Chapter 2
Replication Techniques for Availability

Robbert van Renesse and Rachid Guerraoui

Abstract The chapter studies how to provide clients with access to a replicated object that is logically indistinguishable from accessing a single yet highly available object. We study this problem under two different models. In the first, we assume that failures can be detected accurately. In the second we drop this assumption, making the model more realistic but also significantly more challenging. Under the first model, we present the primary-backup and chain replication techniques. Under the second model, we present techniques based on voting. We conclude with a discussion on reconfiguration.

2.1 Introduction

Replication is creating multiple copies of a possibly mutating object (file, file system, database, and so on) with the objective to provide high availability, high integrity, high performance, or any combination thereof. For high availability and integrity, the replicas need to be diverse, so failures are sufficiently independent. For high performance, there just needs to be a sufficient number of replicas in order to meet the load imposed on the replicated object.

In this chapter, we will focus on replication techniques that ensure high availability. In particular, we will study techniques that provide clients with access to a replicated object that is logically indistinguishable from accessing a single (non-replicated), yet highly available, object. This "indistinguishable from a single object" property is sometimes called linearizability, one-copy semantics, or simply consistency, and is ensured by enforcing a total order on client operations. Of course, such a strategy can only work under a restricted failure model. For example, if failed communication links partition the replicas, then it may be impossible to provide both availability and consistency for an object.

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 19–40, 2010.
© Springer-Verlag Berlin Heidelberg 2010

While a number of different replication techniques exists, two different approaches have become particularly well-known: *active replication* and *passive replication*. In active replication, also known as *state machine replication*, client operations are ordered by an ordering protocol and directly forwarded to a collection of replicas. Each replica individually executes the operation. Keeping the replicas consistent requires that processing be *deterministic*: given the same client operation and the same state, the same state update is produced by each replica.

In passive replication, also known as *primary-backup replication*, one of the replicas is designated *primary*. It executes the operations and sends the resulting state updates to each of the replicas (including itself), which, passively, apply the state updates in the order received. Note that in passive replication it is not necessary that operations be deterministic—typically, the primary will resolve non-determinism and produce state updates, which are deterministic.

These approaches have various advantages and drawbacks when compared with one another. If operations are compute intensive, then active replication can waste computational resources, but if state updates are large, passive replication can waste network bandwidth. Active replication cannot deal with non-deterministic processing but can mask failures without performance degradation, while passive replication may involve a detection and recovery delay in case the primary crashes.

Various hybrid solutions that combine both approaches are common. Some processing is executed on just one replica, while other processing is performed by all replicas. They are neither purely active nor passive approaches, and face different trade-offs.

This chapter avoids discussion of how operations are processed. Instead, it models an object's state by the sequence of operations. For example, if the object represents a bank account, we keep track of the history of deposit and withdraw operations, rather than of the running total. Doing so makes is easier to talk about consistency, as we can compare histories stored at different replicas and determine if one is a prefix of the other, or not. If all we had is a running total, then such a comparison would be impossible.

The chapter is organized as follows. In Section 2.2 we will present a convenient model of an unreplicated object. Then, using this model, we will describe two replication techniques that assume a simple failure model in Section 2.3. In Section 2.4 we will make the failure model more realistic (and more challenging) while discussing how to adapt the replication techniques accordingly. Section 2.5 discusses approaches for reconfiguring a replicated object. Finally, Section 2.6 concludes with a brief comparison of the techniques discussed in this chapter.

2.2 Model

For simplicity, we will assume that there is only one object. We find it convenient to model an object as a finite sequence of uniquely identified *deltas*, $H = \langle d_1, d_2, ..., d_b \rangle$, encoding a history of b updates applied to the object. A delta is a tuple $(update\ identifier, operation)$. A particular update identifier can only appear once in the history, although two different deltas may well contain the

same operation. A client may add a delta by invoking `updateHist(`*operation*`)`. The update identifier for the delta is automatically generated. An invocation of `updateHist(`*operation*`)` is expected to return a new history that can be used to compute the response of the operation.

While convenient for specification, in practice most services would not maintain the history of operations, but instead only the state resulting from applying the operations to a well-defined initial state, while `updateHist()` would normally return a simple result.

2.2.1 Environment

We also have to define a model of the distributed environment. Initially we will assume that processors are *fail-stop* [17]. More specifically:

- a processor follows its specification until it crashes (we say it is faulty);
- a crashed processor does not perform any action (*e.g.*, does not recover);
- a crash is detected eventually by every correct (non-crashed) processor;
- no process is suspected of having crashed until after it actually crashes.

The environment is assumed to be asynchronous: message delays and processing delays are arbitrarily long, and clocks on the processors are not synchronized.

We assume the processors are totally ordered: $p_1 < p_2 < \dots$. We also assume that the network is point-to-point and FIFO:

- messages from the same source are delivered in the order sent;
- messages between correct processors are eventually delivered.

2.2.2 Specification

In the unreplicated case, a single server stores the history of the object. Figure 2.1 depicts the specification (pseudo-code) for the client and the server. The specification distinguishes the function that can be invoked by the application (`updateHist(`*operation*`)`) and the events that can be invoked by the underlying system.

In the face of concurrency, functions and events act like monitors: on a processor only one thread of control can execute at a time. By invoking **wait until** *condition* the thread releases control until the condition is satisfied. Examples of events include failure notifications and receipt of messages.

Function `updateHist(`*op*`)` generates a unique identifier (typically consisting of a client identifier and a sequence number incremented for each request), and then sends a request message to the server. The response from the server contains the unique identifier, as well as the server's copy of the history, and is added to the set *responses*. The client waits until the request identifier appears among the *responses* or until the server is reported having failed. The server simply adds a delta to the history for each new update request and returns the resulting history.

