtained by having each replica read its clock independently, the states at the replicas will diverge.

We can avoid divergence due to non-determinism by having the primary predict the value. It can do this by using local information, e.g., it reads its clock when a file operation arrives. Or it can carry out a pre-step in the protocol in which it requests values from the backups, waits for f responses, and then computes the predicted value as a deterministic function of their responses and its own. The predicted value is stored in the log along with the client request and propagated to the other replicas. When the operation is executed, the predicted value is used.

Use of predicted values can require changes to the application code. There may need to be an up-call to obtain a predicted value from the application prior to running the protocol. Also, the application needs to use the predicted value when it executes the request.

4.5 Client Recovery

If a client crashes and recovers it must start up with a *request-number* larger than what it had before it failed. It fetches its latest number from the replicas and adds 2 to this value to be sure the new *request-number* is big enough. Adding 2 ensures that its next request will have a unique number even in the odd case where the latest request it sent before it failed is still in transit (since that request will have as its request number the number the client learns plus 1).

5 Pragmatics

The description of the protocols presented in the previous section ignores a number of important issues that must be resolved in a practical system. In this section we discuss how to provide good performance for node recovery, state transfer, and view changes. In all three cases, the key issue is efficient log management.

5.1 Efficient Recovery

When a replica recovers from a crash it needs to recover its log. The question is how to do this efficiently. Sending it the entire log, as described in Section 4.3, isn't a practical way to proceed, since the log can get very large in a long-lived system.

A way to reduce expense is to keep a prefix of the log on disk. The log can be pushed to disk in the background; there is no need to do this while running the protocol. When the replica recovers it can read the log from disk and then fetch the suffix from the other replicas. This reduces the cost of recovery protocol substantially. However the replica will then need to execute all the requests in the log (or at least those that modify the state), which can take a very long time if the log is big.

Therefore a better approach is to take advantage of the application state at the recovering replica: if this state is on disk, the replica doesn't need to fetch the prefix of the log that has already been applied to the application state and it needn't execute the requests in that prefix either. Note that this does not mean that the application is writing to disk in the foreground; background writes are sufficient here too.

For this approach to work, we need to know exactly what log prefix is captured on disk, both so that we obtain all the requests after that point, and so that we avoid rerunning operations that ran before the node failed. (Rerunning operations can cause the application state to be incorrect unless the operations are idempotent.)

Our solution to this problem uses *checkpoints* and is based on our later work on Byzantine-fault tolerance [2, 1]. Every O operations the replication code makes an upcall to the application, requesting it to take a checkpoint; here O is a system parameter, on the order of 100 or 1000. To take a checkpoint the application must record a snapshot of its state on disk; additionally it records a checkpoint number, which is simply the *op-number* of the latest operation included in the checkpoint. When it executes operations after the checkpoint,

it must not modify the snapshot, but this can be accomplished by using copy-on-write. These copied pages then become what needs to be saved to make the next snapshot and thus checkpoints need not be very expensive.

When a node recovers, it first obtains the application state from another replica. To make this efficient, the application maintains a Merkle tree [9] over the pages in the snapshot. The recovering node uses the Merkle tree to determine which pages to fetch; it only needs to fetch those that differ from their counterpart at the other replica. It's possible that the node it is fetching from may take another checkpoint while the recovery is in process; in this case the recovering node re-walks the Merkle tree to pick up any additional changes.

In rare situations where a node has been out of service for a very long time, it may be infeasible to transfer the new state over the network. In this case it is possible to clone the disk of an active replica, install it at the recovering node, and use this as a basis for computing the Merkle tree.

After the recovering node has all the application state as of the latest checkpoint at the node it is fetching from, it can run the recovery protocol. When it runs the protocol it informs the other nodes of the current value of its state by including the number of the checkpoint in its RECOVERY message. The primary then sends it the log from that point on.

As mentioned checkpoints also speed up recovery since the recovering replica only needs to execute requests in the portion of the log not covered by the checkpoint. Furthermore checkpoints allow the system to garbage collect the log, since only the operations after the latest checkpoint are needed. Keeping a larger log than the minimum is a good idea however. For example, when a recovering node runs the recovery protocol, the primary might have just taken a checkpoint, and if it immediately discarded the log prefix reflected in that checkpoint, it would be unable to bring the recovering replica up to date without transferring application state. A large enough suffix of the log should be retained to avoid this problem.

