Chapter 7

Data Replication and Resiliency

In the last chapter, we discussed the problem of making a user-level action into an atomic one. A user-level action is a logical operation that accesses or modifies many data objects. The goal of an atomic action is to ensure that either the action completes successfully, or it appears as if the action had not executed at all. That is, the state of partially executed actions should never be visible, even if failures occur. We saw that if a node fails, making even some of the data objects required by the action unavailable, there was nothing else to do but to "abort" the action and make it appear as if nothing had happened.

In this chapter, we will discuss a different approach where the action can be completed successfully even if some failures occur in the system. That is, we want the action to be *resilient* to failures. The goal is still to execute the actions atomically, with the difference that we are interested in successful completion, rather than rollback, if failures occur. Due to this, the techniques employed are quite different from the techniques for supporting atomic actions discussed in the previous chapter.

Clearly, if a data object resides at a single node, then nothing can be done to successfully complete an action which needs that data item, if that node fails. Hence to be able to finish an operation despite failures of nodes, data items need to be replicated on many nodes, such that failures of a few nodes do not make some data item inaccessible to user operations.

Data replication, though it provides resiliency against failures, introduces new problems of consistency and replica management. Since the purpose of replication is to provide tolerance against failures, the replication should not be visible at the user or action level. For performing actions, it should appear as if there is a single copy of

each data item. An action will perform operations on the logical data items, and the underlying system will map it to operations on the multiple copies of the data items. To be correct, the mapping must ensure that the concurrent execution of actions on replicated data is equivalent to a correct execution on non-replicated data. That is, the execution should be equivalent to a serial execution of actions on non-replicated data. This correctness property is called the *one-copy serializability* criterion. The methods to manage replicated data such that the one-copy serializability criterion is satisfied are called *replica control algorithms*. One-copy serializability requires that the different copies of a data object must be in a *mutually consistent* state so that the user actions get the same view of the data object. Due to this consistency requirement, replica control algorithms are also called *consistency control algorithms*.

In a system with replication, different data items are replicated on different nodes. Each node manages some copies of some data items. For simplicity of exposition, we will assume that all the data objects are replicated on all the nodes in the system under consideration.

There are two types of failures that need to be handled by a replica control algorithm: node failures and communication failures. Node failures cause copies of the data on that node to become inaccessible. The rest of the network is connected and the remaining copies of the data objects are available. The replica control algorithms have to ensure that even if some sites fail, making some copies of a data item unavailable, the operations on the logical data can be performed, and the one-copy serializability criterion is satisfied.

The second failure, which is a lot more disruptive, is a communication failure leading to network partitioning. In this, nodes and links fail in a manner such that the remaining nodes are partitioned into *groups*. Nodes in each group or *partition* can communicate with each other, but cannot communicate with nodes of the other group. Ideally, a replica control protocol should also maintain one-copy serializability under network partitioning. Clearly, if no restriction is placed on processing in different partitions, then the mutual consistency of the different copies cannot be preserved when the network reconnects, and a user operation may get a different view of the data object depending on which copies of the data object it accesses, thereby violating the one-copy serializability criterion. A replica control protocol that can handle partitions must place restrictions on processing in different partitions so that mutual consistency is not violated. That the protocol has to do this without any communication between different partitions is one of the major challenges.

In this chapter, we will describe some replica control methods to mask the failures in a system with replication. Replica control methods can be *optimistic* or *pessimistic* [DGS85]. In optimistic strategies, if network partitioning occurs, no restriction is placed on processing in any partition in the hope that operations

being executed in different partitions will not conflict. The pessimistic strategies, on the other hand, prevent inconsistencies from occurring by limiting access to data. During partitioning, each partition makes the worst-case assumptions about what is happening in other partitions, and operates under these pessimistic assumptions. There are three common pessimistic approaches for replica control: primary site, active replication, and voting. We will discuss all three approaches during the course of the chapter.

The focus of the chapter is how to make data resilient to node failures (by replicating the data). We do not discuss the effect of failure on the process which performs the actions, and which wants to access the data. For now, we assume that the processes executing the actions remain alive; failures only make some copies of the data unavailable. How to make the processes themselves resilient to failures is the topic of the next chapter.

7.1 Optimistic Approaches

As mentioned above, optimistic strategies do not place any restriction on processing, if a network partitioning occurs, in the optimistic hope that operations being executed in different partitions will not conflict. Under this optimistic assumption, the serializability in each group can be preserved, but overall processing may not satisfy the one-copy serializability criterion, and global inconsistencies may arise (i.e., copies from different partitions may not be mutually consistent). If global inconsistencies arise, then optimistic strategies try to resolve them after different partitions join and are able to communicate with each other. The different optimistic strategies differ from each other in how they resolve the inconsistencies, since there is really not much that needs to be done by these strategies during the processing. In this section, we will briefly discuss some optimistic approaches. The pessimistic approaches will be discussed in later sections, and will form the bulk of this chapter.

During a network partitioning, if operations are performed in each partition independently, the copies of data in different partitions may be inconsistent. When the partitions rejoin, such inconsistencies have to be detected, and resolved, if possible. One approach that can be used to detect inconsistencies that occur due to write operations in different partitions is *version vectors*. This technique was employed in the LOCUS operating system [S+83].

In this approach, files are treated as the basic data objects, and are the unit of replication. A file f can have many copies, all on different nodes. An update is said to originate from a node i if the request from the user arrived at node i. Whenever an update is to be performed on f, all copies of f that are accessible from the node from

where the update request originated are updated. Each copy of a file has associated with it a *version vector*, whose size is n, where n is the number of sites at which the file is stored. The version vector V of a copy of the file f represents the number of updates originating at different nodes that were performed on this copy. That is, at a node j, the ith vector entry v_i keeps count of the number of updates originating from site i that were performed on the copy of f at the node j.

Clearly, if the network is fully connected, then each update will be performed on each copy of the file, and the version vector of each copy will be the same. However, if a partitioning occurs, then the version vectors of the different copies in different partitions may diverge, depending on the nature of operations performed in different partitions.