(a) **Client code**

```
var server initially "server address";
var responses initially ∅;

event failure(p) :
    if p = server then server := ⊥;

event receive("response", r) from p :
    responses := responses ∪ {(p, r)};

function updateHist(op) :
    if server ≠ ⊥ then
        uid := genUID();
        send("updateHist", (uid, op)) to server;
        wait until (·, (uid, ·)) ∈ responses ∨ server = ⊥ ;
        if ∃_H (·, (uid, H)) ∈ responses then return H;
    end
    return ERROR("unavailable");
```

(b) **Server code**

```
var H initially ⊥;

event receive("updateHist", (uid, op)) from client :
    H := H :: (uid, op);
    send("response", (uid, H)) to client;
```

Fig. 2.1 Code for an unreplicated object.

The code for the unreplicated case serves as the specification of the semantics that we want to preserve when replicating the object onto multiple servers. From this specification one may derive that:

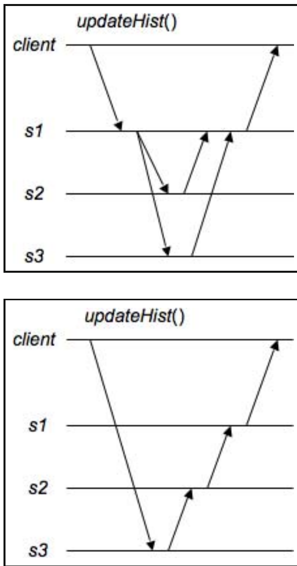
- Consider any two invocations `updateHist(`op_1`)` and `updateHist(`op_2`)` that return respectively histories H_1 and H_2. Either H_1 is a strict prefix of H_2 or vice versa.
- Furthermore, if `updateHist(`op_1`)` returns before `updateHist(`op_2`)` is invoked, then H_1 is a prefix of H_2.

Note that a client can also use `updateHist()` to query the state of the object by submitting a no-op update. We will look at optimizations for read-only operations in Section 2.3.3.

2.3 Fail-Stop Failure Model

The basic approach to replicating an object is as follows:

1. allocate a collection of processors (also called servers) $p_1, \ldots, p_n$;
2. place an empty history H_i on each p_i;
3. add updates in the same order to each H_i.



Primary-Backup

A client sends requests to the minimum server. The minimum server forwards the request to the other servers and awaits responses before responding to the client.

Chain Replication

A client sends update requests to the maximum server (*head*), which forwards the request to the next lower server until it reaches the minimum server (*tail*). The tail responds to the client.

Fig. 2.2 Normal case message patterns for Primary-Backup and Chain Replication.

In the following, we present two replication techniques, assuming a fail-stop model of the environment: *Primary-Backup* and *Chain Replication*, which are summarized in Figure 2.2.

2.3.1 Primary-Backup

In Primary-Backup (PB) [1, 3], perhaps the most common replication method in use today, the processor that has not crashed and that has the lowest identifier is designated *primary*. The remaining correct processors are called *backups*. During normal operation, a client sends an “updateHist” request to the primary and receives a response from the primary.

The client code (Figure 2.3) is similar to the unreplicated case, except that clients deal with the case of a failed primary. If the primary fails, the client determines the new primary to whom it retransmits its update. The client continues to do so until it receives a response or until there are no servers left.

Figure 2.4 shows the server code. We describe the underlying steps below.

1. Upon receipt of an “updateHist” request, the server may conclude correctly that it is the primary (because of accurate failure detection). If it had not yet detected the crash of a former primary itself, then it can do so now by removing all servers below itself from the list of servers that it maintains.
2. Next the server checks to see if the delta corresponding to the “updateHist” request is already in the history. This is possible because the primary may have

```
var servers initially {p_1, ..., p_n};
var responses initially ∅;

event failure(p) :
    servers := servers \ {p}

event receive("response", r) from p :
    responses := responses ∪ {(p, r)};

function updateHist(op) :
    uid := genUID();
    repeat
        if servers = ∅ then return ERROR("unavailable");
        primary := min(servers);
        send("updateHist", (uid, op)) to primary;
        wait until (·, (uid, ·)) ∈ responses ∨ primary ∉ servers ;
        if ∃_H (·, (uid, H)) ∈ responses then
            return H;
    end
```

Fig. 2.3 Client code for an object replicated using Primary-Backup.

received the update when it was still a backup. In the normal case, however, the delta is not in the history.
3. In either case, the primary sends a "sync" request containing the primary's desired history to all backups.
4. A backup, upon receipt of a "sync" request, first updates its estimate of the list of correct servers by removing all servers that are below the source of the request, i.e., the current primary.
5. The backup verifies that indeed the request came from the primary (minimum server on its list), as the request might be a tardy message from a server that was primary but that has since crashed. If the request came from a current server (which must be the primary), then the backup updates its history and sends a response.
6. When the primary received responses from all current backups, it updates its own copy of the history and sends the result as a response to the client.

It is important that the server does not respond to the client until after the server received responses from all available backups. To see why, say a client submits an "updateHist" request to the primary, and the primary adds the corresponding delta to its history and responds. If the server crashes before sending a "sync" request to the backups, the update may be lost even though the client receives a response. A new client will contact a new primary and miss the previous update, violating one-copy semantics.