5.2 State Transfer

State transfer is used by a node that has gotten behind (but hasn't crashed) to bring itself up-to-date. There are two cases, depending on whether the slow node has learned that it is missing requests in its current view, or has heard about a later view. In the former case it only needs to learn about requests after its *op-number*. In the latter it needs to learn about requests after the latest committed request in its log, since requests after that might have been reordered in the view change, so in this case it sets its *op-number* to its *commit-number* and removes all entries after this from its log.

To get the state the replica sends a $\langle$GETSTATE *v, n', i*$\rangle$ message to one of the other replicas, where v is its *view-number* and n' is its *op-number*.

A replica responds to a GETSTATE message only if its *status* is *normal* and it is currently in view v. In this case it sends a $\langle$NEWSTATE *v, l, n, k*$\rangle$ message, where v is its *view-number*, l is its log after n', n is its *op-number*, and k is its *commit-number*.

When replica i receives the NEWSTATE message, it appends the log in the message to its log and updates its state using the other information in the message.

Because of garbage collecting the log, it's possible for there to be a gap between the last operation known to the slow replica and what the responder knows. Should a gap occur, the slow replica first brings itself almost up to date using application state (like a recovering node would do) to get to a recent checkpoint, and then completes the job by obtaining the log forward from the point. In the process of getting the checkpoint, it moves to the view in which that checkpoint was taken.

5.3 View Changes

To complete a view change, the primary of the new view must obtain an up-to-date log, and we would like the protocol to be efficient: we want to have small messages, and we want to avoid adding steps to the protocol.

The protocol described in Section 4.2 has a small number of steps, but big messages. We can make these messages smaller, but if we do, there is always a chance that more messages will be required.

A reasonable way to get good behavior most of the time is for replicas to include a suffix of their log in their DOVIEWCHANGE messages. The amount sent can be small since the most likely case is that the new primary is up to date. Therefore sending the latest log entry, or perhaps the latest two entries, should be sufficient. Occasionally, this information won't be enough; in this case the primary can ask for more information, and it might even need to first use application state to bring itself up to date.

6 Optimizations

This section describes several optimizations that can be used to improve the performance of the protocol. Some optimizations were proposed in the paper on Harp [8]; others are based on later work on the PBFT replication protocol, which handles Byzantine failures [2, 1].

6.1 Witnesses

Harp proposed the use of *witnesses* to avoid having all replicas actively run the service. The group of $2f + 1$ replicas includes $f+1$ *active* replicas, which store the application state and execute operations, and f witnesses, which do not. The primary is always an active replica. Witnesses are needed for view changes and recovery. They aren't involved in the normal case protocol as long as the $f + 1$ active replicas are processing operations. They fill in for active replicas when they aren't responding; however, even in this case witnesses do not execute operations. Thus most of the time witnesses can be doing other work; only the active replicas run the service code and store the service state.

6.2 Batching

PBFT proposed the use of batching to reduce the overhead of running the protocol. Rather than running the protocol each time a request arrives, the primary collects a bunch of requests and then runs the protocol for all of them at once.

The use of batching can be limited to when the primary is heavily loaded. This way it has little impact on latency. When the primary isn't busy it processes requests as soon as they arrive. When it is busy, it batches, but requests will be arriving frequently and therefore it needn't wait very long to collect the next batch. Batching can provide significant gains in throughput when the system is heavily loaded by amortizing the cost of the protocol over a large number of client requests.

6.3 Fast Reads

This section discusses two ways to reduce the latency for handling of read requests; both techniques additionally improve overall throughput.

6.3.1 Reads at the Primary

Harp proposed a way to improve performance for reads, by having the primary execute them without consulting the other replicas. This communication is not needed because read operations don't modify state and therefore need not survive into the next view.

However, having the primary perform read requests unilaterally could lead to the read returning a result based on an old state. This can happen if the request goes to an old primary that is not aware that a view change has occurred. For example, suppose a network partition has isolated the old primary from other replicas. Meanwhile a view change has happened and the new primary has executed more operations, so that the state at the old primary is stale.