A vector V of a copy of a file is said to *dominate* another vector V' of another copy of the same file, if $v_i \geq v'_i$ for all $i = 1, ..., n$. If a vector V dominates V', it indicates that for every node j, more updates made from j were performed on the copy with V, as compared to the copy with V'. In other words, updates seen by the copy with V' are a subset of updates seen by the copy with the version vector V. Two vectors are said to *conflict* if neither dominates. This represents the case where the copies have seen different updates.

When two groups join and are able to communicate with each other, vectors are compared. If a vector of one group dominates the other, then though this situation also represents an inconsistency (the states of the two copies are not the same), this inconsistency can be resolved easily by copying the file with version V onto the copy with version V', since the copy with V is a more recent copy of the file than the copy with V'. If the vectors of the groups that are joining conflict, then there is no straightforward way to resolve the inconsistency, and the version vector approach leaves it to the manager of the system to manually do whatever is necessary.

In other words, in this optimistic approach, if after partitioning the operations performed in the different operations are non-conflicting, then the version vector method provides an approach for combining the different versions in different partitions into a single consistent version after the partitions rejoin. But if the operations performed in the different groups conflict, then no mechanism is provided to ensure that this is resolved and mutual consistency is violated.

Consider the example of a three node network whose partition graph is shown in Fig. 7.1 [DGS85]. The nodes A, B, and C have initially the same vector, each entry being 0 (we consider a single file case). The system partitions into two groups, one containing nodes A and B, and the other containing the node C. Now suppose the node A makes two updates on the file. Since A and B are connected, the version vectors of nodes A and B will be $< 2, 0, 0 >$, while the version vector of node C is $< 0, 0, 0 >$. Now suppose node B splits off with node A and joins node C. Since

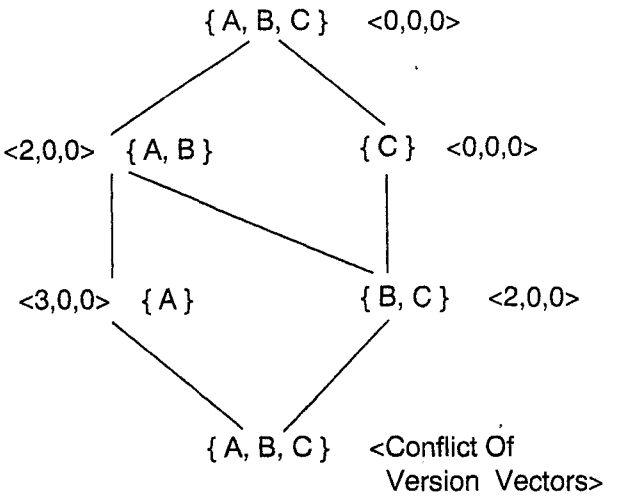


Figure 7.1: Example with version vectors

the version vector of node B dominates the version vector of node C, the conflict is resolved by copying the file from node B to node C (along with the version vector). Now during this grouping, suppose node A and node C both make update requests. The request of node A will make the version vector of node A $< 3, 0, 0 >$, as the update by C will have no effect on the version vector at node A. Similarly, the version vector at node B or C will be $< 2, 0, 1 >$. Now if these two groups join, the version vectors conflict, since no vector dominates the other. This has to be resolved manually.

Version vectors can detect only inconsistencies arising due to updates on a single file. They cannot detect read-write conflict, as the reads on a file are not recorded. If an action or a transaction accesses multiple files, as is common in databases, then besides the write-write conflict, the read-write conflict also has to be detected in order to detect the violations of transaction serializability. Consequently, for such applications, the version vectors approach is not suitable.

An extension of this approach was proposed in which both reads and writes of transactions are logged [Dav84, DGS85]. When the different groups rejoin, then a precedence graph is formed to detect inconsistencies. In order to be able to construct a precedence graph, each partition maintains a log of all reads and writes on data items. It is assumed that a transaction always reads a data item before writing it, and each partition follows some concurrency control protocol such that transactions in the partition are serializable. For a partition i, let the serialization order (i.e., the serial

order to which the execution of the transactions is equivalent) of the transactions be $T_i1, T_i2, ..., T_in$.

When the partitions rejoin, a precedence graph is constructed as follows. Transactions are nodes in this graph, and edges represent dependencies between transactions. For two transactions T_ij and T_ik in the same partition i, an edge is added from T_ij to T_ik if (a) T_ik has read a value produced by T_ij, or (b) T_ij read a value that was later modified by T_ik. The concurrency control protocol of the partition will always ensure that the precedence graph of transactions in a partition is always acyclic. To complete the precedence graph, conflicts between transactions of different partitions must also be represented. An edge is added from node T_ij (representing a transaction in partition i) to T_lk, $i \neq l$, if T_ij has read an item written by T_lk. This edge represents a read-write conflict between the two transactions operating in two separate partitions. A write-write conflict will be reflected as two read-write conflict edges.

If there is no conflict between transactions of different partitions, the precedence graph will be acyclic. If the graph is not acyclic, it means that there are conflicts. Conflicts are resolved by aborting some transactions till the precedence graph is acyclic. There are many different ways to select the transactions to be aborted [Dav84]. Once the graph is acyclic, all inconsistencies have been resolved and the databases of the different partitions can be merged. Note that aborting transactions in order to resolve inconsistencies arising due to the optimistic strategy may require that committed transactions also be aborted. That is, with partitioning, a transaction commit is not final, as it may need to be aborted later when the partitions rejoin.

7.2 Primary Site Approach

In this section, we describe the *primary site approach*, which is a pessimistic strategy for managing replication to support resiliency [AD76]. This approach has been used in a variety of contexts, not just for supporting data resiliency. In a distributed system, this approach works well if only nodes can fail, or if both nodes and communication links can fail but node failures can be distinguished from network partitions. We will first describe the general strategy, and then one particular implementation of resilient data objects using this approach.

7.2.1 Basic Approach

The goal of the primary site approach is to continue providing access to the data (in general, any resource) even if some nodes or links in the system fail. For now, we

assume that only node failures occur in the system, and the operational nodes stay connected. The goal now is to ensure that operations on the data can be performed even if up to k nodes in the system fail. That is, the goal is to support *k-resilient* data objects.

