Chapter 7
From Viewstamped Replication to Byzantine Fault Tolerance

Barbara Liskov

Abstract The paper provides an historical perspective about two replication protocols, each of which was intended for practical deployment. The first is Viewstamped Replication, which was developed in the 1980's and allows a group of replicas to continue to provide service in spite of a certain number of crashes among them. The second is an extension of Viewstamped Replication that allows the group to survive Byzantine (arbitrary) failures. Both protocols allow users to execute general operations (thus they provide state machine replication); both were developed in the Programming Methodology group at MIT.

7.1 Introduction

This paper describes two replication algorithms. The first, Viewstamped Replication, was developed in the 1980's; it handles failures in which nodes fail by crashing. The second, PBFT (for "Practical Byzantine Fault Tolerance"), was developed in the late 1990's and handles Byzantine failures in which failed nodes can behave arbitrarily and maliciously. Both replication techniques were developed in my research group, the Programming Methodology Group at the Massachusetts Institute of Technology.

The paper has three goals:

- To describe Viewstamped Replication. The protocol is not very complex but this was not evident in the papers that described it since they presented it in the context of specific applications that used it. The goal in this paper is to strip out the extra information and focus on the basics of the protocol.
- To show how PBFT is based on Viewstamped Replication. I believe that because my group had developed Viewstamped Replication, we were in an advantageous position relative to other groups when it came to working on replication techniques that survived Byzantine failures. Furthermore, PBFT can be viewed as an extension of Viewstamped Replication; the paper shows how this works.
- To provide some historical information about what was happening when these two protocols were invented.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 121–149, 2010.
© Springer-Verlag Berlin Heidelberg 2010

7.2 Prehistory

I began work in distributed computing in about 1980. Prior to that time I had been working on data abstraction [21, 22] and the design of the CLU programming language [20]. In the work on CLU I decided to focus on sequential programs since there seemed to be enough other things to worry about. The work on CLU led to the invention of a number of novel programming language constructs in addition to support for data abstraction, including support for parametric polymorphism, iteration abstraction, and exception handling. However, the design of CLU ignored all issues of concurrency.

I always intended to return to consideration of concurrency when the design of CLU was complete. However, when this happened in the late 1970's, distributed computing had become a possibility. At that point the Internet existed and it was being used to send email and do file transfer. Additionally, it was hoped that the Internet could be used to run applications distributed over many machines, but there was little understanding of how that could be accomplished.

Therefore, I decided to focus on distributed computing rather than thinking just about parallel programs that ran on a single machine. I started a project to define a programming language for use in building distributed implementations. This work led to the invention of a programming language and system called Argus [17, 19].

Argus was an object-oriented language. Its programs were composed of objects called *guardians*, which provided operations called *handlers*. Each guardian ran entirely on one machine. However a computation running in one guardian could transfer control to another by making a *handler call*; these calls were one of the early examples of remote procedure calls.

Additionally, Argus ran computations as atomic transactions. A transaction started in some guardian, perhaps in response to input from a user. The execution of the transaction could include remote handler calls, and these in turn might do further remote calls. At the end, the transaction either committed, in which case all modifications at all guardians had to be installed, or it aborted, in which case all its modifications were undone. Additionally there could be many transactions running in parallel, and Argus ensured that they did not conflict. In other words it provided *serializability* for transactions.

One of our concerns in Argus was ensuring that effects of committed transactions survived even in the presence of failures. Clearly one way to achieve this is to record these effects on a stable storage medium, such as a disk, as part of committing the transactions. However, that approach only ensures that modifications will not be lost. It does not provide *availability* since there could be times when clients are unable to access the information; in fact clients could see less availability over what they could obtain by storing the information on their own machine since a failure of either the client machine or the machine that stored the information would make the information unavailable. Additionally, distributed computing provides the possibility of better availability for clients: with enough replicas we could ensure that the service would always be available with high probability.

The question then became how to achieve a correct and efficient replication protocol. This concern led to the development of Viewstamped Replication.

7.3 Viewstamped Replication

Viewstamped Replication, which I will refer to as VR from now on, was invented by Brian Oki and myself. The goal was to support a replicated service, running on a number of replicas. The service maintains a state, and makes that state accessible to a set of *client* machines.

VR was intended from the outset to satisfy two goals. The first was to provide a system where the user code running on the client machines could be unaware that it was dealing with a replicated service. As far as this code was concerned it was interacting with a service provided by a single server, albeit one that was more available than one might expect if the service ran on a single machine. Thus we required that the effect of running operations against the service be identical to what would happen if there were just one copy of the information [27, 2].

The second goal for VR was to provide *state machine replication* [13, 30]: clients could run general operations to observe and modify the service state. An alternative to state machine replication is to provide clients only the ability to read and overwrite individual words or blocks. To illustrate the difference between these two approaches, consider a banking system that provides operations to deposit and withdraw money from an account, as well as operations to observe an account balance and to transfer money from one account to another. These operations typically involve both reads and writes of the system state. State machine replication allows the banking system to be implemented directly: the replicated service provides operations to deposit, withdraw, etc. If only reads and writes are provided, the operations and the synchronization of concurrent requests must be implemented by the application code. Thus state machine replication provides more expressive power than the alternative, and simplifies what application code needs to do. The decision to support state machine replication meshed with the goal of Argus to make it easier to implement applications.

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 failures.

VR was developed under the assumption that the only way nodes fail is by crashing: we assumed that a machine was either functioning correctly or it was completely stopped. We made a conscious decision to ignore Byzantine failures, in which nodes can fail arbitrarily, perhaps due to an attack by a malicious party. At the time, crashes happened fairly frequently and therefore they seemed the most important to cope with. Additionally, the crash model is simpler to handle than the Byzantine model, and we thought we had enough to deal with trying to invent a replication method for it.

VR was 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. We assumed that messages might be lost, delivered late or out of order, and delivered more than once; however, we assumed that if sent repeatedly a message would eventually be delivered. Messages might be corrupted in transit, but we assumed we could distinguish good and bad messages, e.g., through checksums. We did not consider a malicious party that controls the network and therefore we did not think about the need to use cryptography to prevent spoofing.

Brian and I began working on replication in the fall of 1985. Brian completed his Ph.D. thesis in May 1988 [26] and a paper on our approach appeared in PODC in August 1988 [25]. These papers explained replication within the context of support for distributed transactions. In this paper I focus on just the replication protocol and ignore the details of how to run transactions. The description of VR provided here is very close to what appeared in 1988; I discuss the differences in Section 7.5.1.

A later project on the Harp file system applied VR to building a highly available file system. A paper on Harp appeared in SOSP in 1991 [18]. The Harp project extended VR to provide efficient recovery of failed replicas. It also introduced two important optimizations, to speed up the processing of read operations, and to reduce the number of replicas involved in normal case execution.

The work on VR occurred at about the same time as the work on Paxos [14, 15] and without knowledge of that work.

The papers on VR and Harp distinguished what was needed for replication from what was needed for the application (transaction processing in VR, a file system in Harp), but in each case a specific application was also described. In this paper I focus on VR as a generic replication service, independent of the application.

7.3.1 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 f replicas participating, since those replicas might be crashed and unable to reply. However, it is possible that the f replicas we didn't hear from are merely slow to reply, e.g., because of congestion in the network. In this case up to f of the replicas that processed the operation might fail after doing so. Therefore we require that at least $f + 1$ replicas participate in processing the operation, since this way we can guarantee that at least one replica both processed the request and didn't fail subsequently. Thus the smallest group we can run with is of size $2f + 1$.