The primary is allowed to stream multiple updates to the backups, without waiting for responses between updates. This allows for higher throughput. An important invariant of PB replication, however, is that whenever the primary responds to the client, the history of the primary is a prefix of the histories held by the backups. By implication, a response received by the client reflects a history that is stored at all

```
const me = "my address";
var H initially ⊥;
var servers initially {p_1, ..., p_n};
var responses initially ∅;

event failure(p) :
    servers := servers \ {p}

event receive("response", r) from p :
    responses := responses ∪ {(p, r)};

event receive("sync", (H', uid)) from primary :
    ∀p ∈ servers : p < primary ⇒ servers := servers \ {p};
    if primary ∈ servers then
        H := H';
        send("response", uid) to primary;
    end

function sync(H') :
    uid_2 := genUID();
    ∀p ∈ servers : send("sync", (uid_2, H')) to p;
    wait until ∀p ∈ servers : (p, uid_2) ∈ responses ;

event receive("updateHist", (uid, op)) from client :
    ∀p ∈ servers : p < me ⇒ servers := servers \ {p};
    H' := if (uid, ·) ∉ H then H :: (uid, op) else H ;
    sync(H');
    H := H';
    send("response", (uid, H)) to client;
```

Fig. 2.4 Server code for an object replicated using Primary-Backup.

available replicas. Without this invariant, a future "sync" request by a new primary might remove the client's update, violating consistency guarantees.

For simplicity, a "sync" request sends the entire history from the primary to the backup. In practice it is usually sufficient to send the update along with a collision-resistant hash of the history prior to applying the update. On receipt, a server checks to make sure that the hash matches its history, and if so appends the update. In the rare case that there is no match (which is possible in certain failure scenarios), then the primary and the backup have to synchronize the entire history using additional messages.

Why It Works

Consider first liveness (*i.e.*, the property that each `updateHist()` operation eventually terminates) and assume there is at least one correct replica. Suppose by contradiction that a correct client submits an "updateHist" request and never receives any response. Consider now the time after which all faulty replicas have crashed. By the fail-stop model, there is a time after which a correct primary is elected and known as such to the client. (Such a primary exists for we assume at least one correct replica). The client eventually submits its request to this correct primary. This

primary sends its "sync" message to all correct backups, which eventually respond. The primary responds back to the client: a contradiction.

For safety (*i.e.*, linearizability), suppose two invocations `updateHist`(op_1) and `updateHist`(op_2) return respectively histories H_1 and H_2. If both return results from the same primary, then clearly one is a prefix of the other. Moreover, H_1 is a prefix of H_2 if `updateHist`(op_2) is invoked after `updateHist`(op_1) has returned. Now assume they came from two primaries, p_1 returning H_1, and then p_2 returning H_2, such that $p_1 < p_2$. This means p_2 became primary only after p_1 crashed. For p_1 to return H_1, p_1 had to make sure all backups have acknowledged, and thus stored, H_1. No matter which processes become primary subsequently, all histories of p_2 must have H_1 as a prefix, including H_2.

For a complete treatment of the Primary-Backup technique and its correctness under various failure models, see [3].

2.3.2 Chain Replication

In Chain Replication (CR) [16], the servers are organized in a chain with a *head* (the maximum server) and a *tail* (the minimum server). A client sends update requests to the head, which forwards the request along the chain towards the tail. The tail responds to the client.

Figures 2.5 and 2.6 show the pseudo-code. Function `predecessor`(*servers*, p), returns the smallest $p' \in$ *servers* larger than p, or $\perp$ if p is the largest element in *servers*. Similarly, function `successor`(*servers*, p) returns the largest $p' \in$ *servers* smaller than p, or $\perp$ if p is the smallest element. As in PB replication, we have simplified the presentation by sending the entire history along the chain instead of only a collision-resistant hash and the update.

The CR technique simplifies the server code compared to PB replication. In particular, it is never necessary for a server to wait for other servers. An "updateHist" request received by the head can be applied immediately to the local history, forwarded (as a "sync" request) to the next server, and then forgotten. The head is responsible for ordering requests as they arrive from clients, but otherwise just serves as a backup. When a "sync" request arrives at the tail, the tail applies the update just like the other servers on the chain. Knowing that the update is now applied to all non-crashed replicas, the tail can respond to the client, finishing the entire update request.

No complicated recovery is necessary after a server fails. All that is necessary is for servers to keep track of who their successor and predecessor are. For this, the servers use two techniques. First, failure notification allow servers to update their estimate of the list of correct servers. Second, a server can make deductions based on the messages that it receives. For example, if a replica receives an update request from a client, the replica knows that any predecessors on the chain must have failed and that it is now the head of the chain.

In the face of failures, a client may have to retransmit an outstanding update request. Because an update can get lost anywhere in the chain, the client code of Figure 2.5 implements this by periodically retransmitting until a response is received.

```
const T = retransmission delay;
var servers initially {p_1, ..., p_n};
var responses initially ∅;

event failure(p) :
    servers := servers \ {p}

event receive("response", r) from p :
    responses := responses ∪ {(p, r)};

function updateHist(op) :
    uid := genUID();
    repeat
        if servers = ∅ then return ERROR("unavailable");
        head := max(servers);
        send("updateHist", (uid, op)) to head;
        wait up to T seconds until (·, (uid, ·)) ∈ responses;
        if ∃_H (·, (uid, H)) ∈ responses then
            return H;
    end
```

Fig. 2.5 Client code for an object replicated using Chain Replication.

The time between retransmissions is defined by a constant T, which in practice should be set to a value so that most responses are expected to be received within that amount of time. There is a trade-off: if T is set too short, unnecessary retransmissions would create additional load. On the other hand, the larger T, the longer it takes to recover from a failure. However, the actual value of T does not affect correctness.

As in PB replication, multiple update requests can be streamed for increased throughput. The important invariant maintained by the chain is that for any two servers, the history of the server with the lower identifier is a prefix of the server with the higher identifier. This is true at any point in time, even in the face of crashes, and thus simplifies recovery with respect to Primary-Backup.