For supporting *k-resilient* data, the data is replicated on at least $k+1$ nodes in the system. One of the nodes having the data is designated as *primary*, and the rest are designated as *backups*. The nodes are logically organized in a linear fashion, with the primary as the first node. All requests for operations on the data are sent to the primary site. If a request is sent to a backup site, the backup forwards the request to the primary.

If the operation requested is a read, then the primary site simply performs the operation and returns the results to the requesting process. If the operation is an update, then *before* performing the update, the primary site sends the update request to at least k of its backups. When all these backups have received the request, then the primary performs the operation, and returns the results, if any. All the backups perform the update operations they receive from the primary site. Since all requests arrive first at the primary site which forwards the update requests to other backups, all backups also get the requests in the same order as the primary site (since the communication channels are assumed to be reliable and order-preserving). Hence, on performing these requests, the data at the backups will be in the same state as the data on the primary. An alternative method to reduce the computation to be performed by the backups is for the primary to periodically checkpoint the state of the data object on the backups. In this case, the backups only need to perform the operations that were performed by the primary after the checkpoint. These are typically performed only when a backup becomes the primary.

Now consider what happens if some sites fail. If more than k sites fail nearly simultaneously, then nothing can be done, as the degree of replication is only $k+1$. If the total number of failures is less than k, then this scheme can mask the failures. Suppose that the failed sites are backup nodes. In this case, the user service is not disrupted, as the user gets its responses from the primary site.

If the primary site fails, then a new primary has to be elected. There are various ways for electing a primary (the reader is referred to [Gar82] for a discussion on election algorithms). We describe one simple approach. Since the nodes are ordered in a linear chain, the highest node in this chain that is alive becomes the new primary. The new primary is now responsible for performing the user operations. If the primary site records each operation on the backups, then the new primary site can start performing user requests after it has processed all the operations forwarded to it by the previous primary. If checkpoints are used, the backup starts executing from the last checkpoint and first performs all the operations performed by the previous

primary since the last checkpoint was established.

The one-copy serializability criterion is clearly satisfied when there are no failures, since all requests go to the single site — the primary site. Since all backups also get the update requests in the same order as the primary site, they are in the same state as the failed primary after performing the requests. Hence, when a backup takes over as primary, again the overall one-copy serializability criterion is preserved.

Now let us discuss failures that cause network partitioning. In this case, clearly only the partition that has the primary site can function, the other partitions cannot. That is, requests originating in the partition having the primary site can be serviced by the primary site. Requests originating in the other partitions cannot be forwarded to the primary site, and hence cannot be serviced. Hence, two different approaches are used for masking these two types of failures. In case of node failures which do not cause partitioning, the failure of a primary site is handled by one of the backups taking over the role of the primary. In the case of partitioning, no such action is done and the partition that has the primary site continues to function.

Since the primary site method employs two different approaches to handle node failures and network partitions, it can only work if network partitions can be distinguished from node failures. That is, the approach can work only if a node that is unable to communicate with the primary site can determine if this is due to site failures or network partitions. In the first case, where it knows that the primary site has failed, election is held to select a new primary. In the second case, the node simply waits until the partitions merge and it is able to communicate again with the primary.

7.2.2 Resilient Objects Using the Primary Site Approach

Now we describe an implementation of the primary site approach for supporting resilient objects [BJRA85]. Objects here are not simple data objects that are read or written, but are treated as an instance of abstract data types, as described earlier in Chapter 2. These types of objects give rise to nested actions, and the implementation has to handle that correctly. This method aims to support *k-resilient* objects. It also handles only node failures, and uses checkpointing to transfer state information from the primary site to the backups.

The objects are deterministic and if an operation is invoked on different copies of an object that are in the same state, then each object will execute the same sequence of steps and will reach the same state after the operation is completed. An operation on an object may be a *top-level operation* or a *nested operation*. A top-level operation is one that is requested by a user process. An operation on an object may invoke operations on other objects, giving rise to nested operations.

Each operation on an object is a sequence of steps $A = \{a_1, a_2, ..., a_n\}$, where each step a_i is a primitive operation, which could act on the data object or be a call to an operation on another object. If a step is a call to another operation, this gives rise to nested actions, and the step is called an *external step*. The *index* of an external step a_i is i.

Since an operation on the object is not a primitive operation, but consists of a sequence of primitive steps, nodes may fail during the execution of operations. Besides ensuring that the data object is accessible even if up to k sites fail, we also want the data to be resilient so that the ongoing operations are not disrupted by failures. Hence, another goal of supporting resilient objects is *forward progress*, which means that even if up to k nodes fail, the executing operations will be successfully completed.

Each object is replicated on at least $k + 1$ sites, one of which is the primary site, to which operations on the object are sent. The primary site periodically *checkpoints* the state of the object on the backups. During checkpoints, enough information is transmitted by the primary site to its backup so that any backup is capable of completing the ongoing operation from the checkpoint.

Checkpointing is not sufficient for ensuring forward progress. If the last checkpoint was established after the step a_i of an operation A by the primary site before it failed, then the backup which becomes the primary will start executing A from that state. A problem arises if the primary site had performed an external step since its last checkpoint. The new primary site does not know about this (since the checkpoint does not reflect the external call), and hence will invoke the external operation again. That is, for an operation A, the call to the external operation is made twice due to failure, while in normal processing, only one such call is made. This can clearly lead to an inconsistent state. Hence, the new primary site should not execute such external calls again. At the same time, the new primary site has to get the results of the external call in order to complete the processing starting from the checkpoint.

This is achieved as follows. Each operation A is assigned a unique operation-id (a timestamp-based scheme can generate unique-ids). A primary site makes this operation-id known to the backups before it starts executing A. Each step is assigned a unique step-id which is the operation-id concatenated with the index of the step. If there is an external call during the operation A, then the step-id of the step responsible for the call becomes the operation-id for that operation. The same method is followed in further nested calls, if any. At the end of the external call, the primary site transmits the results of the step to all the backups *before* it returns the result to the caller of the operation. Each backup keeps the results of these calls indexed with the step-id of the call. These are called *retained results*: when an external call is made, then its results are retained by the backups. This is a form of limited checkpointing by the

primary site.