The membership of the replica group was fixed in VR. If a replica crashed, then when it recovered it rejoined the group and continued to carry out the replication protocol.

7.3.2 Architecture

The architecture for running VR is presented in Figure 7.1. 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., the banking service. 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.

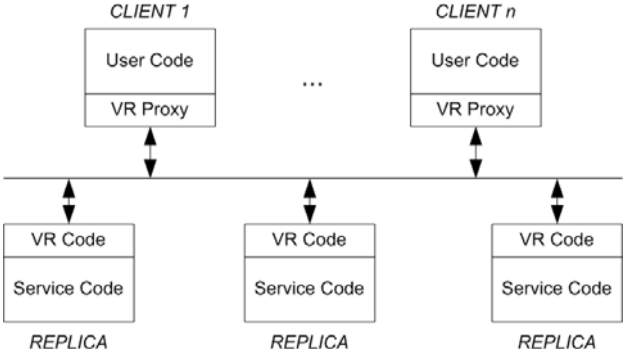

Fig. 7.1 VR Architecture; the figure shows the configuration when $f = 1$.

7.3.3 Approach

One key requirement for a replication protocol is to ensure that every operation executed by the replica group survives into the future in spite of up to f failures. The second key requirement is providing a means to handle concurrent client operations. State machine replication requires a single total ordering of client requests; the challenge is to ensure this when client requests are made concurrently.

Very early in our work on VR, we settled on an approach to replication that uses a *primary*. The primary is just one of the replicas, but it has a special responsibility: it decides on the order for client requests. This way we provide an easy solution to the ordering requirement. Additionally the primary executes the client request and returns the result to the client, but it does this only after at least $f + 1$ replicas (including itself) know about the request. Thus we ensure that no matter what happens in the future, at least one non-faulty replica knows about the request.

The downside of having a primary, however, is that it might fail, yet the protocol needs to continue. Furthermore the continuation must be a legal extension of what happened in the past: the state of the system must reflect all client operations that were previously executed, in the previously selected order.

Our solution to the problem of a faulty primary is to allow different replicas to assume this role over time. The system moves through a sequence of *views*. In each view one of the replicas is selected to be the primary. The other replicas monitor the primary, and if it appears to be faulty, they carry out a *view change* protocol to select a new primary.

Thus VR consists of three protocols, to handle processing of requests, view changes, and also node recovery. These protocols are described in the next section.

7.4 The VR Protocol

This section describes how VR works.

Figure 7.2 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*; the primary is chosen round-robin, starting with replica 1, as the system moves to new views. A *status* of *normal* indicates the replica is handling client requests; this case is discussed in Section 7.4.1. A *status* of *view-change* indicates a replica is engaged in carrying out the view change protocol, which is discussed in Section 7.4.2. A node that has crashed and recovered has a *status* of *recovering* while it interacts with the other replicas to find out what happened while it was failed; recovery is discussed in Section 7.4.3.

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. In addition it records its own *client-id* and a count of the number of requests it has made. A client is allowed to have only a single outstanding request at a time. Each request is given a number by the client and later requests must have larger numbers than earlier ones. The request number is used by the replicas to avoid running requests more than once and therefore if a client crashes and recovers it must start up with a number larger than what it had before it failed; otherwise its request will be ignored. The request number is also used by the client to discard duplicate responses to its requests.

Every message sent to the client informs it of the current *view-number*; this allows the client to track the primary. Every message sent from one replica to another contains the *view-number* known to the sender. Replicas only process messages that match the *view-number* they know. If the sender has a smaller *view-number*, the receiver discards the message but sends a response containing the current *view-number*. 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.

- The *configuration*, i.e., the IP address and replica number for each of the $2f+1$ replicas. The replicas are numbered 1 to $2f+1$. Each replica also knows its own replica number.
- 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 *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.

Fig. 7.2 VR state at a replica.

7.4.1 Normal Operation

This section describes how VR works when the primary isn't faulty. The protocol description assumes that the *status* of each participating replica is *normal*, i.e., it is handling client requests; this assumption is critical for correctness as discussed in Section 7.4.2.

The protocol description assumes the client and all the participating replicas are in the same view; nodes handle different view numbers as described above. The description ignores a number of minor details, such as re-sending requests that haven't received responses. It assumes that each client request is a new one, and ignores suppression of duplicates. Duplicates are suppressed using the *client-table*, which allows old requests to be discarded, and the response for the most recent request to be re-sent.

The *request processing protocol* works as follows:

1. The client sends a ⟨REQUEST *op, c, s, v*⟩ message to the primary, where *op* is the operation (with its arguments) the client wants to run, *c* is the *client-id*, *s* is the number assigned to the request, and *v* is the *view-number* known to the client.
2. When the primary receives the request, it advances *op-number* and adds the request to the end of the *log*. Then it sends a ⟨PREPARE *m, v, n*⟩ message to the other replicas, where *m* is the message it received from the client, *n* is the *op-number* it assigned to the request, and *v* is the current *view-number*.
3. Non-primary replicas process PREPARE messages in order: a replica won't accept a prepare with *op-number n* until it has entries for all earlier requests in its *log*. When a non-primary replica *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 adds the request to the end of its *log* and sends a ⟨PREPAREOK *v, n, i*⟩ message to the primary.
4. The primary waits for f PREPAREOK messages from different replicas; at this point it considers the operation 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 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.

5. At some point after the operation has committed, the primary informs the other replicas about the commit. This need not be done immediately. A good time to send this information is on the next PREPARE message, as piggy-backed information; only the *op-number* of the most recent committed operation needs to be sent.
6. When a non-primary replica learns of a commit, it waits until it has executed all earlier operations and until it has the request in its *log*. Then it executes the operation by performing the up-call to the service code, but does not send the reply to the client.

Figure 7.3 shows the phases of the normal processing protocol.

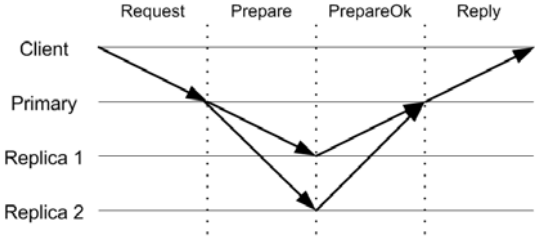

Fig. 7.3 Normal case processing in VR for a configuration with $f = 1$.

The protocol could be modified to allow non-primary replicas 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 VR processes PREPARE messages in *op-number* order.

The protocol need not involve 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 7.4.3.

7.4.2 View Changes

View changes are used to mask failures of the primary.

Non-primary replicas monitor the primary: they expect to hear from it regularly. Normally the primary is sending PREPARE and COMMIT messages, but if it is idle (due to no requests) it sends pings. If a timeout expires without communication (and after some retries), the replicas carry out a view change to switch to a new primary. Additionally, if the client receives no reply to a request, it resends the request to all; this way it learns about the new view, and also prompts the new primary to send it the reply.

The correctness condition for view changes is that every operation that has been executed by means of the up-call to the service code at one of the replicas must survive into the new view in the order selected for it at the time it was executed. This up-call is usually done at the old primary first, and the replicas carrying out

the view change may not know whether the up-call occurred. However, the up-call occurs only for committed operations. This means that the primary has received at least f PREPAREOK messages from other replicas, and this in turn implies that the operation has been recorded in the logs of at least $f+1$ replicas (the primary and the f replicas that sent the PREPAREOK messages).