Why It Works

The liveness and safety arguments are similar to those of PB replication. Assume there is at least one correct replica and consider a correct client that submits an "updateHist" request and never receives any response. Consider the time after which all faulty replicas have crashed and the chain is stable. The client eventually submits its request to the head and eventually gets a response from the tail (possibly the head and the tail are the same replica if only one server is correct): a contradiction.

For safety, suppose invocations `updateHist(`op_1`)` and `updateHist(`op_2`)` return histories H_1 and H_2 resp. If both return results from the same tail, then clearly one is a prefix of the other. Moreover, H_1 is a prefix of H_2 if `updateHist(`op_2`)` is invoked after `updateHist(`op_1`)` has returned. Now assume they came from two tails, p_1 returning H_1 and p_2 returning H_2, such that $p_1 < p_2$. This means p_2

```
const me = "my address";
var H initially ⊥;
var servers initially {p_1, ..., p_n};

event failure(p) :
    servers := servers\{p}

event receive("sync", (H', client, uid)) from prev :
    ∀p ∈ servers : me < p < prev ⇒ servers := servers\{p};
    if prev = predecessor(servers, me) then
        H := H';
        sync(H, client, uid);
    end

function sync(H', client, uid) :
    next = successor(servers, me);
    if next ≠ ⊥ then
        send("sync", (H', client, uid)) to next;
    else
        send("response", (uid, H)) to client;
    end

event receive("updateHist", (uid, op)) from client :
    ∀p ∈ servers : p > me ⇒ servers := servers\{p};
    if (uid, ·) ∉ H then H := H :: (uid, op);
    sync(H, client, uid);
```

Fig. 2.6 Server code for an object replicated using Chain Replication.

became tail only after p_1 crashed. Because of the invariant discussed above, at the time p_1 crashed p_2's history must have H_1 as a prefix. From there on, all histories of p_2 must have had H_1 as a prefix, including H_2.

For a complete treatment of Chain Replication and its correctness under the fail-stop model, see [16].

2.3.3 Queries

In many cases, one would like to distinguish *query operations*, operations that do not modify the state of the object, and optimize their performance. We explain below how this can be achieved in both PB and CR.

We provide a function `queryHist()` through which a client can consult a history without modifying it. The pseudo-code for the unreplicated case is depicted in Figure 2.7.

The `queryHist()` function also generates a unique identifier and sends a message to the server. The server simply returns the history for query operations. The following properties are now also ensured:

- Consider any two invocations (`updateHist(op)` or `queryHist()`) that return respectively histories H_1 and H_2. Either H_1 is a strict prefix of H_2 or vice et versa. Furthermore, if the first invocation returns before the second is invoked, then H_1 is a prefix of H_2.

(a) **Client code**

```
function queryHist() :
    if server ≠ ⊥ then
        uid := genUID();
        send("queryHist", uid) to server;
        wait until (·,(uid,·)) ∈ responses ∨ server = ⊥ ;
        if ∃_H (·,(uid,H)) ∈ responses then return H;
    end
    return ERROR("unavailable");
```

(b) **Server code**

```
event receive("queryHist", uid) from client :
    send("response", (uid,H)) to client;
```

Fig. 2.7 Code for the query function of an unreplicated object.

(a) **Client code**

```
function queryHist() :
    uid := genUID();
    repeat
        if servers = ∅ then return ERROR("unavailable");
        primary := min(servers);
        send("queryHist", uid) to primary;
        wait until (·,(uid,H)) ∈ responses ∨ primary ∉ servers ;
        if ∃_H (·,(uid,H)) ∈ responses then
            return H;
    end
```

(b) **Server code**

```
var first initially true;

event receive("queryHist", uid) from client :
    ∀p ∈ servers : p < me ⇒ servers := servers\{p};
    if first then
        sync(H);
        first := false;
    end
    send("response", (uid,H)) to client;
```

Fig. 2.8 Code for the query function of a replicated object using Primary-Backup.

In PB, during normal operation, a client c sends requests ("updateHist" or "queryHist") to the primary and receives responses from the primary. The client code for PB's `queryHist` function is given in Figure 2.8(a). Figure 2.8(b) shows the server code. Both are almost the same as in the unreplicated case. Normally the primary can respond immediately. There is, however, a special case. If a server that used to be a backup but is now a primary receives a request for the first time, it must synchronize its state with that of the other backups. The reason for this is that the current

(a) **Client code**

```
function queryHist() :
    uid := genUID();
    repeat
        if servers = ∅ then return ERROR("unavailable");
        tail := min(servers);
        send("queryHist", uid) to tail;
        wait until (·,(uid,·)) ∈ responses ∨ tail ∉ servers ;
        if ∃_H (·,(uid,H)) ∈ responses then
            return H;
    end
```

(b) **Server code**

```
event receive("queryHist", uid) from client :
    ∀p ∈ servers : p < me ⇒ servers := servers\{p};
    send("response", (uid,H)) to client;
```

Fig. 2.9 Code for the query function of an object replicated using Chain Replication.

primary may have received deltas from the former primary that are not included in some of the backup servers. If they would be reflected in the response to the client, a crash of this new primary could lose deltas that a client has seen. Synchronizing state on the first query operation ensures that the backups have the same state as the primary.

Figure 2.9 shows the client and server code for the `queryHist` function in the context of Chain Replication. It is the tail that responds to the client. The tail handles a query request in much the same way as in the unreplicated case, responding immediately to the client. Unlike update requests, the client need only interact with the tail of the chain for queries.