Retained results can solve the problem described above. When an execution of an operation needs to make an external call, the node (the current primary site) executing the operation first checks to see if there are any retained results for this step-id. If so, it simply returns the result of the operation and does not make the external call. Note that since each operation is deterministic, starting execution from a checkpoint, a new primary site will assign the same step-id to the external call, since the index of the step will be the same as in the original primary site. This scheme makes sure that an external call is executed exactly once on the object, and no results are lost due to failures. The retained results can be deleted once the parent operation responsible for the nested call terminates.

To ensure one-copy serializability, proper concurrency control is also needed. With the primary site approach, since all requests are routed to the primary site, any centralized concurrency control protocol can be used. In the method described above, two-phase locking was used. However, this is not sufficient to handle failures. For masking a failure completely, the new primary site must also have all the information about the locks, otherwise it may resolve concurrent requests in a manner that may never occur with a centralized database. This can be done simply by the primary site distributing the information about locks to its backups. This can be done at each lock operation, or by piggybacking the information on other message exchanges to reduce the communication cost.

7.3 Resiliency with Active Replicas

In the primary site approach for supporting resiliency, only the primary site is really active and services user requests. The backups are largely passive replicas which record operations forwarded to them by the primary. By keeping only one copy active, supporting the one-copy serializability property is simplified. In this section, we will discuss another approach for supporting resiliency where all the replicas are simultaneously active. Since all replicas are active, other conditions have to be satisfied to ensure that the one-copy serializability property is satisfied and the different copies remain mutually consistent. As we will see, one way to satisfy the condition is by atomic broadcast. Though we are discussing this approach for supporting data resiliency, like the primary site approach, the method of active replication is very general and has also been used in other contexts to support resiliency. This approach is the most suitable for supporting resiliency against node failures only. Hence, we assume for this section that only node failures occur, and the communication network stays connected.

7.3.1 State Machine Approach

The method of using active replicas is also called the *state machine approach* [Sch90]. This approach views the system as consisting of servers and clients. In the case of data replication, the nodes having the copy of the data will be servers, while the nodes requesting operations on the data will be clients. We assume that the clients and servers are separate nodes. The state machine approach is a general approach for supporting fault tolerance by replication. It can handle Byzantine failures as well as fail stop failures. Our discussion is limited to fail stop failures only and is based on [Sch90].

In the state machine approach, resiliency against failures is supported by replicating the servers on different nodes. For *k-resiliency* (i.e., a system that can mask up to k node failures), the data is replicated on at least $k + 1$ nodes. Instead of sending a request to a designated node, a request is sent to all replicas. All replicas are equivalent and perform the request, and any can send the reply for the operation. Since any replica is allowed to service any request, to preserve one-copy serializability, it is essential that all replicas service a request in the same state so that all will provide the same result.

If each replica is initially in the same state, and gets the same set of requests in the same order, then each will produce the same output for an operation (as objects are deterministic). Hence, the key for supporting resiliency is to ensure that all replicas get the same sequence of requests. This, in turn, requires *agreement* and *order* properties be satisfied [Sch90]. Agreement states that all nonfaulty replicas receive every request, and order states that every nonfaulty replica process the requests in the same order.

For fail stop processors, the agreement requirement can be relaxed. [Sch90]. The requirement of order can be relaxed. If a request r is such that its processing does not modify the state of the state machine, then that request needs to be sent to only one nonfaulty state machine. This is so because the response from a nonfaulty state machine is guaranteed to be correct and the same as the response from other nonfaulty state machines. (Note that this does not hold if Byzantine failures are possible, since in that case, the response of a state machine cannot be trusted.)

The order requirement can be relaxed for requests that commute [Sch90]. Two requests r and r' commute for a state machine if the outputs produced by the state machine and the final state of the state machine by processing r before r' are the same as would result from processing r' before r. Clearly, if the result of executing two requests is the same, regardless of the order in which they are executed, then the order in which they are received at different state machines is not important. We will consider the general case only where we do not make any assumptions about

the commutativity of operations.

Agreement can be satisfied by a Byzantine agreement protocol, if the failures are Byzantine, or by a reliable broadcast protocol, if nodes are fail stop. Since we assume that the nodes are fail stop, agreement can be satisfied by reliably broadcasting each request to all the replicas. We have seen protocols for reliable broadcast earlier in Chapter 4.

The order requirement can be satisfied by assigning unique identifiers to requests and having state machine replicas process the requests according to the total order relation on the identifiers of the requests. This means that at any given time, a state machine should process a request that not only has the smallest identifier, but should not accept a future request that may have a smaller identifier. A request at a state machine replica is considered to be *stable* if no request can come later to the state machine from any client which has a smaller identifier [Sch90]. This means that the order requirement can be satisfied if each state machine replica processes the stable request with the smallest identifier. Since each request is reliably broadcast to each replica, if each replica follows this rule for servicing a request, then all replicas will service the requests in the same order. This reduces the problem of satisfying the order requirement to determining the stability of a request.

Stability can be determined if requests are assigned unique identifiers using the logical clocks discussed earlier in Chapter 2 and by using the following approach [Sch90]. If these clocks are used for assigning identifiers to requests, then we know that the identifiers will be consistent with the "causal" relationship. With logical clocks, a request r is stable at a state machine if the state machine has received a request with a *larger identifier* than r from *every* nonfaulty client in the system. Since the replica has received requests with a larger identifier from all clients, the property of logical clocks ensures that no request can be received later from a client with a request with a smaller identifier than what has already been received by the state machine. This implies that the state machine will never receive a request from any node which will have a smaller identifier than that of r. Hence, the request r is stable at the state machine.

There are various other ways to determine stability for satisfying the property [Sch90]. One approach to satisfy both the agreement and order is to use atomic broadcast. This approach will be used in the example discussed later in the section.

7.3.2 Resilient Objects Using Atomic Broadcasts

We now consider the object-action model for supporting resiliency which uses the state machine approach discussed above. With objects, we can have nested operations, since an operation on an object may invoke other operations. We have

already seen, while discussing the implementation of resilient objects using the primary site approach, that this introduces new problems. Here we will discuss one method that supports resilient objects by using active replicas [Jal89]. As with the state machine approach, we assume that the sites can fail, but that the network does not get partitioned.