Therefore the view change protocol must obtain 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. 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 and might assign this number to a different operation. However if two operations are assigned the same number, how can we ensure that the right one is executed at that point in the order?

To solve this problem, we introduced the notion of 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*. Operations are assigned viewstamps: each operation processed by the primary of view v has a viewstamp with that view number, and we associate a viewstamp with every entry in the log. VR guarantees that viewstamps are unique: different client requests are never assigned the same viewstamp. Should a replica receive information about two different operations with the same *op-number* it retains the operation with the higher viewstamp.

VR got its name from these viewstamps.

Viewstamps are used in the *view change protocol*, which 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 suspects the primary is faulty advances its *view-number*, sets its *status* to *view-change*, and sends a ⟨DOVIEWCHANGE *v, l, k, i*⟩ to the node that will be the primary of the next view (recall that the identity of the primary can be determined from the view number). Here v is its *view-number*, l is the replica's log, and k is the *op-number* of the latest committed request known to the replica.
2. When the new primary receives $f+1$ of these messages from different replicas, including itself, it selects as the new *log* the *most recent* of those it received in these messages: this is the one whose topmost entry has the largest viewstamp. It sets the *op-number* to that of the latest entry in the new *log*, changes its *status* to *normal*, and informs the other replicas of the completion of the view change by sending a ⟨STARTVIEW *v, l, k*⟩ message, where l is the new log and k is the *op-number* of the latest committed request it heard about in the responses.
3. The new primary executes (in order) any committed operations that it hadn't executed previously, sends the replies to the clients, and starts accepting client requests.

4. When other replicas receive the STARTVIEW message, they replace their *log* with that 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, and change their *status* to *normal*. Then they continue the protocol for all operations not yet known to be committed by sending PREPAREOK messages for these operations.

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.

To avoid storing a viewstamp in every log entry, VR maintained this information in an auxiliary *view-table*. The *view-table* contained for each view up to and including the current one the *op-number* of the latest request known in that view.

The *view-table* can be used to improve the performance of the view change protocol. The protocol described above is costly because the DOVIEWCHANGE and STARTVIEW messages contain the full log and therefore are large. The cost can be greatly reduced by sending only a suffix of the log in the messages. To send less than the full log, however, requires a way to bring a replica that has missed some earlier view changes up to date. That replica may have requests in its log that were renumbered in subsequent view changes that it didn't participate in. The *view-table* can be used to quickly determine which of its log entries need to be replaced; then it can be brought up to date by providing it with the requests it is missing.

The *view table* can also be used during state transfer to identify the information needed to bring the replica up to date.

Correctness

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. 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 a group of at least $f+1$ non-failed replicas is able to communicate. This follows because if the 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 if the primary is unresponsive. Furthermore, once a node has advanced its *view-number* it no longer accepts messages from older views; instead it informs senders in older views about the new view. This in turn causes those replicas to advance their *view-number* and to take steps to move to that view. As a result 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 learn about new views and cause requests to be executed in a later view if necessary.

More generally liveness depends on properly setting the timeouts used to determine whether the primary is faulty so as to avoid unnecessary view changes.

7.4.3 Recovery

VR assumes a fixed group of replicas. When a replica recovers after a crash it rejoins the system, so that it can start acting as one of the group members again. A replica is considered to be failed from the moment it crashes until the moment when it is ready to rejoin the group.

If nodes record their state on disk before sending messages, a node will be able to rejoin the system immediately. The reason is that in this case a recovering node 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. Failure independence can be accomplished by placing the replicas at different geographical locations to avoid loss of availability when there is a power failure or some local problem like a fire.

VR assumed failure independence and did not require writing to disk during normal case processing. Instead it wrote to disk during the view change. This section describes a recovery protocol that assumes the disk write during a view change. A protocol that requires no disk writes even during the view change is described in Section 7.4.3.

Each replica has non-volatile state consisting of the *configuration* and the *view-number* of the latest view it knows; the rest of the state, e.g., the *log*, is volatile.

The view change protocol is modified slightly, to update the *view-number* on disk. A non-primary replica does the disk write before sending its *log* in the DoViewChange message and the primary does the disk write before sending the StartView message to the other replicas.

When a node recovers it reads the non-volatile information from disk and sets its *status* to *recovering*. It also computes a starting *view-number*: this is what it read from disk, except that if it would be the primary of this view, it advances this number by one. Then it 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.

The *recovery protocol* is as follows:

1. The recovering replica, *r*, sends a ⟨Recovery *v, r*⟩ message to all other replicas, where *v* is its starting *view-number*.
2. A replica *i* replies to a Recovery message only when its status is *normal*, its *view-number* $\geq v$, and it is the primary of its view. In this case the replica sends a ⟨RecoveryResponse *v, l, k, i*⟩ message to the recovering replica, where *v* is its *view-number*, *l* is its *log*, and *k* is the latest committed request.
3. The recovering replica waits to receive a RecoveryResponse message. Then it updates its state using the information in the message. It writes the new *view-number* to disk if it is larger than what was stored on disk previously, changes its status to *normal*, and the recovery protocol is complete. The replica then sends PrepareOk messages for all uncommitted requests.

The protocol just described is expensive because *logs* are big and therefore the messages are big. A way to reduce this expense is discussed in Section 7.5.6.

Correctness

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. This condition is sufficient to ensure that any action the replica took before it fails, such as sending a DoViewChange message, will be reflected in its state.

The reason why the condition holds is because the recovering replica always starts up in a view at least as recent as the view it was in when it failed, and it gets its state from the primary of that view, which ensures it learns the latest state of the view. In more detail, there are three cases of interest:

1. If before it failed the replica had just sent a PrepareOk response to a Prepare message, when it recovers it will either hear from the primary that sent that Prepare message, or from the primary of a later view. In the former case, the *log* it receives will include the operation it sent a PrepareOk message for previously. In the latter case, the *log* will reflect a later state that takes account of its PrepareOk message if it mattered, i.e., if it led to a commit.

2. If before it failed the replica had just sent a DoViewChange message containing its *log*, when it recovers it will either hear from the primary of that view, or from the primary of a later view. In the former case it will receive a *log* that takes account of its message if it was used by the primary; in the latter case, it will receive a *log* that reflects a later state that takes account of its message if it mattered for moving to the later view.
3. If before it failed the node had just sent a RecoveryResponse message then it was the primary when it failed and therefore when it recovers it will hear from a primary of a later view. The *log* it receives from this primary will reflect a later state that takes account of this RecoveryResponse message if it mattered for moving to the later view.

Avoiding Non-volatile Storage

It is possible to avoid any use of non-volatile storage during the protocol. This can be accomplished by adding a "pre-phase" to the view change protocol:

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 other replicas, where *v* identifies the new view.
2. When replica *i* receives $f+1$ StartViewChange messages from different replicas, including itself, it sends a ⟨DoViewChange *v, l, k, i*⟩ message to the new primary.

After this point the view change protocol proceeds as described previously.