To prevent a primary returning results based on stale data, Harp used leases [3]. The primary processes reads unilaterally only if it holds valid leases from f other replicas, and a new view will start only after leases at $f + 1$ participants in the view change protocol expire. This ensures that the new view starts after the old primary has stopped replying to read requests, assuming clocks rates are loosely synchronized.

In addition to reducing message traffic and delay for processing reads, this approach has another benefit: read requests need not run through the protocol. Thus load on the system can be reduced substantially, especially for workloads that are primarily read-only, which is often the case.

6.3.2 Reads at Backups

Leases aren't needed if it is acceptable for the result of a read to be based on stale information. In this case it also works to execute read requests at backups.

To support causality, there must be a way for a client to inform a backup of its previous operation. One way to do this is for the client to maintain a *last-request-number* in its state. When the client does a write, the primary returns the *op-number* that it assigned to that request, and the client stores this in *last-request-number*. When the client sends a read request it includes this number, and the replica responds only if it has executed operations at least this far; in its response it includes its *commit-number*, which the client stores in *last-request-number*.

The "reads at backups" provides a form of load balancing; effectively it allows the backups to be used as caches that are up to date enough to satisfy the causality requirements. However unlike the first approach ("reads at the primary") it does not provide external consistency. The first approach provides causality even in a setting where there are many storage repositories. In this setting, the second approach requires a different way of capturing causality, e.g., with vector timestamps [12] or Lamport clocks [4].

7 Reconfiguration

This section describes a reconfiguration protocol that allows the members of the group of replicas to change over time. Reconfiguration is needed to replace failed nodes that cannot be recovered, as well as for other reasons, e.g., to use more powerful computers as replicas, or to position the replicas in different data centers.

The reconfiguration protocol can also be used to change the threshold, f, the number of failures the group is intended to handle: the new group can be bigger or smaller than the old one. Changing the threshold is useful to allow the system to adapt to changing circum-

- The *epoch-number*, initially 0.
- The *old-configuration*, initially empty.

Figure 4: Additional state needed for reconfiguration.

stances, e.g., if experience indicates that more or fewer failures are happening than expected.

The approach to handling reconfigurations is as follows. A reconfiguration is triggered by a special client request. This request is run through the normal case protocol by the old group. When the request commits, the system moves to a new *epoch*, in which responsibility for processing client requests shifts to the new group. However, the new group cannot process client requests until its replicas are up to date: the new replicas must know all operations that committed in the previous epoch. To get up to date they transfer state from the old replicas, which do not shut down until the state transfer is complete.

7.1 Reconfiguration Details

To handle reconfigurations we add some information to the replica state, as shown in Figure 4. In addition there is another *status*, *transitioning*. A replica sets its status to *transitioning* at the beginning of the next epoch. New replicas use the *old-configuration* for state transfer at the start of an epoch; this way new nodes know where to get the state. Replicas that are members of the replica group for the new epoch change their status to *normal* when they have received the complete log up to the start of the epoch; replicas that are being replaced shut down once they know their state has been transferred to the new group.

Every message now contains an *epoch-number*. Replicas only process messages (from either clients or other replicas) that match the epoch they are in. If they receive a message with a later epoch number they move to that epoch as discussed below. If they receive a message with an earlier epoch number they discard the message but inform the sender about the later epoch.

Reconfigurations are requested by a client, c, e.g., the administrator's node, which sends a $\langle$RECONFIGURATION *e, c, s, new-config*$\rangle$ message to the current primary. Here e is the current *epoch-number* known to c, s is $c's$ *request-number*, and *new-config* provides the IP addresses of all members of the new group. The primary will accept the request only if s is large enough (based on the *client-table*) and e is the current *epoch-number*. Additionally the primary discards the request if *new-config* contains fewer than 3 IP addresses (since this is the minimum group size needed for VR). The new threshold is determined by the size of *new-config*: it is the largest value f' such that $2f' + 1$ is less than or equal to the size of *new-config*.

If the primary accepts the request, it processes it in the usual way, by carrying out the normal case protocol, but with two differences: First, the primary immediately stops accepting other client requests; the reconfiguration request is the last request processed in the current epoch. Second, executing the request does not cause an up-call to the service code; instead, a reconfiguration affects only the VR state.

The processing of the request happens as follows:

1. The primary adds the request to its log, sends a PREPARE message to the backups, and stops accepting client requests.

2. The backups handle the PREPARE in the usual way: they add the request to their log, but only when they are up to date. Then they send PREPAREOK responses to the primary.