Each data object is replicated on all the nodes in the system. Any process can request operations on the replicated objects. A node employs messages to request operations (on behalf of processes or operations on processes) and to return the results of an operation. A request for an operation is sent to all the nodes, and the operation is performed on all the replicas of the object. The requests for operations are broadcast to all the nodes by using an atomic broadcast protocol. We have discussed various methods to atomically broadcast a message earlier in Chapter 4. The use of atomic broadcast protocol ensures that each operational node gets every request, and that the different requests are delivered at different nodes in the same order. This ensures the mutual consistency of different replicas.

If an operation is requested on an object O_1 and that operation performs an operation on another object O_2, then all the nodes performing the operation on O_1 will request the operation on O_2. These are *images* of the same independent request. For consistency of data objects, only one of these requests should be serviced. For this, a numbering scheme is used. Each operation on an object is assigned an id-number. The id-numbers should be such that each independent request has a unique id-number, and all images of an independent request have the same id-number. If these conditions are satisfied, the images of a request can be identified by comparing id-numbers. A top-level request is always an independent request, with only one image.

Assume that each node is assigned a unique node number and has a counter that is incremented whenever the node receives or broadcasts a message. For a top-level operation request at a node, the id-number of the request is the node number of that node followed by the value of the counter. For a nested operation, the id-number of the request is the id-number of the parent request followed by a sequence number, where the sequence number is one more than the number of nested operations requested by the parent operation before requesting this nested operation.

A message to request an operation on an object carries the id-number of the request. Each node maintains two queues. For a node i, the queue req_i is the queue where requests are kept, and res_i is the queue which stores the results of the operations. A request (result) is broadcast by a node only if a copy of that request (result) does not exist in the respective queue, that is, if there is no request (result) in $req_i (req_i)$ with the same id-number as the incoming request (result). Due to this checking, a request cannot be deleted from the queue immediately after it has been

serviced. It has to be kept until no other copy of the request can arrive. For this, when a request is serviced or a result is returned to the caller, it is not deleted from the queue, but is merely *marked*. A request or a result can be deleted only after a "sufficient" time has elapsed (which depends on the communication delays and the relative processing speeds of nodes). The actions to be performed by a node i are shown in Fig. 7.2.

```
*[
    request (r)  →
        if r ∉ req_i then broadcast(r)

    □ receive (m)  →
        if m is a request for an operation then
            if m ∉ req_i then add (req_i, m )
        if m is a result then
            if m ∉ res_i then add (res_i, m )

    □ not empty (req_i) →
        r = first unmarked request from req_i
        a = result of performing the operation r
        mark (req_i, r)
        if (i is a requester for the operation r) then return (a)
        if a ∉ res_i then
            add(res_i, a)
            broadcast(a)
]
```

Figure 7.2: Actions of a node i for supporting resilient objects

When a request is generated at a node, it is broadcast only if the same request does not already exist in its request queue. If it exists in the queue, it means that this is a nested operation request and some other node has already broadcast it. When a request is received, it is added to the request queue if the same request does not already exist in the queue. Similar action is taken when a result is received.

If there are requests in the queue, the first unserviced (i.e., unmarked) request is considered. The operation is performed and the request is marked. Marking a request in the request queue implies that the operation has been performed on the local replica. If the node is also a requester for this operation, the result is returned

(to the requesting process). For a nested operation, all nodes are requesters; for a top-level operation, only one node is the requester. If a result with the same id-number already exists in the result queue, the result is marked; otherwise the result is added to the queue, marked, and broadcast.

It might appear that only one message for each request or result will be broadcast. However, in some cases, more than one message may be transmitted for a request (result). This happens because of the delay in delivering a broadcast message. Another node may broadcast the request (result) before it gets the message from some other node. Hence, the scheme for supporting resiliency has to handle duplicate requests.

It is clear that as long as at least one node is alive, operations can be performed on the data objects. The use of atomic broadcast for requests ensures that all active copies perform the same sequence of operations and are always mutually consistent and up-to-date. Any object is capable of servicing a request, and an ongoing operation is successfully completed even if nodes fail during its execution.

So far, we have focused on an operation on an object. However, a transaction or a user action may consist of many operations on different objects and proper concurrency control measures have to be applied to ensure one-copy serializability. With active replicas, any of the concurrency control methods can be used. The basic requirement is that all replicas (or state machines) use the same method for concurrency control that resolves conflicting requests in the same manner. If this is ensured, since the same requests arrive at each replica in the same order, each replica will face the same conflicts and will resolve it in the same manner, thereby preserving mutual consistency.

One method is to use the two-phase locking protocol. For each object, two more operations are defined: *lock* and *unlock*. When a user action makes a request for an operation on an object, it first performs a lock operation on that object. If the result of the lock operation is the granting of the proper lock, the request for the operation is made. When no more requests need to be made, all the objects locked during the operation are unlocked by the unlock operation.

Another issue with active replicas is *reintegration of failed nodes*. We have so far implicitly assumed that when a node fails, it remains failed, and we have shown that the remaining nodes in the system can satisfy any request. However, in reality, when a node fails, it recovers after it is repaired. Clearly, a repaired node cannot be reintegrated into the system directly, since its state is now out of date (i.e., the failed and recovered state machine has an old state). First, the recovered node will have to update its state before servicing any request. For proper reintegration, the repaired node will have to update its request based on the state of other nodes in the system.

One way to support reintegration is to define an operation called *state (O)* for an

object O, which returns the state of the object O. Suppose a node n becomes alive after having failed. Then n first requests the operation *state (O)* for each object O that resides on the node to update its copy of the object. It starts servicing requests only after it receives the result of the state (O) operation. Suppose the node n' services this state (O) request of n. It is possible that n may miss requests that arrive in the system after n' prime sends the result of state (O) but before n actually starts receiving requests. To avoid this after sending state (O), for some time the node n' also forwards to n the requests for operations that it gets.

7.4 Voting

In this section, we will discuss another pessimistic approach for replica control that employs voting. By *voting* we mean that performing an operation on replicated data is decided collectively by replicas through voting. A voting algorithm ensures that conflicting operations are not performed concurrently. A major advantage of many of the voting algorithms is that they can mask both node and communication failures, and do not require that a node distinguish between the two types of failures.