2.4 Crash Failure Model

So far we assumed a fail-stop model. In particular, (1) if a server fails, all processes (clients and servers) eventually detect the failure, and (2) no process detects the failure of a server unless that server has actually failed. It is common to call this *perfect failure detection* [4]. In practice, failure detection is achieved using timeouts: Every server is periodically pinged and if it does not respond within a predetermined time period, the failure of the server is suspected. Unfortunately, unless there is a known bound on message latency, such a mechanism does not implement perfect failure detection. While crashes are correctly detected eventually, correct servers may be falsely suspected.

False failure detections might partition the distributed system into two disjoint subsets each containing clients and servers, that is, the processes in each of the

subsets might wrongly consider those in the other subset as having failed. In each partition, the processes might elect a primary under PB replication (resp. construct a chain in CR) and clients in different partitions might see divergent histories as a result. For instance, one client might deposit an amount of money in an account after accessing the first partition, and, later, another client, which accesses the same account but within the other partition, might not see the deposited amount.

To avoid such inconsistencies, implementations of Primary-Backup and Chain Replication have to use large timeouts to make the probability of false detection low. The larger the timeout period, the larger the response time of a request directly following a failure. Setting the timeout period low increases the probability of false detections. This is not a good trade-off, and thus it would be better if we devised a replication technique that can tolerate false detections.

In this section, we present replication techniques that do not attempt to detect failures; instead the techniques seamlessly mask failures altogether. The first such technique is Stake Replication. The second technique, Broker Replication, builds on the first. Both techniques are summarized in Figure 2.10. Before describing the techniques in more detail, we will briefly review the quorum concept.

2.4.1 Quorums

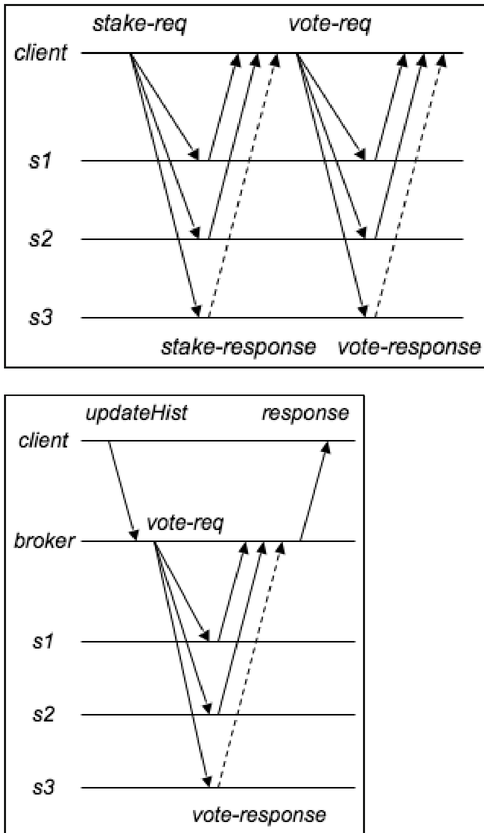
Consistency and availability of a replicated object can be preserved in the face of false failure detections using a mathematical abstraction called *quorums* [18, 9, 19]. A quorum system is a set of subsets of processes, each called a quorum, such that the following properties are satisfied (see [8, 14] for more formal treatments):

Consistency: any two quorums intersect in at least one process;
Availability: at least one of the quorums (which ones is unknown) contains processes that never crash.

A simple instantiation of quorums is the following. There are n processes, of which fewer than $n/2$ are allowed to crash. Quorums then are all sets that have $\lceil \frac{n+1}{2} \rceil$ processes. It is easy to verify that this construction satisfies Consistency and Availability.

2.4.2 Stake Replication

In Stake Replication (SR), the division of labor between clients and servers is much different from before. The servers are still responsible for ordering client operations and making sure that there is a persistent history of deltas. They do so, for each position in the history, by voting on what the next delta should be. Once a quorum of servers voted for the same delta, then the delta is decided and permanent. However, servers not in that quorum may have voted differently or not at all. This is the case for every individual delta, and thus none of the servers knows what the history is. They only know how they voted for each delta.



Stake Replication

A client broadcasts first a "stake-request" to all servers. Upon successful completing, the client requests all servers to vote.

Broker Replication

A client sends an update request to a broker. Brokers use an optimized version of Stake Replication (eliminating "stake-request" messages in the common case) before responding to clients.

Fig. 2.10 Normal case message patterns for Stake Replication and Broker Replication.

In the SR implementation that we will present, the clients reconstruct the history from examining the votes rather than by obtaining the history directly from the servers as before. The servers do not communicate with one another; they only respond to requests from clients. Because the technique uses quorums, the technique can tolerate at most $\lfloor (n-1)/2 \rfloor$ failures given n servers.

Figures 2.11 to 2.13 show the code for Stake Replication. The `updateHist()` function, in Figure 2.11, simply invokes another function `consensus()` for each delta in the history. The client passes the desired delta to `consensus()`, but because of concurrent updates introduced by other clients, the actual delta added to the history may be different from the desired one. The client repeats until the desired delta has been successfully added to the history, and then the function can return the history.

The secret sauce is in the client's `consensus()` function of Figure 2.12 and the server code of Figure 2.13. Define *version* to be the length of the history, $|H|$. The consensus protocol only deals with one version at a time. Within a version,

```
var H initially ⊥;

function updateHist(op) :
    uid := genUID();
    repeat
        delta := consensus(|H|, uid, op);
        H := H :: delta;
        if delta = (uid, op) then
            return H;
    end
```

Fig. 2.11 Client code for Stake Replication, part 1.

there is a notion of logical time that we call *stakes*. A stake is a tuple consisting of a round number and the identifier of the client that owns the stake, so that two different clients cannot own the same stake. Stakes are lexicographically ordered, first by round, and then by client identifier.