The recovery protocol also needs to be a bit different:

1. The recovering replica, *r*, sends a ⟨Recovery *r*⟩ message to all other replicas.
2. A replica *i* replies to a Recovery message only when its status is *normal*. In this case the replica sends a ⟨RecoveryResponse *v, l, k, i*⟩ message to the recovering replica, where *v* is its *view-number*. If *i* is the primary of its view, *l* is its *log*, and *k* is the latest committed request; otherwise, these values are *nil*.
3. The recovering replica waits to receive $f+1$ RecoveryResponse messages from different replicas, 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 replica then sends PrepareOk messages for all uncommitted requests.

This protocol works (in conjunction with the revised view change protocol) because it is using the volatile state at $f+1$ replicas as stable state. When a replica recovers it doesn't know what view it was in when it failed and therefore it can't send this information in the RECOVERY message. 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.

7.5 Discussion of VR

7.5.1 Differences from the Original

The protocol described in the previous section is close to what was described in the VR papers, but there are a few differences. In all cases, the technique described here was adopted from our later work on Byzantine fault tolerance.

First, in the original protocol only replicas that participated in the view change for a particular view were considered to be in that view. This means that view changes happened more frequently than in the protocol described in this paper. In the protocol described here, view changes happen only to mask a failed primary (or a primary that appears to be failed, but is merely slow or has trouble communicating). In the original protocol, a view change was also needed when a non-primary replica failed or became partitioned from the group, and another view change was needed when the replica recovered or the partition healed.

Second, in the original protocol the primary was chosen differently. Rather than being selected deterministically based on the *view-number*, as discussed here, it was chosen at the end of the view change to be the replica with the largest log. This allowed the old primary to continue in this role in view changes that were run for other reasons than failure of the primary.

Third, in the original protocol, replicas competed to run a view change. Several replicas might be running the protocol at once; each of them collected state information from the other replicas and since they might end up with a different initial state for the next view, there had to be a way to choose between them. The approach presented in this paper takes advantage of our way of choosing the primary (using just the *view-number*) to avoid this problem by having the primary of the next view determine the initial state of the view.

A final point is that in the original protocol, replicas exchanged "I'm alive" messages. These exchanges allowed them to notice failures of other replicas and thus do view changes; they were needed because a failure or recovery of any replica led to a view change. The protocol described here instead depends on the client sending to all replicas when it doesn't get a response.

7.5.2 Two-Phase Commit

Clearly VR was heavily influenced by the earlier work on two-phase commit [10]. Our primary is like the coordinator, and the other replicas are like the participants. Furthermore the steps of the protocol are similar to those in 2PC and even have names (prepare, commit) that come from 2PC. However, unlike in two-phase commit, there is no window of vulnerability in VR: there is never a time when a failure of the primary prevents the system from moving forward (provided there are no more than f simultaneous failures). In fact, VR can be used to replicate the coordinator of two-phase commit in order to avoid this problem.

7.5.3 Optimizations

Latency

As illustrated in Figure 7.3, the VR protocol requires four message delays to process operations. This delay can be reduced for both read operations that observe but do not modify the state and *update* operations that both observe and modify the state.

Read Operations. The paper on Harp [18] proposed a way to reduce the delay to two messages for reads. The primary immediately executed such a request by making an up-call to the service code, and sent the result to the client, without any communication with the other replicas. This communication is not needed because read operations don't modify state and therefore need not survive into the next view. This approach not only reduced the latency of reads (to the same message exchange that would be needed for a non-replicated service); it also reduced bandwidth utilization and improved throughput since PREPARE messages for read operations didn't need to be sent to the other replicas (although it isn't clear that Harp took advantage of this).

However, executing read operations this way would not be correct if it were possible that a view change had occurred that selected a new primary, yet the old one didn't know about this. Such a situation could occur, for example, if there were a network partition that isolated the old primary from the other replicas, or if the old primary were overloaded and stopped participating in the protocol for some period of time. In this case the old primary might return a result for the read operations based on old state, and this could lead to a violation of external consistency [9]. To prevent this violation, Harp used *leases*: the primary processed reads unilaterally only if it held valid leases from f other replicas, and a new view could start only when the leases at all participants in the view change protocol had expired. The Harp mechanism depended on loosely-synchronized clocks for correctness [24]. In fact it is easy to see that loosely synchronized clock rates (i.e., assuming a bound on the skew of the rates at which the clocks tick) are all that is needed.

Updates. One message delay can be removed from operations that modify the service state as follows. When a replica receives the PREPARE message, in addition to sending a PREPAREOK message to the primary it does the up-call (after it has executed all earlier requests) and sends a reply to the client. The client must wait for $f + 1$ replies; this way we are certain that the operation has committed since it is known at this many replicas.

The approach leads to a delay of 3 messages to run an update. The revised protocol requires more messages, since the non-primaries must reply to the client (as well as to the primary). However, these messages can be small: the client can identify a "preferred" replier, and only this replica needs to send the full reply; the others just send an ack.

A final point is that reduced latency for updates is possible only with some help from the service code. The problem is that the update requests are being executed speculatively, since up-calls are made before the operation commits. Therefore it's

possible that a view change will make it necessary to undo the effects of that up-call on the service state.

The optimization for updates was not proposed in Harp, but instead is based on later work done on Byzantine-fault tolerance as discussed in Section 7.8.2.

Witnesses

Another innovation in Harp was a way to avoid having all replicas run the service. In Harp the group of $2f + 1$ replicas included $f + 1$ *active* replicas that stored the system state and executed operations. The other f replicas, which were referred to as *witnesses*, did not. The primary was always an active replica. The witnesses didn't need to be involved in the normal case protocol at all as long as the $f + 1$ active replicas were processing operations. Witnesses were needed for view changes, and also to fill in for active replicas when they weren't responding. However most of the time witnesses could be doing other work; only the active replicas had to be dedicated to the service.

7.5.4 Performance in the Normal Case

Avoiding disk writes during operation execution provides the opportunity for VR to outperform a non-replicated service in the case where the message delay to the replicas is less than the delay due to a disk I/O.

The Harp paper shows this effect. The paper presents results for NFS [29] running the Andrew benchmark [11] and also Nhfstone [31], for a configuration where the communication delay between the replicas was about 5 ms. In both cases Harp was able to do better than an unreplicated system. The paper reports that in addition the system saturated at a higher load than the unreplicated system did. In these experiments, the gain came from avoiding synchronous disk I/O in the foreground; these disk writes are required for update operations done at a single machine by the NFS specification.

At the time we did the work on Harp, a delay of 5 ms was possible only on a local area net. Harp ran in such a setting; Harp placed all replicas in the same data center connected by a local area network. This was not an ideal set up, because, as mentioned earlier, the replicas ought to be failure independent. The paper on Harp proposed a partial solution for the failure-independence problem, by handling power failures. Each replica had an Uninterruptible Power Supply, to allow nodes to write some information to disk in the case of a power failure. Harp pushed the log to disk on a regular basis (in the background), so that it would be able to write what remained in volatile memory to disk in the case of a power failure.

Today we need not sacrifice failure independence to outperform an unreplicated system. Instead we can place replicas in different data centers to achieve failure independence, yet still have a communication delay that is smaller than writing to disk.

7.5.5 Performance of View Changes

The Harp project also addressed the problem of efficient view changes.

The view change protocol is lightweight: there is only a single message delay from the time the replicas decide a view change is needed until the new primary has the state of the new view. After this point the primary can run the protocol for uncommitted requests and it can accept new requests. However it cannot execute these requests until it has executed all earlier ones.