Voting-based methods have become extremely popular, and in recent times a large number of voting algorithms have been proposed. Voting schemes can be broadly considered as belonging to two categories: *static methods* and *dynamic methods*. In static approaches, the vote assignment and quorum requirements do not change, while in dynamic methods, vote assignment, total number of copies, or other information about the system may change with time in an attempt to adapt to the changing system state (in terms of failures or recoveries). We will discuss protocols of both of these in this section.

7.4.1 Static Voting Methods

Weighted Voting

The first voting approach was proposed by Thomas in [Tho79], which proposed a restricted form of voting. The concept was later generalized to *weighted voting* [Gif79]. Here we discuss the general weighted voting method.

In weighted voting, each replica of the node is assigned some number of votes. Any node that wants to perform a read operation on the data must first acquire at least r votes from the nodes in the system before it can actually read the data. Similarly, a node must first acquire at least w votes before it can write the data. The r and the w are called the *read quorum* and the *write quorum*, respectively. Let the total votes (i.e., the sum of votes of each of the replicas) be v. The quorums must satisfy two conditions:

1. $r + w > v$
2. $w > v/2$

The first condition guarantees that every read and write quorum intersect. This not only ensures that a read and a write operation cannot be performed concurrently, it also ensures that every read quorum has a replica that contains the latest copy of the data in which the latest update is reflected. The second condition ensures that two write quorums intersect, which prevents write-write conflicts, and if the system is partitioned into two groups, it allows a write to be performed in, at most, one group. Note that two read quorums need not intersect, as there is no read-read conflict.

The read and write operations work as follows. Each replica of the data has a *version number* associated with it, which is initialized to 0. When a transaction performs a read or a write operation on the replicated data, it first broadcasts a request for votes to all the nodes. All nodes that receive this request reply to the sender with the version number of their replica and the number of votes they possess. The requester node collects these votes until it has received replies from a group of nodes whose votes are equal to or more than required for the quorum. After acquiring the quorum, the node can perform the operation.

For a read operation, the node checks the version number of all the replicas in the read quorum that it has collected. Since each read and write quorum intersect, at least one of these replicas will be the latest, and will have the highest version number. Hence, the requester node reads the data from one of the nodes in the quorum that has the highest version number.

For the read to work, it is essential that after a write operation, all nodes in the write quorum have the latest copy of the data. In a write operation, the requester node makes sure that all the nodes in the quorum are written using the latest value. This may require reading values from some quorum members and writing them onto others before performing the operation. After the write operation, all the replicas in the write quorum will have the latest copy of the data.

The weighted voting approach can handle both site and communication failures, without requiring a node to distinguish between them. If a requesting node is unable to collect the quorum, then it cannot perform the operation, regardless of whether it was unable to get the quorum due to node failures or due to network partitioning. Under only site failures, read and write operations can be performed if the set of nodes that are alive (and connected) have a total of votes that is greater than w. If the number of votes is less than w, but more than r, then only read operations can be performed. If they are less than r, then even read operations cannot be done.

Suppose the system partitions into two groups of nodes. Since $w > v/2$, at most, one group can have the write quorum. Hence, updates can be performed in, at

most, one partition, thereby disallowing situations where inconsistencies can result. In the partition that has w votes, read operations can also be performed (as $r < w$). However, the other partition will not even have a read quorum, since $r + w > v$. This makes sure that only those read operations are allowed that can access the latest copy of the data. In general, if the system partitions into multiple groups, the following scenarios are possible:

1. One group has a read and a write quorum, and all other groups have neither read nor write quorums. In this case, all activity is allowed in one group, while no operations can be performed in others.

2. Some groups have a read quorum, but no group has a write quorum. In this situation, read operations can be performed in many groups, but updates are performed in none.

3. No group has even a read quorum. In this, no operation can be performed in any group. This may happen if the groups are very small.

It is clear that weighted voting preserves consistency by disallowing situations where inconsistencies can occur. The performance of voting clearly depends on factors like vote assignment, read and write quorums, ratio of read and write operations, etc. We will discuss some of these issues later in the chapter.

In a homogeneous system, frequently it is assumed that each node has one vote. There are two extremes for quorum selection in this case. The first is *read one and write all*, in which r is 1, and so w equals v (to satisfy $r + w > v$). In this case, for reading, any node can be read, but an update must be made on all the nodes in the system. If even one node has failed, update operations cannot be performed. If an update is allowed, then when the failed node recovers, it may perform a read operation (as $r = 1$), thereby giving outdated data to the requester. The second is *majority voting*, where both r and w equal the majority of total votes (i.e., $\lceil v/2 \rceil$). In this case, for both read and write operations, a majority of the nodes must take part. If a network partitions, the majority group functions (if one exists), while the others do not.

Example. Consider a system with six nodes A, B, C, D, E, and F, each with one vote. Let w be 4 and r be 3; these satisfy the constraints on vote assignment. If the node B fails but the rest of the nodes remain connected, then operations will be performed in the rest of the nodes, as each node will be able to get a quorum of 4. If, due to communication failure, the network partitions into two groups $\{A, B, C, D\}$ and $\{E, F\}$, then in the first group, both read and update operations can be performed, while in the second group, no operation is allowed. If the communication failure

partitions the set of nodes into the groups $\{A, B, C\}$ and $\{D, E, F\}$, then the read operation will be permitted in both, but no update operation will be permitted in either partition. If nodes C and D fail and partition the remaining nodes into groups $\{A, B\}$ and $\{E, F\}$, then no operation of any kind will be permitted in either group.

Hierarchical Voting

A major problem with majority voting is that the number of nodes required in a quorum for performing an operation increases linearly with the number of replicas. Many schemes have been proposed recently which arrange the nodes in a logical structure and define the quorum based on that in order to reduce the communication cost [AA89, Kum91a, LG90, CAA90a]. We will now discuss one such scheme, called *hierarchical voting* [Kum91a]. The hierarchical method reduces the number of nodes that must be in a quorum by introducing a multiple-level algorithm that involves multiple rounds of voting.