A *vote* is a tuple consisting of a stake and a delta. In the implementation that we are describing, the client that owns the stake picks the delta, and therefore guarantees that two votes with the same stake will also have the same delta. The objective of a client is to get a quorum of servers to vote on the same stake and delta. We say that a vote is *decided* when this objective is reached. The consensus protocol runs in a loop, trying monotonically increasing stakes, until its corresponding vote is decided.

Each server maintains a stake and a vote for each version. In each iteration of the client's loop, the client first creates a new stake and tries to get a quorum of servers to progress to that particular stake. For this, the client broadcasts a "stake-request" message to all servers, containing the current version and the stake. Upon receipt, the server checks to see what stake it is at for that particular version. If the client's stake is further along, then the server advances its stake accordingly. In any case, it returns the stake that it now holds, as well as its last vote for that version. The client waits until it has received more than $n/2$ responses to ensure that it has responses from a quorum of servers.. Because fewer than $n/2$ are faulty, it will eventually receive a sufficient number of responses. Also, all servers that respond have advanced their stake to at least the client's stake.

The client now determines the maximum stake among the responses. If this maximum is further along than the client's, then the client declares failure, and tries again with a new stake. However, if all servers are at the client's stake, then the client tries to get all servers to vote using its current stake. However, for reasons explained below, it cannot just use the delta that it is trying to add to the history. Instead, it determines among the responses from the servers the maximum vote (that is, the vote with the maximum stake). If that maximum vote is $((0,0),\perp)$, meaning that no server has voted as of yet, then the client can use its own delta. But if not, the client uses the delta with the maximum stake.

Subsequently, the client tries to get all servers to vote on its selected delta for the given stake by broadcasting a "vote-request" message. Each server, upon re-

```
const servers = {p_1, ..., p_n};
const me = genUID();
var sResponses initially ∅;
var vResponses initially ∅;

event receive ("stake-response", r) from p :
    sResponses := sResponses ∪ {(p, r)};

event receive ("vote-response", r) from p :
    vResponses := vResponses ∪ {(p, r)};

function staked (version, stake) :
    return {(p, (vn, s, s', vote)) ∈ sResponses | vn = version ∧ s = stake};

function voted (version, stake) :
    return {(p, (vn, s, c)) ∈ vResponses | vn = version ∧ s = stake};

function consensus (version, uid, op) :
    round := 0;
    repeat
        stake := (round, me);
        broadcast ("stake-request", (version, stake)) to servers;
        wait until |staked (stake)| > n/2 ;
        S := staked (stake);
        maxStake := max{s' | (p, (vn, s, s', vote)) ∈ S};
        if stake = maxStake then
            maxVote := max{vote | (p, (vn, s, s', vote)) ∈ S};
            if maxVote = ((0, 0), ⊥) then
                delta := (uid, op);
            else
                delta := maxVote.delta;
            broadcast ("vote-request", (version, stake, delta)) to servers;
            wait until |voted (stake)| > n/2 ;
            V := {(p, (vn, s, c)) ∈ voted (stake) | c = ACCEPTED};
            if |V| > n/2 then
                return delta;
        end
        round := maxStake.round + 1;
    end
```

Fig. 2.12 Client code for Stake Replication, part 2.

ceipt, checks to see if it has not advanced its stake (because of a concurrent "stake-request" by another client). If so, the server votes as requested and responds with an `ACCEPTED` message. If the server did advance its stake, the server abstains from voting and responds `DENIED`. The client again waits for more than $n/2$ responses. If more than $n/2$ servers accepted the vote, then the vote is decided and `consensus()` returns the corresponding delta. If not, the client tries again with a new stake.

Why It Works

Again, we consider both liveness and safety. This technique, in fact, cannot guarantee termination of `updateHist()`, because two clients can alternate advancing

```
var stakes[ ] initially (0,0);
var votes[ ] initially ((0,0),⊥);

event receive ("stake-request", (vn,s)) from client :
    if s > stakes[vn] then
        stakes[vn] := s;
    send ("stake-response", (vn,s,stakes[vn],votes[vn])) to client;

event receive ("vote-request", (vn,s,vote)) from client :
    if s = stakes[vn] then
        votes[vn] := vote;
        send ("vote-response", (vn,s,ACCEPTED)) to client;
    else
        send ("vote-response", (vn,s,DENIED)) to client;
```

Fig. 2.13 Server code for Stake Replication.

stakes for a version without ever getting the servers to vote for one of their stakes. Indeed, no replication protocol can be designed that is guaranteed to terminate (a consequence of [7]). However, we can show that in the absence of contention, the protocol that we described is guaranteed to terminate.

To wit, consider any particular version, and assume only one client is active. It will send a "stake-request" to all servers, and wait for a response from more than $n/2$ servers. Because fewer than $n/2$ servers are faulty, eventually it will receive the required responses. If some of the servers have advanced further than the client, the client chooses a new stake that is further than any in the responses. Because there are no other clients active by assumption, eventually the client will be able to advance its stake sufficiently far along so that more than $n/2$ of the servers will advance to the client's stake and no further. At this point the subsequent "vote-request" is also guaranteed to succeed, with all servers accepting the stake and delta selected by the client.

For safety, we have to make sure that no two votes, with different deltas, can be decided for the same version. Stake Replication guarantees safety through the following invariant: if a vote (s,v) is decided, then any vote (s',v') (decided or not) with $s' > s$ has $v' = v$. This invariant depends on two important features of the protocol. First, when a client receives responses to its "stake-request" from a quorum of servers, it knows that the servers that responded can no longer vote using lower stakes than the one it requested. Second, it also collected for those servers the maximum stake that they voted on thus far. If any vote has decided, then, by quorum intersection, the responses must include a response from one of those servers. By clients always selecting the maximum vote, the invariant is guaranteed. If the maximum vote is $((0,0),\perp)$, it is guaranteed that no stake lower than the client's can have decided, and thus the client is free to choose any delta in that case, without fear of violating the invariant.