Harp ensured that a new primary can start processing new requests with little delay. It accomplished this by having non-primary replicas execute operations eagerly, so that they were almost up to date when they took over as primary.

7.5.6 State Management

When a replica recovers from a crash it needs to bring itself up to date. The question is how to do this efficiently.

One way for a replica to recover its state after a crash is to start with the initial application state and re-run the log from the beginning. But clearly this is not a practical way to proceed, since the log can get very large, and recovery can take a long time, even if we eliminate read operations and updates whose modifications are no longer needed, e.g., modifications of files that were subsequently deleted.

Harp had a more efficient solution that took advantage of non-volatile state at the replica, namely the state of the service running at the replica. Given this state, it is only necessary to run the requests in the suffix of the log after the latest request executed before the replica failed. Doing things this way allowed the size of the log to be reduced, since only the suffix was needed, and greatly shortened the time needed to recover.

The solution in Harp was to retain a log suffix large enough to allow any active replica to recover. (Recall that Harp had $f+1$ active replicas and f witnesses that did not store state nor participate in normal processing when the active replicas were not faulty.) Each active replica tracked when effects of requests made it to disk locally. As soon as the effects of a request had made it to disk at all active replicas, the request could be removed from the log. In Harp this point was reached speedily in the normal case of no failures of active replicas because even non-primary replicas executed requests eagerly, as discussed in the preceding section. Removal of log entries stalled while an active replica was out of service and therefore the log was certain to contain all requests that replica might not have processed. When an active replica recovered, it fetched the log from the other replicas, and re-ran the requests in log order.

This approach ensures that all requests needed to recover the replica state exist in the log. But it leads to the possibility that an operation might be executed both before a node fails and again as part of recovery. Note that even if a replica wrote the latest viewstamp to disk each time it executed an operation, it cannot know for sure whether the service code executed the operation before the failure. And in general we would like to avoid writing the viewstamp to disk on each operation.

Of course there is no difficulty in re-executing operations on the service state if those operations are *idempotent*. The solution in Harp was to make operations idempotent by doing extra processing. It pre-processed operations at the primary to predict their outcome, and stored this extra information along with the request in the *log*. For example, if the request created a file in a directory, Harp predicted the slot into which the file would be placed and stored the slot number in the log. Therefore, when the operation re-ran, the file would be placed in that slot, even though this is not where it would have gone based on the current state (which recorded the result of operations ordered after that one).

In the work on Byzantine-fault tolerance, we came up with a different approach that avoided the need to make operations idempotent. That approach is discussed briefly in Section 7.8.4.

7.5.7 Non-deterministic Operations

State machine replication requires that each operation be deterministic. However, applications frequently have non-deterministic operations. For example, reads and writes are non-deterministic in NFS because they require setting "time-last-read" and "time-last-modified". If this is accomplished by having each replica read its clock independently, the states at the replicas will diverge.

The paper on Harp explained how to solve the problem, using the same pre-processing approach that was used to provide idempotency. The primary preprocessed the operation to predict the outcome and sent the information to the other replicas in the PREPARE message. All replicas then used the predicted outcome when the request was executed.

7.6 Byzantine Fault Tolerance

After the end of the Harp project, we stopped working on replication protocols for a while. Then toward the end of 1997, DARPA published a Broad Area Announcement (BAA) requesting proposals on the topic of survivability, and I asked Miguel Castro, who was a student in my group at the time, to think about how we might respond.

By this time there was a realization that malicious attacks and Byzantine behavior needed to be dealt with, and this kind of issue was central to the BAA. Looking at this BAA got us interested in Byzantine-fault-tolerant replication protocols, and we began trying to invent such a protocol. This work led to PBFT, the first practical replication protocol that handles Byzantine faults.

A first paper on PBFT was published in OSDI 1999 [5]. That paper described the basic approach using public-key cryptography and it did not include the recovery mechanism. The complete protocol is described in in OSDI 2000 [6], in TOCS [7], and also in Miguel's Ph.D. thesis [4].

In this section I do not attempt to describe PBFT, which is well-covered in the literature. What I do instead is to present a simplified version of PBFT, similar to what was described in the first OSDI paper, with the goal of showing how PBFT

grew out of VR. In retrospect PBFT can be seen as an extension of VR to handle the possibility of Byzantine-faulty nodes. However, it was far from straightforward to come up with the extension at the time we were doing the work.

In addition to extending VR to handle Byzantine nodes, PBFT introduced an innovation in the form of proactive recovery, and provided a number of optimizations to improve performance; a brief discussion is provided in Section 7.8.

7.6.1 Approach

Like VR, PBFT ensures reliability and availability when up to f replicas are faulty. However, it allows replicas to fail in a Byzantine manner. This means they can behave arbitrarily: in addition to not replying to requests, or to replying in obviously bad ways, they can also appear to be working correctly as far as other nodes can tell. For example, they might appear to accept modification requests, yet discard the state.

PBFT uses $3f+1$ replicas to tolerate up to f faulty nodes; this many replicas is known to be the minimum required in an asynchronous network [3]. The system must be able to execute a request using responses from $2f+1$ replicas. It can't require more than this many replies because the other f replicas might be faulty and not replying. However, the f replicas we do not hear from might merely be slow to reply, and therefore up to f of the replies might be from faulty nodes. These replicas might later deny processing the request, or otherwise act erroneously.

We can mask such bad behavior, however, since we have replies from at least $2f+1$ replicas, and therefore we can be sure that at least $f+1$ honest replicas know about the request. Since every request will execute with $2f+1$ replicas, we can guarantee that at least one honest replica that knows of this request will also participate in the processing of the next request. Therefore we have a basis for ensuring ordered execution of requests.

Like VR, PBFT uses a primary to order client requests and to run a protocol in a way that ensures that each request that is executed will survive into the future, in spite of failures, in its assigned place in the order. However, in PBFT we have to allow for the fact that replicas might be Byzantine, which leads to differences in the PBFT protocol compared to the VR protocol.

Additionally PBFT needs to allow for an adversary that controls the network. The adversary can remove messages, cause them to be delivered late and out of order, and corrupt them; it can also create new messages and attempt to spoof the protocol. To prevent spoofing, PBFT uses cryptography; all messages are signed by the sender, and we assume that the secret key used by an honest node to sign messages is not known to the adversary. PBFT also needs to avoid replay attacks, but the needed ingredients are already present in the VR protocol, e.g., viewstamps, since VR had to handle replays, although in that case we assumed the network was not acting maliciously.

The architecture for PBFT is similar to that shown in Figure 7.1, except that now there must be $3f+1$ replicas to survive f failures instead of $2f+1$. Another point is that PBFT was explicitly based on this architecture: PBFT separated the protocol

layer from the application layer. The code for PBFT was made available as a library that could be loaded on the clients and the replicas.

7.7 The PBFT Protocol

One way in which PBFT handles Byzantine faulty nodes is by doing each step of the protocol at at least $2f + 1$ replicas, rather than the $f + 1$ needed in VR. However this change alone is not sufficient to provide a correct protocol. The problem is that in VR some decisions are made by just one replica. For example, in normal case processing the primary tells the other replicas the viewstamp assigned to each client request. In VR the other replicas act on this information; since we assume that the primary is honest, we can rely on the viewstamp it assigns and also we can assume it reports honestly on the client request.