In the hierarchical approach, the set of nodes is logically organized as a multiple-level tree. The root level is level 0, and the leaves exist at level m, where m is the depth of the tree. The physical copies of the object are stored only at the leaves of the tree, or at level m. The higher-level nodes of the tree correspond to logical groups. We assume that all nodes at a level have the same number of children. The number of nodes (each representing a logical group) at level 1 is represented by l_1. The number of subgroups of a node at level i is represented by l_{i+1}. Note that l_{i+1} is not the number of nodes at level $i + 1$, but the number of children of a node at level i. The total number of nodes at level $i + 1$ will be $l_1 * l_2 * ... * l_i * l_{i+1}$.

A quorum is associated with each level. A read (write) quorum at a level i is defined as the number of subgroups of a node in the level $i - 1$ quorum that must be included in the quorum to obtain a read (write) access to the group. That is how many of the l_i nodes must be included in the quorum for each level $i - 1$ node that is included in the level $i - 1$ quorum. The read (write) quorum at level i is denoted by r_i (w_i). Note that the definition is recursive, and hence the process of acquiring a quorum at a level will be recursive. A quorum at level 1 implies quorum collection at all levels right down to level m. Hence a quorum consensus algorithm just needs to specify the quorum requirements at level 1.

Suppose that a quorum consensus algorithm requires a read quorum of r_1 at level 1, and a write quorum of w_1 at level 1. It has been shown that this quorum consensus algorithm is correct (i.e., it will provide write-write and read-write mutual exclusion) if for all levels $i = 1, 2, ..., m$,

1. $r_i + w_i > l_i$, and
2. $2w_i > l_i$

The reason for the correctness is as follows. At level 1, the conditions ensure that any read and write quorums will have at least one node in common at level 1. At level 2, when a quorum is collected in the subgroups of this common node, again the conditions will ensure that there is at least one node in common. This will continue all the way down to level m, and at least one physical copy (i.e., leaf node) will be common to the two quorums. Hence, once the quorums at each level are fixed (satisfying the conditions given above), then a read operation must collect a quorum of $r_1(w_1)$ of level 1 nodes. The recursive definition of a quorum, then, ensures the consistency. An algorithm for collecting the quorum is given in [Kum91a].

It is clear that with this approach, for a read operation, for each node in the quorum at level 1, r_2 nodes will be needed in the quorum, and for each node at level 2, r_3 nodes will be needed in the quorum, and so on. Hence, the total number of physical copies in a read quorum is $r_1 * r_2 * ... * r_m$, and the total number of physical copies in a write quorum is $w_1 * w_2 * ... * w_m$.

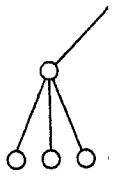
If each l_i is kept as 3 (an algorithm is given in [Kum91a] to organize a given set of nodes into a multiple-level tree with l_i as 3), then the depth of the tree is $log_3(n)$, where n is the total number of replicas of an object. The total number of physical copies that are finally read are $2^{log_3(n)}$, which is equivalent to $n^{0.63}$. That is, for an operation with a hierarchical voting method, only $n^{0.63}$ replicas need to be read, whereas in majority voting, $\lceil (n+1)/2 \rceil$ number of replicas will be read. Clearly, for higher values of n, there will be a reduction in the number of copies to be read. However, the cost for this is that the quorum collection process requires $log_3(n)$ rounds, whereas it requires only one round in majority voting.

Example. Consider a collection of 27 replicas organized in a three-level hierarchy, as shown in Fig. 7.3 [Kum91a]; in this, l_1, l_2 and l_3 are 3 each. There are various read and write quorums possible that will satisfy the constraint at each level that $r_i + w_i > 3$. For different quorums, a different number of leaf nodes are eventually read. Some of the different possible combinations of quorums and the number of copies that are read or written are shown in Table 7.1 [Kum91a] (in

No.	r_1	w_1	r_2	w_2	r_3	w_3	R	W
1.	1	3	1	3	1	3	1	27
2.	1	3	1	3	2	2	2	18
3.	1	3	2	2	2	2	4	12
4.	2	2	2	2	2	2	8	8

Table 7.1: Possible quorums

the table, the last two columns, R and W, represent the total number of copies read



or written). I
quorum can b
the write quo
Besides l
approach pro
The propose
nodes. The c
a quorum, a r
If some of th
nodes. Henc
$O(N)$ to $O($
The tech
incurred in r
$|F|$, which is
$|F|/m$. The
that any of th
and write qu
The quorum
and maximu
these, are sp
The *mu*
weighted v
assignment
and write q

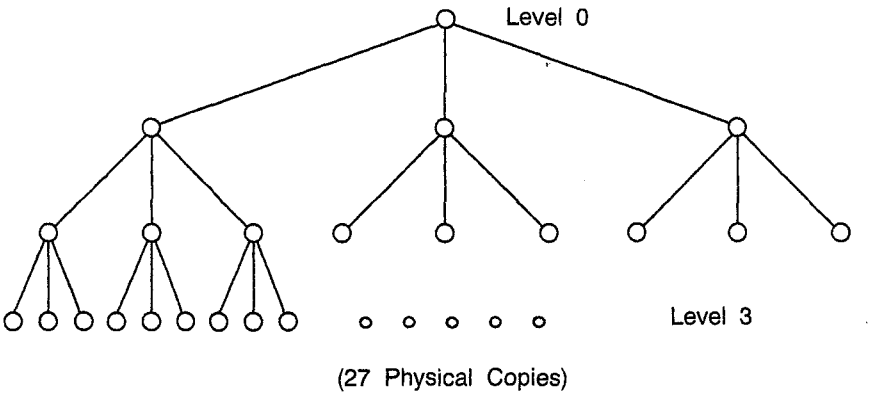


Figure 7.3: Hierarchical voting example

or written). Note that if the read and write quorums are set at 2 each, then a write quorum can be collected with as few as 8 copies. With the weighted voting technique, the write quorum is at least 14. Also, note that R and W are often less than 27.