For good examples and treatments of Stake Replication techniques, see [6, 11].

```
var H initially ⊥;
var requests initially ∅;

event receive("updateHist", delta) from client :
    requests := requests ∪ {(client, delta)};

function mainLoop() :
    repeat
        wait until requests ≠ ∅ ;
        r := selectOne(requests);
        requests := requests \ {r};
        while r.delta ∉ H do
            delta := consensus(|H|, r.delta.uid, r.delta.op);
            H := H :: delta;
        end
        send("response", (r.delta.uid, H)) to r.client;
    end
```

Fig. 2.14 Broker code for Broker Replication, which is an extension of the client code of Stake Replication.

2.4.3 Broker Replication

A disadvantage of Stake Replication is that clients need to reconstruct the history. This is inconvenient, and can involve considerable overhead. To remedy this, most SR implementations only support an "overwrite" delta, such that the last delta in a history completely determines the state of the replicated object. Then it is unnecessary to reconstruct the entire history, but only the last delta.

Broker Replication (BR) is an extension of Stake Replication that overcomes this disadvantage of SR in a different way. Unlike SR, BR does incorporate a collection of servers that maintain the state. BR is a three-tiered solution. One tier contains servers just like in Stake Replication. The middle-tier contains a set of processes that we shall call *brokers*. They run essentially the client code of SR in an infinite loop, deciding deltas and maintaining the resulting history (see Figure 2.14). The remaining tier are the ultimate clients that send requests to, and receive responses from, brokers. To tolerate f failures, the middle-tier must include $f+1$ brokers. (To save on hardware and messages, the broker processes usually run on the same machines that are used for running server processes.)

Clients can go through any broker in order to update and access history. However, contention between brokers is avoided, and performance thus improved, if all clients used the same broker. To accomplish this, BR implementations use a weak leader election protocol to elect a leader among available brokers. The election is weak in the sense that the protocol may accidentally elect more than one leader at a time. This does not affect safety, but does affect performance and liveness.

Figure 2.15 shows a simple example of client code that uses weak leader election. The event up(p) signifies that broker p is now believed to be reachable by the client, while down(p) signifies that the connection between the client and p is suspicious. These events may provide mistaken information, but we assume that if a broker has

```
var brokers initially ∅;
var responses initially ∅;

event up(p) :
    brokers := brokers ∪ {p};

event down(p) :
    brokers := brokers\{p};

event receive("response", r) from p :
    responses := responses ∪ {(p,r)};

function updateHist(op) :
    uid := genUID();
    repeat
        wait until brokers ≠ ∅ ;
        leader := min(brokers);
        send("updateHist", (uid,op)) to leader;
        wait until (·,(uid,·)) ∈ responses ∨ leader ∉ brokers ;
        if ∃_H (·,(uid,H)) ∈ responses then
            return H;
    end
```

Fig. 2.15 Client code for Broker Replication.

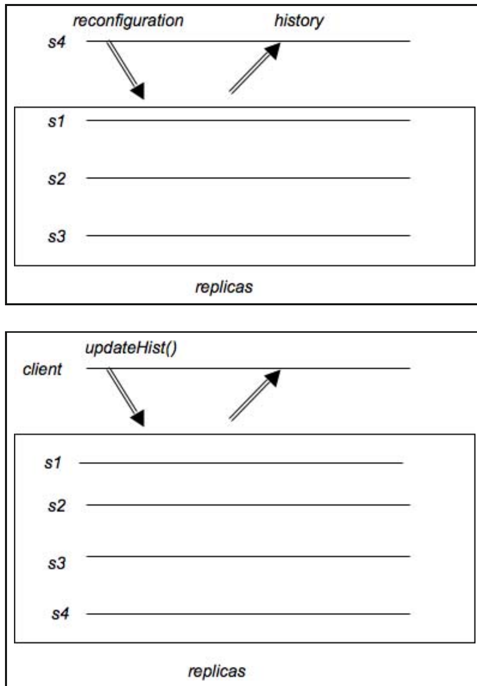
in fact crashed, then it will eventually be marked down and never up again. The client uses the broker with the minimum identifier among the brokers that it thinks are reachable.

An important optimization of the BR technique is to use a single stake for multiple operations. That is, when a broker receives an "updateHist" request, it first tries to re-use its last stake. It broadcasts a "vote-request" message to all servers. If it receives `ACCEPTED` responses from a quorum of servers, then the request completes and the broker can respond to the client. If not, then the broker has to establish a new stake. Assuming the weak leader election mechanism works reasonably well, establishing a stake will be a rare event.

Why It Works

Consider liveness of `updateHist()`. If there is a broker that never crashes, there is a lowest broker b that never crashes. If no broker lower than b completes the client's request, then b will eventually receive and execute the request. The liveness thus depends on the liveness of SR. As updates are performed by brokers, safety is inherited from the brokers as well.

Good examples of the BR technique are Viewstamped Replication [15] and Paxos [13].



New server as a client

To join (again), a server acts as a client of the replicated system and issues a reconfiguration request.

New server is now in

Another client can now make use of the new server.

Fig. 2.16 Reconfiguration.

2.5 Recovery and Reconfiguration

So far, we assumed that a process that fails does not recover. From a practical perspective, this is too strict a limitation, especially for long lived services that need to be permanently available. One would typically seek for techniques where a replica that recovers after crashing is integrated again into the system.