3. When the primary receives f of these responses from different replicas, it increments its *epoch-number*, sends COMMIT messages to the other old replicas, and sends $\langle$STARTEPOCH *e, n, old-config, new-config*$\rangle$ messages to replicas that are being added to the system, i.e., those that are members of the new group but not of the old group. Here e is the new *epoch-number* and n is the *op-number*. Then it executes all client requests ordered before the RECONFIGURATION request that it hadn't executed previously and sets its *status* to *transitioning*.

Now we explain how the two groups move to the new epochs. First we explain the processing at replicas that are members of the new group; these replicas may be members of the old group, or they may be added as part of the reconfiguration. Then we explain processing at replicas that are being *replaced*, i.e., they are members of the old group but not of the new group.

7.1.1 Processing in the New Group

Replicas that are members of the replica group for the new epoch handle reconfiguration as follows:

1. When a replica learns of the new epoch (e.g., because it receives a STARTEPOCH or COMMIT message), it initializes its state to record the old and new configurations, the new *epoch-number*, and the *op-number*, sets its *view-number* to 0, and sets its *status* to *transitioning*.

2. If the replica is missing requests from its log, it brings its state up to date by sending state transfer messages to the old replicas and also to other new replicas. This allows it to get a complete state up to

the *op-number*, and thus learn of all client requests up to the reconfiguration request.

3. Once a replica in the new group is up to date with respect to the start of the epoch, it sets its *status* to *normal* and starts to carry out normal processing; it executes any requests in the log that it hasn't already executed and, if it is the primary of the new group, it starts accepting new requests. Additionally, it sends ⟨EPOCHSTARTED *e, i*⟩ messages to the replicas that are being replaced.

Replicas in the new group select the primary in the usual way, by using a deterministic function of the configuration for the new epoch and the current view number.

Replicas in the new group might receive (duplicate) STARTEPOCH messages after they have completed state transfer. In this case they send an EPOCHSTARTED response to the sender.

7.1.2 Processing at Replicas being Replaced

1. When a replica being replaced learns of the new epoch (e.g., by receiving a COMMIT message for the reconfiguration request), it changes its *epoch-number* to that of the new epoch and sets its *status* to *transitioning*. If the replica doesn't yet have the reconfiguration request in its log it obtains it by performing state transfer from other old replicas. Then it stores the current configuration in *old-configuration* and stores the new configuration in *configuration*.

2. Replicas being replaced respond to state transfer requests from replicas in the new group until they receive $f' + 1$ EPOCHSTARTED messages from the new replicas, where f' is the threshold of the new group. At this point the replica being replaced shuts down.

3. If a replica being replaced doesn't receive the EPOCHSTARTED messages in a timely way, it sends STARTEPOCH messages to the new replicas (or to the subset of those replicas it hasn't already heard from). New replicas respond to these messages either by moving to the epoch, or if they are already active in the next epoch, they send the EPOCHSTARTED message to the old replica.

7.2 Other Protocol Changes

To support reconfigurations, we need to modify the view change and recovery protocols so that they work while a reconfiguration is underway.

The most important change is that a replica will not accept messages for an epoch earlier than what it knows. Thus a replica will not accept a normal case or view change message that contains an old *epoch-number*; instead it informs the sender about the new epoch.

Additionally, in the view change protocol the new primary needs to recognize that a reconfiguration is in process, so that it stops accepting new client requests. To handle this case the new primary checks the topmost request in the log; if it is a RECONFIGURATION request, it won't accept any additional client requests. Furthermore, if the request is committed, it sends STARTEPOCH messages to the new replicas.

The recovery protocol also needs to change. An old replica that is attempting to recover while a reconfiguration is underway may be informed about the next epoch. If the replica isn't a member of the new replica group it shuts down; otherwise, it continues with recovery by communicating with replicas in the new group. (Here we are assuming that new replicas are *warm* when they start up as discussed in Section 7.5.)

In both the view change and recovery protocols, RECONFIGURATION requests that are in the log, but not in the topmost entry, are ignored, because in this case the reconfiguration has already happened.