In PBFT, however, the primary might be lying. For example, it might assign the wrong viewstamp, one assigned in the past to a different request. Or, it might provide a bogus client operation or replay a previous request by the client. Another possibility is that it might send different PREPARE messages to the other replicas, e.g., instructing some of them to perform request $r1$ at viewstamp v and others to perform request $r2$ at the same viewstamp. Note that the interesting case here is when the primary does something that can't be recognized as bad just by looking at the message! It's much easier to handle cases where the message is not sent or is garbled.

Our solution to handling these misbehaviors of the primary was to add an extra phase to the protocol, at the beginning, prior to the prepare phase. We called this the *pre-prepare* phase. Additionally replicas check various details of what the primary is doing and refuse to process messages that are not what they should be.

The following is a description of a simplified version of the PBFT protocol. The protocol is based on the one presented in [5] and uses public-key cryptography rather than symmetric cryptography. Both clients and replicas have known public keys and use their secret keys to sign their messages; all messages are signed in this way. In the full version of PBFT, public key cryptography is avoided almost always. This improves the performance of the protocol but also complicates it, as discussed further in Section 7.8.1.

The protocol presented here requires that replicas process requests in order, similar to what was done in VR. For example, a replica won't process a PREPARE message for a particular viewstamp unless it knows about all requests that have been assigned earlier viewstamps. The unsimplified version of PBFT relaxed this constraint and allowed various protocol messages to be processed out of order.

The state of a PBFT replica is the same as was presented before, in Figure 7.2, with one important difference. In VR the log contains just the request messages sent by the client. In PBFT, each log entry also contains some of the protocol messages used to run the request assigned to that *op-number*.

The simplified *request processing protocol* works as follows. As in VR, replicas process requests only when their *status* is *normal*. Also they ignore requests from

earlier views and if they learn of a later view, or if they learn they are missing entries in their log, they bring themselves up to date before processing the request.

1. The client c sends a $\langle$REQUEST *op, c, s, v*$\rangle_{\sigma_c}$ message to the primary, where *op* is the request, c is the *client-id*, s is the number the client assigned to the request, v is the *view-number* known to the client, and σ_c is the client's signature over the message.
2. If this is not a new request, or if the signature isn't valid, the request is discarded. Otherwise the primary advances *op-number* and adds the request to the end of the *log*. Then it sends a $\langle\langle$PREPREPARE *d, v, n*$\rangle_{\sigma_p}$ *m*$\rangle$ message to the other replicas, where m is the client message, d is a digest (a cryptographic hash) of m, n is the *op-number* assigned to the request, and σ_p is the primary's signature.
3. A replica i discards PREPREPARE requests with invalid signatures, or if it had already accepted a different request at that viewstamp. If the request is valid, it waits until it has PREPREPARE entries in its *log* for all requests with earlier *op-numbers*. Then it adds the PREPREPARE message to its *log* (and updates the *client-table*) and sends a $\langle$PREPARE *d, v, n, i*$\rangle_{\sigma_i}$ message to all replicas, where d is the digest of the client request and σ_i is i's signature.
4. When replica i receives valid PREPARE messages for which it has the matching PREPREPARE message in its log, it adds them to the log. When it has the PREPREPARE message from the primary and $2f$ valid matching PREPARE messages, all from different non-primary replicas, including itself, for this request and all earlier ones, we say the request is *prepared at replica i*. At this point, replica i sends a $\langle$COMMIT *d, v, n, i*$\rangle_{\sigma_i}$ message to all other replicas.
5. When replica i receives $2f+1$ valid COMMIT messages, all from different replicas including itself, and when additionally the request is prepared at replica i, replica i executes the request by making an up-call to the service code, but only after it has executed all earlier requests. Then it returns the result to the client.

The first thing to notice about the protocol is the extra *pre-prepare* phase. Since we can't trust the primary to tell the truth we instead use $2f+1$ replicas; if this many replicas agree, we can rely on what they say since at least $f+1$ of them will be honest, and at least one honest replica will know what has happened before, e.g., whether some other request has already been assigned that viewstamp.

Here we are relying on a principle at work in a Byzantine setting: *we can trust the group but not the individuals*. This principle is used in every step of the protocol; messages from a sufficient number of replicas are needed to ensure that it is correct to take that step.

Thus each replica needs to see $2f+1$ valid matching COMMIT messages to decide that it can execute the request. Additionally the client needs to see matching reply messages. In this case, however, $f+1$ matching responses is sufficient because at least one of them comes from an honest replica, and an honest replica won't send such a response unless the request has gone through the complete protocol.

The phases of the protocol are illustrated in Figure 7.4. It may seem that the PBFT protocol has an extra commit phase as well as the pre-prepare phase. However, the COMMIT messages in PBFT correspond to the PREPAREOK messages in VR.

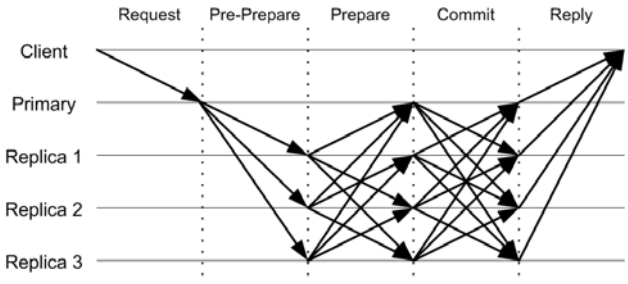

Fig. 7.4 Normal case processing in PBFT.

The protocol uses all-to-all communication for the PREPARE and COMMIT messages and therefore uses $O(n^2)$ communication. All-to-all communication wasn't needed for VR. It could be avoided in PBFT by funneling messages through the primary, but the primary would need to sends copies of the messages it received, since ultimately all replicas need to be able to see the messages of all the others. Thus if we funneled messages through the primary, its messages would be bigger (i.e., there would be no change in bandwidth utilization), and the protocol latency would increase.

7.7.1 View Changes

The main problem with view changes in a Byzantine setting is figuring out which operations must make it into the new view. We must ensure that any operations that executed at an honest replica survive into the next view in their assigned order. In PBFT, an honest replica will execute an operation only after it receives $2f + 1$ COMMIT messages. The problem now, however, is that it's possible that only one honest replica that received this many messages participates in the view change protocol, and furthermore, dishonest replicas might also participate in the view change protocol and claim that some other operation should receive that viewstamp.

For example, suppose that request $r1$ has executed at viewstamp v at some honest replica, and then a view change occurs. The view change protocol will hear from at least one honest replica that knows about $r1$. However, as many as f dishonest replicas might participate in the view change protocol and claim that some other request $r2$ has been assigned to v. In fact, if the primary is bad and assigns different requests to the same viewstamp, $2f$ replicas might claim this: the f liars and also f honest replicas that were told that $r2$ should run at viewstamp v.

Clearly we can't resolve this question by relying on a majority opinion!

The way PBFT resolves this dilemma is to rely on *certificates*. A certificate is a collection of matching valid signed messages from $2f + 1$ different replicas. A certificate represents a proof that a certain thing has happened, e.g., that a request has prepared. Since the messages are signed using public key cryptography, any replica is able to evaluate a certificate and decide for itself whether it is valid.

The certificates are composed of the messages replicas receive while running the protocol. In particular we use a *prepare certificate* consisting of a $\langle$PREPREPARE *d, v, n*$\rangle_{\sigma_p}$ message and $2f$ PREPARE messages, all for the same request (represented as a digest) with the same viewstamp. A replica has such a certificate for each request that has prepared at it.