In the PB and CR techniques, a failed server is removed from the configuration altogether. On the other hand, in theory, the SR technique supports reconfiguration already. If a server stores its state on disk, then a crash followed by a recovery is not technically a crash, but appears as a transient network outage during which the server could not be reached. However, some servers may never be able to recover.

We want to be able to dynamically add servers to a replicated service. A new replica needs to obtain a unique identifier. Also, except for the SR technique, each server (or broker in the case of BR) needs to obtain a copy of the current version of the history. A general way of handling configuration changes is to make the configuration part of the replicated object, and add special update operations to change the configuration [12]. A server initially acts like a client, executing the operation `updateHist(`ADD_SERVER(*network address*)`)`. The operation returns the current history, which the new server adopts as its own current history. The identifier of

the server is simply the number of ADD_SERVER operations in the history, guaranteeing uniqueness. At this point, the server is fully integrated into the system.

Treating configuration update requests the same as normal update requests precisely ensures that the joining of this replica is totally ordered with other requests and this is key to ensuring consistency without hampering availability [12]. When completed, clients are informed about the new configuration (Figure 2.16). The same technique can also be used to remove servers from a configuration.

Many *group communication systems* such as Isis [2] integrate replication and reconfiguration mechanisms into a single tool for simplifying development of highly available services. For a treatment of group communication systems, see [5, 10].

Table 2.1 Comparison between the replication techniques discussed in this chapter. The table shows (normal case) latency in number of rounds, overhead in number of messages per update operation, and the number of failures tolerated. Here n is the number of servers.

technique	# rounds	# messages	# failures tolerated
PB	4	$2n$	$n-1$
CR	$n+1$	$n+1$	$n-1$
SR	4	$4n$	$\lfloor (n-1)/2 \rfloor$
BR	4	$2n$	$\lfloor (n-1)/2 \rfloor$

2.6 Conclusion

In this chapter we discussed four techniques to replication: Primary-Backup, Chain Replication, Stake Replication, and Broker Replication. Table 2.1 compares the four techniques with respect to update operations. Latency and message overheads in this table are simplified, and the extensive literature on replication techniques discusses many variants of these basic techniques. The first two are appropriate only for environments in which crash failures can be accurately detected, and depend on recovery for continued availability. The latter two techniques are significantly more robust, masking crash failures without attempting to detect them, but tolerate about half as many failures for the same number of servers.

References

1. Alsberg, P., Day, J.: A principle for resilient sharing of distributed resources. In: Proc. of the 2nd Int. Conf. on Software Engineering, pp. 627–644 (Oct. 1976)
2. Birman, K.P., Joseph, T.A.: Exploiting virtual synchrony in distributed systems. In: Proc. of the 11th ACM Symp. on Operating Systems Principles, Austin, TX, pp. 123–138 (Nov. 1987)
3. Budhiraja, N., Marzullo, K., Schneider, F., Toueg, S.: The primary-backup approach. In: Mullender, S. (ed.) Distributed systems, 2nd edn., ACM Press, New York (1993)
4. Chandra, T., Toueg, S.: Unreliable failure detectors for asynchronous systems. In: Proc. of the 11th ACM Symp. on Principles of Distributed Computing, pp. 325–340. ACM SIGOPS-SIGACT, Montreal (Aug. 1991)

5. Chockler, G., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Computing Surveys 33, 427–469 (1999)
6. El Abbadi, A., Skeen, D., Cristian, F.: An efficient, fault-tolerant protocol for replicated data management. In: Proc. of the 4th ACM Symp. on Principles of Database Systems, pp. 215–229. ACM SIGACT, Portland (Mar. 1985)
7. Fischer, M., Lynch, N., Patterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
8. Garcia-Molina, H., Barbara, D.: How to assign votes in a distributed system. J. ACM 32(4), 841–860 (1985)
9. Gifford, D.: Weighted voting for replicated data. In: Proc. of the 7th ACM Symp. on Operating Systems Principles, pp. 150–162. ACM SIGOPS, Pacific Grove (Dec. 1979)
10. Guerraoui, R., Schiper, A.: Software-based replication for fault tolerance. IEEE Computer 30(4) (1997)
11. Herlihy, M.: A quorum consensus replication method for abstract data types. Trans. on Computer Systems 4(1), 32–53 (1986)
12. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. Trans. on Programming Languages and Systems 6(2), 254–280 (1984)
13. Lamport, L.: The part-time parliament. Trans. on Computer Systems 16(2), 133–169 (1998)
14. Naor, M., Wool, A.: The load, capacity, and availabiliity of quorum systems. SIAM Journal on Computing 27(2), 423–447 (1998)
15. Oki, B., Liskov, B.: Viewstamped replication: A general primary-copy method to support highly-available distributed systems. In: Proc. of the 7th ACM Symp. on Principles of Distributed Computing, pp. 8–17. ACM SIGOPS-SIGACT, Ontario (Aug. 1988)
16. van Renesse, R., Schneider, F.: Chain Replication for supporting high throughput and availability. In: Sixth Symposium on Operating Systems Design and Implementation (OSDI '04), San Francisco, CA (Dec. 2004)
17. Schlichting, R., Schneider, F.: Fail-stop processors: an approach to designing fault-tolerant computing systems. Trans. on Computer Systems 1(3), 222–238 (1983)
18. Thomas, R.: A solution to the concurrency control problem for multiple copy data bases. In: Proc. of COMPCON'78, pp. 88–93 (1978)
19. Thomas, R.: A majority consensus approach to concurrency control for multiple copy database. ACM Trans. Database Syst. 4(2), 180–209 (1979)