7.3 Shutting down Old Replicas

The protocol described above allows replicas to recognize when they are no longer needed so that they can shut down. However, we also provide a way for the administrator who requested the reconfiguration to learn when it has completed. This way machines being replaced can be shut down quickly, even when, for example, they are unable to communicate with other replicas because of a long-lasting network partition.

Receiving a reply to the RECONFIGURATION request doesn't contain the necessary information since that only tells the administrator that the request has committed, whereas the administrator needs to know that enough new nodes have completed state transfer. To provide the needed information, we provide another operation, ⟨CHECKEPOCH *e, c, s*⟩; the administrator calls this operation after getting the reply to the RECONFIGURATION request. Here c is the client machine being used by the administrator, s is $c's$ *request-number*, and e is the new epoch. The operation runs through the normal case protocol in the new group, and therefore when the administrator gets the reply this indicates the reconfiguration is complete.

It's important that the administrator wait for the reconfiguration to complete before shutting down the nodes being replaced. The reason is that if one of these nodes were shut down prematurely, this can lead to more than f failures in the old group before the state has been transferred to the new group, and the new group would then

be unable to process client requests.

7.4 Locating the Group

Since the group can move, a new client needs a way to find the current configuration. This requires an out-of-band mechanism, e.g., the current configuration can be obtained by communicating with a web site run by the administrator.

Old clients can also use this mechanism to find the new group. However, to make it easy for current clients to find the group, old replicas that receive a client request with an old epoch number inform the client about the reconfiguration by sending it a ⟨NEWEPOCH *e, v, new-config*⟩ message.

7.5 Discussion

The most important practical issue with this reconfiguration protocol is the following: while the system is moving from one epoch to the next it does not accept any new client requests. The old group stops accepting client requests the moment the primary of the old group receives the RECONFIGURATION request; the new group can start processing client requests only when at least $f' + 1$ new replicas have completed state transfer.

Since client requests will be delayed until the move is complete, we would like the move to happen quickly. However, the problem is that state transfer can take a long time, even with our approach of checkpoints and Merkle trees, if the application state is large.

The way to reduce the delay is for the new nodes to be warmed up by doing state transfer *before* the reconfiguration. While this state transfer is happening the old group continues to process client requests. The RECONFIGURATION request is sent only when the new nodes are almost up to date. As a result, the delay until the new nodes can start handling client requests will be short.

8 Correctness

In this section we provide an informal discussion of the correctness of the protocol. Section 8.1 discusses the correctness of the view change protocol ignoring node recovery; correctness of the recovery protocol is discussed in Section 8.2. Section 8.3 discusses correctness of the reconfiguration protocol.

8.1 Correctness of View Changes

Safety. The correctness condition for view changes is that every committed operation survives into all subsequent views in the same position in the serial order. This condition implies that any request that had been executed retains its place in the order.

Clearly this condition holds in the first view. Assuming it holds in view v, the protocol will ensure that it also holds in the next view, v'. The reasoning is as follows:

Normal case processing ensures that any operation o that committed in view v is known to at least $f + 1$ replicas, each of which also knows all operations ordered before o, including (by assumption) all operations committed in views before v. The view change protocol starts the new view with the most recent log received from $f + 1$ replicas. Since none of these replicas accepts PREPARE messages from the old primary after sending the DOVIEWCHANGE message, the most recent log contains the latest operation committed in view v (and all earlier operations). Therefore all operations committed in views before v' are present in the log that starts view v', in their previously assigned order.

It's worth noting that it is crucial that replicas stop accepting PREPARE messages from earlier views once they start the view change protocol (this happens because they change their status as soon as they learn about the view change). Without this constraint the system could get into a state in which there are two active primaries: the old one, which hasn't failed but is merely slow or not well connected to the network, and the new one. If a replica sent a PREPAREOK message to the old primary after sending its log to the new one, the old primary might commit an operation that the new primary doesn't learn about in the DOVIEWCHANGE messages.

Liveness. The protocol executes client requests provided at least $f + 1$ non-failed replicas, including the current primary, are able to communicate. If the primary fails, requests cannot be executed in the current view. However if replicas are unable to execute the client request in the current view, they will move to a new one.