The view change protocol works as follows:

1. A replica i that decides there needs to be a view change advances its viewstamp and sends a $\langle$DOVIEWCHANGE *v, P, i*$\rangle_{\sigma_i}$ to the new primary, where v is the new viewstamp and P is the set of prepare certificates known to i. Since i processes the protocol in request order, there will be prepare certificates for a prefix of its log entries.
2. When the new primary receives $2f+1$ such messages from different replicas, including itself, it sets its viewstamp to the one in the messages and constructs a new log containing an entry for each prepare certificate it received. Then it sets its *status* to *normal* and sends a $\langle$STARTVIEW *mlist, v, O*$\rangle$ message to the other replicas, where *mlist* is the set of $2f+1$ DOVIEWCHANGE messages it received, all from different replicas, and O is a set of $\langle$PREPREPARE *d, v, n*$\rangle_{\sigma_p}$ messages, one for each request in the log.
3. When replica i receives a valid STARTVIEW message, it processes the messages in the *mlist* and reconstructs its log. Then it sets its *status* to *normal* and re-runs the protocol for each request in O (but it only executes requests that it hasn't already executed).

Certificates are used in step 1 of this protocol, so that a replica can reliably inform the primary about the requests that have prepared at it. They are also used in step 2 of the protocol. In this case the certificate consists of the $2f+1$ DOVIEWCHANGE messages; these allow the other replicas to construct a valid log and to check that the set O is correct. Note that all honest replicas will construct the same log given the same set of DOVIEWCHANGE messages.

It's easy to see the relationship of this protocol to the view change protocol in VR. Of course the protocol now needs to run at $2f+1$ replicas rather than $f+1$. Furthermore, since individual replicas aren't trusted in a Byzantine environment, replicas have to prove what they have using certificates, rather than just reporting. A final difference is that to rerun the protocol in the next view, the primary must produce the PREPREPARE messages for all the requests, since these will need to be combined with PREPARE messages to produce certificates in later views. In VR, replicas ran the protocol for preparing requests without requiring the additional PREPREPARE messages.

The protocol above handles view changes where all honest replicas notice a problem with the primary, e.g., that it hasn't sent messages for some time period. In addition, an individual replica can force a view change by proving that the primary is lying. The proof consists of contradictory messages, e.g., two PREPREPARE messages for the same viewstamp but different requests.

The protocol is robust against bad replicas trying to force a view change when one isn't needed. Each replica decides independently about whether a view change

is necessary and therefore $f + 1$ honest replicas must make this decision before the view change will happen.

The protocol isn't very efficient because the DoViewChange and StartView messages are very large. PBFT solves this problem by taking a *checkpoint* periodically. A checkpoint summarizes a prefix of the log. Once the checkpoint has been taken, all entries in the log below that point are discarded. Only the portion of the log beyond the checkpoint need be sent in the view change messages. Checkpoints are discussed further in Section 7.8.4.

Correctness

The correctness condition we need to satisfy is that every operation executed by an honest replica makes it into the next view in the order assigned to it previously. This condition is satisfied because an operation executes at an honest replica only after the replica receives $2f + 1$ Commit messages for it. Here we are concerned only with what happens at correct replicas, because dishonest replicas can do anything.

If an honest replica receives this many Commit messages, this means that that request has prepared at at least $f + 1$ honest replicas, and each of these replicas has a prepare certificate for it and also for all earlier requests. Furthermore at least one of these $f + 1$ honest replicas will participate in the view change and report these requests with their certificates. Therefore the request will end up in the new log in the position assigned to it previously.

7.8 Discussion of PBFT

This section provides a brief discussion of some of the issues addressed by the full PBFT protocol; more information can be found in [5, 6, 7, 4].

7.8.1 Cryptography

PBFT uses symmetric cryptography most of the time. It uses public keys only to establish secret keys between pairs of replicas and also between replicas and clients.

Using symmetric cryptography represents an important optimization, since it is much more efficient than public key cryptography. However, it has a fairly substantial impact on the protocol because it is now more difficult for replicas to provide proofs. With public keys a certificate containing $2f + 1$ valid matching messages acts as a proof: any of the other replicas can vouch for the validity of these messages since all of them do this using the sender's public key. With symmetric cryptography this simple technique no longer works, and PBFT contains mechanisms to get around this shortcoming.

7.8.2 Optimizations

PBFT provides a number of important optimizations. Most significant are optimizations that reduce the latency for request processing from 5 message delays, as shown

in Figure 7.4, to 2 (for reads) and 4 (for updates), and an optimization to reduce the overhead of running the protocol by batching.

Read Operations

The simple and very robust VR technique of having the primary carry out the read doesn't work for PBFT since the primary can lie. Instead the client sends the request to all replicas, which execute the request immediately and send the reply to the client. The client waits for $2f + 1$ matching replies (actually one full reply and $2f$ digests). If it succeeds in getting this many matching replies it accepts the answer and the operation is over. Otherwise the operation must run through the primary in the normal way.

This optimization succeeds provided there is no contention (and also assuming that the replicas aren't faulty). Each replica runs the request when it arrives, which means that different replicas will run it at different spots in the serial order. However, even so they will produce the same answer provided there is no update operation that modifies the portion of the state being observed by the read and that happens at about the same time as the read.

Update Operations

Rather than waiting until they receive $2f + 1$ COMMIT messages, replicas instead execute an update operation at the time they send their COMMIT message for it (and after executing all operations before it). The client waits for $2f + 1$ matching responses (again, one full reply and $2f$ digests). This way we are sure that the operation will survive into the new view, since at least one of the replicas that sent the response is honest and will be consulted in the next view change. Waiting for only $f + 1$ replies, as is done in the base protocol, isn't sufficient since with this many replies, it is possible that only one honest replica knows of the prepare, and it might not be consulted in the next view change.

A final point is that this way of running updates is the basis for the update optimization for VR that was discussed in Section 7.5.3.

Batching

PBFT is a fairly heavyweight protocol in terms of the amount of message traffic required to run it. However, this traffic can be greatly reduced through batching. Batching simply means running the protocol for a number of requests at once.

Batching has no impact on latency when the system isn't busy: in this case the primary doesn't batch, but instead starts the protocol for each operation when its request message arrives. However, when the load goes up, the primary switches to running requests in batches. Batching thus reduces the overhead of running the protocol by amortizing the cost across all the requests in the batch, without much impact on latency, since when the system is busy the next batch fills up quickly.

7.8.3 Selecting the Primary

The idea of selecting the primary round-robin based on the current view-number comes from PBFT. PBFT requires a way of choosing the primary that cannot be affected by the adversary. In the original version of VR the same node could continue as primary as long as it participated in the view changes. In a Byzantine setting this wouldn't work because the primary might be malicious.

7.8.4 Recovery

PBFT provides a full recovery solution that supports doing disk writes in the background; it does not require disk writes during either normal case processing or view changes, and does not require making requests idempotent. The technique also provides for efficient application-level state transfer using Merkle trees [23], and a way of keeping the log small by taking checkpoints. The recovering replica uses the application-level state transfer to recover its state to the most recent checkpoint, and then runs the log from that point on to get up to date.

Checkpoints require some help from the application, both to create the checkpoint, and to revert to a checkpoint. Reverting is needed to support the update optimization. Since the update is performed speculatively before it commits, it might need to be undone in case of a view change. In PBFT, undoing is accomplished by reverting to the previous checkpoint and then running forward using the log. The application can make use of conventional copy-on-write techniques to support checkpoints.