Replicas monitor the primary and start a view change by sending the STARTVIEWCHANGE messages if the primary is unresponsive. When other replicas receive the STARTVIEWCHANGE messages they will also advance their *view-number* and send STARTVIEWCHANGE messages. As a result, replicas will receive enough STARTVIEWCHANGE messages so that they can send DOVIEWCHANGE messages, and thus the new primary will receive enough DOVIEWCHANGE messages to enable it to start the next view. And once this happens it will be able to carry out client requests. Additionally, clients send their requests to all replicas if they don't hear from the primary, and thus cause requests to be executed in a later view if necessary.

More generally liveness depends on properly setting the timeouts used to determine whether to start a view change so as to avoid unnecessary view changes and thus allow useful work to get done.

8.2 Correctness of the Recovery Protocol

Safety. The recovery protocol is correct because it guarantees that when a recovering replica changes its status to *normal* it does so in a state at least as recent as what it knew when it failed.

When a replica recovers it doesn't know what view it was in when it failed. However, when it receives $f + 1$ responses to its RECOVERY message, it is certain to learn of a view at least as recent as the one that existed when it sent its last PREPAREOK, DOVIEWCHANGE, or RECOVERYRESPONSE message. Furthermore it gets its state from the primary of the latest view it hears about, which ensures it learns the latest state of that view. In effect, the protocol uses the volatile state at $f+1$ replicas as stable state.

One important point is that the nonce is needed because otherwise a recovering replica might combine responses to an earlier RECOVERY message with those to a later one; in this case it would not necessarily learn about the latest state.

Another important point is that the key to correct recovery is the combination of the view change and recovery protocols. In particular the view change protocol has two message exchanges (for the STARTVIEWCHANGE and DOVIEWCHANGE messages). These ensure that when a view change happens, at least $f + 1$ replicas already know that the view change is in progress. Therefore if a view change was in progress when the replica failed, it is certain to recover in that view or a later one.

It's worth noting that having two message exchanges is necessary. If there were only one exchange, i.e., just an exchange of DOVIEWCHANGE messages, the following scenario is possible (here we consider a group containing three replicas, currently in view v with primary $r1$):

1. Before it failed the recovering replica, $r3$, had decided to start a view change and had sent a DOVIEWCHANGE message to $r2$, which will be the primary of view $v+1$, but this message has been delayed in the network.

2. Replica $r3$ recovers in view v after receiving RECOVERYRESPONSE messages from both $r1$ and $r2$. Then it starts sending PREPAREOK messages to $r1$ in response to $r1$'s PREPARE messages, but these PREPARE messages do not arrive at $r2$.

3. Replica $r3$'s DOVIEWCHANGE message arrives at $r2$, which starts view $v + 1$.

Step 3 is erroneous because the requests that committed after $r3$ recovered are not included in the log for view $v+1$. The round of STARTVIEWCHANGE messages prevents this scenario.

We could avoid the round of STARTVIEWCHANGE messages by having replicas write the new view number to disk before sending the *DoViewChange* messages; this is the approach used in the original version of VR.

Liveness. It's interesting to note that the recovery protocol requires $f + 2$ replicas to communicate! Nevertheless the protocol is live, assuming no more that f replicas fail simultaneously. The reason is that a replica is considered failed until it has recovered its state. Therefore while a replica is recovering, there must be at least $f + 1$ other replicas that are not failed, and thus the recovering replica will receive at least $f+1$ responses to its RECOVERY message.

8.3 Correctness of Reconfiguration

Safety. Reconfiguration is correct because it preserves all committed requests in the order selected for them. The primary of the old group stops accepting client requests as soon as it receives a RECONFIGURATION request. This means that the RECONFIGURATION request is the last committed request in that epoch. Furthermore new replicas do not become active until they have completed state transfer. Thus they learn about all requests that committed in the previous epoch, in the order selected for them, and these requests are ordered before any client request processed in the new epoch.

An interesting point is that it's possible for the primaries in both the old and new groups to be active simultaneously. This can happen if the primary of the old group fails after the reconfiguration request commits. In this case it's possible for a view change to occur in the old group, and the primary selected by this view change might re-run the normal-case protocol for the reconfiguration request. Meanwhile the new group might be actively processing new client requests. However the old group will not accept any new client requests (because the primary of the new view checks whether the topmost request in the log is a reconfiguration request and if so it won't accept client requests). Therefore processing in the old group cannot interfere with the ordering of requests that are handled in the new epoch.

It's worth noting that it is important that the new epoch start in view 0 rather than using the view in which the old epoch ended up. The reason is that STARTEPOCH messages could be sent from old epoch replicas with different view numbers; this can happen if there is a view change in the old group and the new primary re-runs the reconfiguration request. If the new group accepted the view number from the old group, we could end up with two primaries in the new group, which would be incorrect.

Liveness. The system is live because (1) the base protocol is live, which ensures that the RECONFIGURATION request will eventually be executed in the old group; (2)

new replicas will eventually move to the next epoch; and (3) old replicas do not shut down until new replicas are ready to process client requests.

New replicas are certain to learn of the new epoch once a RECONFIGURATION request has committed because after this point old group members actively communicate with both old and new ones to ensure that other replicas know about the reconfiguration. Most important here are the STARTEPOCH messages that old nodes send to new ones if they don't receive EPOCHSTARTED messages in a timely way. These messages ensure that even if the old primary fails to send STARTEPOCH messages to the new replicas, or if it sends these messages but they fail to arrive, nevertheless the new replicas will learn of the new epoch.

Old replicas do not shut down too early because they wait for $f' + 1$ EPOCHSTARTED messages before shutting down. This way we know that enough new replicas have their state to ensure that the group as a whole can process client requests assuming no more than a threshold of failures in the new group.

Old replicas might shut down before some new replicas have finished state transfer. However, this can happen only after at least $f'+1$ new replicas have their state, and the other new replicas will then be able to get up to date by doing state transfer from the new replicas.

A final point is that old replicas shut down by the administrator do not cause a problem, assuming the administrator doesn't do this until after executing a CHECKEPOCH request in the new epoch. Waiting until this point ensures that at least $f'+1$ replicas in the new group have their state; therefore after this point the old replicas are no longer needed.

9 Conclusions

This paper has presented an improved version of Viewstamped Replication, a protocol used to build replicated systems that are able to tolerate crash failures. The protocol does not require any disk writes as client requests are processed or even during view changes, yet it allows nodes to recover from failures and rejoin the group.

The paper also presents a protocol to allow for reconfigurations that change the members of the replica group, and even the failure threshold. A reconfiguration technique is necessary for the protocol to be deployed in practice since the systems of interest are typically long lived.

In addition, the paper describes a number of optimizations that make the protocol efficient. The state transfer technique uses application state to speed up recovery and allows us to keep the log small, by discarding prefixes. The use of witnesses allows service state to be stored at only $f + 1$ of the $2f + 1$ replicas. The paper also describes ways to reduce latency of reads and writes, and to improve throughput when the system is heavily loaded.

Today there is increasing use of replication protocols that handle crash failures in modern large-scale distributed systems. Our hope is that this paper will be a help to those who are developing the next generation of reliable distributed systems.

References

[1] CASTRO, M. Practical Byzantine Fault Tolerance. Technical Report MIT-LCS-TR-817, Laboratory for Computer Science, MIT, Cambridge, Jan. 2000. Ph.D. thesis.

[2] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems 20*, 4 (Nov. 2002), 398–461.

[3] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles* (1989), ACM, pp. 202–210.

[4] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. of the ACM 21*, 7 (July 1978), 558–565.

[5] LAMPORT, L. The Part-Time Parliament. Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, Sept. 1989.

[6] LAMPORT, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 10*, 2 (May 1998).

[7] LISKOV, B. From viewstamped replication to byzantine fault tolerance. In *Replication: Theory and Practice* (2010), no. 5959 in Lecture Notes in Computer Science.

[8] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles* (Pacific Grove, California, 1991), pp. 226–238.

[9] MERKLE, R. C. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - Crypto'87*, C. Pomerance, Ed., no. 293 in Lecture Notes in Computer Science. Springer-Verlag, 1987, pp. 369–378.

[10] OKI, B., AND LISKOV, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[11] OKI, B. M. Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT-LCS-TR-423, Laboratory for Computer Science, MIT, Cambridge, MA, May 1988. Ph.D. thesis.

[12] PARKER, D. S., POPEK, G. J., RUDISIN, G., STOUGHTON, A., WALKER, B., WALTON, E., CHOW, J., EDWARDS, D., KISER, S., AND KLINE, C. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983), 240–247.

[13] SCHNEIDER, F. Implementing Fault-Tolerant Services using the State Machine Approach: a Tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.