In addition PBFT provides a *proactive* recovery mechanism, in which nodes are shut down periodically and restarted with their memory intact but with a correct copy of the code. Proactive recovery reduces the likelihood of more than f replicas being faulty simultaneously because their code has been corrupted by a malicious attack.

7.8.5 Non-determinism

VR handles the requirement for determinism by having the primary predict the outcome, as discussed in Section 7.5.7. PBFT can't use this technique since the primary might lie. Instead, PBFT relies on the group to predict the outcome: the primary runs a first phase in which it collects predictions from $2f+1$ different replicas, including itself, and places these predictions in the PREPREPARE message. Later, when the request is executed, replicas compute the outcome using a deterministic function of this information.

7.9 Conclusions

This paper has described two replication protocols. The first is Viewstamped Replication, which was a very early state machine replication protocol that handled machines that failed by crashing. The descriptions of VR that appeared in the litera-

ture describe the protocol along with an application that uses it. The presentation here strips out the application details and presents the protocol in an application-independent way; additionally, some details have been changed so that the protocol described here is close to what was described in the literature, but not identical.

VR allowed failed nodes to restart and then run a recovery protocol to recover their state. The protocol was based on the assumption that replicas were failure independent, and therefore we were able to avoid the use of non-volatile storage during normal request processing. VR did make use of a disk write as part of a view change. The paper describes a variation on the protocol that avoids the need for disk writes entirely, even during view changes.

The paper also presented a simplified version of PBFT. PBFT was the first practical replication protocol that supported state machine replication in a way that survived Byzantine failures. PBFT grew out of VR. It required the use of $3f+1$ replicas rather than $2f+1$. It added an extra phase to normal case processing, to prevent a malicious primary from misleading the other replicas. Also, it used the notion of certificates to ensure that all committed operations make it into the next view in spite of whatever faulty replicas might attempt to do.

Since PBFT was invented there has been quite a bit of research on related protocols. This work covers a number of topics, including: techniques for heterogeneous replication to avoid the problem of correlated failures causing many replicas to fail simultaneously [28, 32]; study of system properties when more than f replicas fail simultaneously [16]; avoiding the use of a primary, either entirely or during normal case processing [1, 8]; reducing the number of replicas that must run the application [34]; and reducing the latency of normal case processing [12, 33].

References

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine Fault-Tolerant Services. In: SOSP 2005, Brighton, United Kingdom (Oct. 2005)
2. Bernstein, P.A., Goodman, N.: The Failure and Recovery Problem for Replicated Databases. In: Second ACM Symposium on the Principles of Distributed Computing, Aug. 1983, pp. 114–122 (1983)
3. Bracha, G., Toueg, S.: Asynchronous Consensus and Broadcast Protocols. Journal of the ACM 32(4), 824–840 (1985)
4. Castro, M.: Practical Byzantine Fault Tolerance. Technical Report MIT-LCS-TR-817, Laboratory for Computer Science, MIT, Cambridge, ph.D. thesis (Jan. 2000)
5. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proceedings of OSDI 1999, New Orleans, LA (Feb. 1999)
6. Castro, M., Liskov, B.: Proactive Recovery in a Byzantine-Fault-Tolerant System. In: Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA (Oct. 2000)
7. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
8. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In: Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI), Seattle, Washington (Nov. 2006)
9. Gifford, D.K.: Information Storage in a Decentralized Computer System. Technical Report CSL-81-8, Xerox Corporation, ph.D. thesis (Mar. 1983)

10. Gray, J.N.: Notes on database operating systems. In: Flynn, M.J., Jones, A.K., Opderbeck, H., Randell, B., Wiehle, H.R., Gray, J.N., Lagally, K., Popek, G.J., Saltzer, J.H. (eds.) Operating Systems. LNCS, vol. 60, pp. 393–481. Springer, Heidelberg (1978)
11. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M.: Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems 6(1), 51–81 (1988)
12. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine Fault Tolerance. In: Proceedings of SOSP 2007, Stevenson, WA (October 2007)
13. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Comm. of the ACM 21(7), 558–565 (1978)
14. Lamport, L.: The Part-Time Parliament. Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA (Sep. 1989)
15. Lamport, L.: The Part-Time Parliament. ACM Transactions on Computer Systems 10(2) (1998)
16. Li, J., Mazieres, D.: Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In: Proceedings of the 4th NSDI, Apr. 2007, USENIX, Cambridge, MA, USA (2007)
17. Liskov, B.: Distributed Programming in Argus. Comm. of the ACM 31(3), 300–312 (1988)
18. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M.: Replication in the Harp File System. In: Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Pacific Grove, California, pp. 226–238 (1991)
19. Liskov, B., Scheifler, R.W.: Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Transactions on Programming Languages and Systems 5(3), 381–404 (1983)
20. Liskov, B., Snyder, A., Atkinson, R., Schaffert, J.C.: Abstraction Mechanisms in CLU. Comm. of the ACM 20(8), 564–576 (1977), also in Zdonik, S. and Maier, D. (eds.) Readings in Object-Oriented Database Systems
21. Liskov, B., Zilles, S.: Programming with Abstract Data Types. In: Proceedings of the ACM SIGPLAN Conference on Very High Level Languages, vol. 9, Apr. 1974, pp. 50–59. ACM Press, New York (1974)
22. Liskov, B., Zilles, S.: Specification Techniques for Data Abstractions. IEEE Transactions on Software Engineering 1(1) (1975)
23. Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988)
24. Mills, D.L.: Network time protocol (version 1) specification and implementation. DARPA-Internet Report RFC 1059 (Jul. 1988)
25. Oki, B., 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, pp. 8–17 (1988)
26. Oki, B.M.: Viewstamped Replication for Highly Available Distributed Systems. Technical Report MIT-LCS-TR-423, Laboratory for Computer Science, MIT, Cambridge, MA, ph.D. thesis (May 1988)
27. Papadimitriou, C.H.: The Serializability of Concurrent Database Updates. Journal of the ACM 26(4), 631–653 (1979)
28. Rodrigues, R., Liskov, B., Castro, M.: BASE: Using Abstraction to Improve Fault Tolerance. ACM Transactions on Computer Systems 21(3) (2003)
29. Sandberg, R., et al.: Design and Implementation of the Sun Network Filesystem. In: Proceedings of the Summer 1985 USENIX Conference, Jun. 1985, pp. 119–130 (1985)
30. Schneider, F.: Implementing Fault-Tolerant Services using the State Machine Approach: a Tutorial. ACM Computing Surveys 22(4), 299–319 (1990)
31. Shein, B., et al.: NFSSTONE - A Network File Server Performance Benchmark. In: USENIX Summer '89 Conference Proceedings, pp. 269–274 (1989)
32. Vandiver, B., Liskov, B., Madden, S., Balakrishnan, H.: Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. In: Proceedings of SOSP 2007, Stevenson, WA (October 2007)

33. Wester, B., Cowling, J., Nightingale, E., Chen, P., Flinn, J., Liskov, B.: Tolerating Latency in Replicated State Machines through Client Speculation. In: Proceeding of the 6th NSDI, Boston, MA (April 2009)
34. Yin, J., Martin, J., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating Agreement from Execution for Byzantine Fault Tolerant Services. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (Oct. 2003)