Besides hierarchical voting, other variations of voting have been proposed. The approach proposed in [AJ92] aims to reduce the communication overhead of voting. The proposed method divides the set of nodes into logical groups of intersecting nodes. The cardinality of each group is $\sqrt{2N}$ (for a N node system). For collecting a quorum, a node first needs to communicate only with the members of its own group. If some of these nodes have failed, the node may need to communicate with other nodes. Hence, the cost of communication, when no failures occur, is reduced from $O(N)$ to $O(\sqrt{N})$.

The technique presented in [JA92] aims to reduce the storage overhead that is incurred in replicating the data on all the nodes. In this approach, a file of the size $|F|$, which is to be replicated, is encoded and then broken into n parts, each of the size $|F|/m$. The n parts are then stored on different nodes. The coding scheme is such that any of the m parts of the file are sufficient to reconstruct the entire file. The read and write quorums have to be redefined when a file is coded and split in this manner. The quorum requirement is not straightforward and the minimum sufficient quorum and maximum necessary quorums, as well as the read and the update algorithms with these, are specified in [JA92].

The *multi-dimensional voting* (MD voting) approach is a generalization of weighted voting which offers more flexibility [CAA90b]. In MD voting, the vote assignment to each node is a k-dimensional vector of non-negative integers. The read and write quorum requirements are also k-dimensional vectors. Each dimension of

the vote and quorum assignment is similar to regular voting and can be combined in many ways, making MD voting more powerful and flexible. In addition, a number p is defined $1 \leq p \leq k$, which specifies the number of dimensions for which a quorum must be satisfied. Hence MD voting requires that in a dimension, the number of votes required must be greater than or equal to the quorum requirement for that dimension, and this should be satisfied for p different dimensions. A MD voting algorithm can therefore be characterized by the number of dimensions it has and the number of dimensions in which a quorum is needed, and is represented by $MD(p, k)$. A key issue in MD voting is how to select the quorum values (it is not always the majority) and the value of p. Methods for selecting these are discussed in [CAA90b].

7.4.2 Dynamically Adaptive Methods

By nature, the static voting methods do not adapt to changes in the system due to failures. For example, with weighted voting, if due to repeated partitioning, the system breaks into small groups, no group will be allowed to perform (update) operations. The reason for this is that weighted voting always requires that the number of sites necessary for performing an operation is the majority (if majority voting is being used) of the total number of sites in the system. So, if a system partitions with one group as the majority group, and if this group further partitions, then no group may have the majority. In this section, we will study some approaches that generalize the weighted voting strategy to avoid this problem of repeated partitioning. These approaches adapt the voting strategy to changes in the system by changing the voting parameters.

Dynamic Voting

Dynamic voting is different than the majority voting scheme in that for an update it requires a majority of the copies that are accessible at the time of the update. By this, it solves the problem caused by repeated partitioning. There are two slightly different approaches that have been proposed for dynamic voting [JM87, Dav89]. The discussion here is based on the method proposed in [JM87], which considers a system in which each node is assigned one vote.

For each copy of the data d_i at node i there is a *version number* VN_i (initialized to zero), which counts the number of successful updates to d_i. The *current version number* of the data d is the maximum taken over the version number of all replicas of d. A replica d_i is said to be *current* if its version number is the same as the current version number of d. A group is said to be a *majority partition* if it contains a majority of the current copies of d.

With each copy d_i is also associated another integer called the *update sites*

cardinality, SC_i, which reflects the number of sites that participated in the most recent update to d_i. Initially, SC_i is set to the total number of sites in the system. Whenever a copy d_i is updated, then SC_i is set to equal the total number of copies of d which were updated during this update.

With dynamic voting, a site can perform an update if it belongs to a majority partition. Therefore, a site has to first determine if it belongs to a majority partition. Consider the case where site 1 wants to perform an update operation. It requests all the nodes it can communicate with to send their version numbers and update sites cardinality. Let the nodes which respond to the request of node 1 be node 2, ..., m. The node determines the maximum version number from all the responses it gets (and its own version number). This version number may be the current version number. It also determines the maximum update sites cardinality from all the responses. If the number of nodes which have the maximum version number is less than half of the maximum update sites cardinality, this set does not form a majority and the operation is rejected. Otherwise, the set of nodes having the maximum version number form a majority and the update is performed on all of them. The version number and the update sites cardinality of the nodes is also updated. The actions performed for an update are shown in Fig. 7.4.

$M = max\{VN_i : 1 \leq i \leq m\}$ (an integer)
$I = \{i : VN_i = M, 1 \leq i \leq m\}$ (a set)
$N = max\{SC_i : i \in I\}$ (an integer)

if $|I| \leq N/2$ **then** reject the operation
else

for all sites in I **do**
perform the update
$VN_i = M + 1$
$SC_i = |I|$

Why not for all sites in the partition?

Figure 7.4: Performing an update with dynamic voting

Clearly, dynamic voting will permit operations to be performed in groups that do not form a majority of the total nodes. Once it allows operations in such groups, to preserve consistency, it has to also ensure that no further group can perform any operation without including any node from this group. That is, if there are groups that do not currently form a majority group and no operation is being performed in

them, rejoining of these groups may form a group which may even have a majority of the total nodes in the system. Even in this case, this group cannot be allowed to perform operations (or it cannot be allowed to form the majority group) until it reconnects with the group that has the current copies.

Suppose after the joining of groups, a site i realizes that it has a copy that is not current. First, it has to "catch up" and update its state. A node is allowed to update its state if, and only if, it belongs to a majority group. Suppose that node 1 can communicate with sites $2, \ldots, m$. The copy of the node 1 is not current if its version number is less than the version number of some site in the group $\{2, \ldots, m\}$, or if it has the highest version number and the set of nodes that have the highest version number do not form a majority partition. If node 1 realizes that its version number is not current, in order to "catch up" it has to perform some actions. First, it determines that the group it belongs to forms a majority group, since only in that situation it can update. If it is a part of a majority partition, it updates by requesting the missing updates. The actions for updating the state are shown in Fig. 7.5.

$M = max\{VN_i : 1 \leq i \leq m\}$
$I = \{A_j : VN_j = M, 1 \leq j \leq m\}$,
$N = max\{SC_k : k \in I\}$
if $|I| \leq N/2$ **then** the node i cannot update its state
else
    Get state from a node with a current copy
    Set VN_i to M
    Set SC_i to $N + 1$

Figure 7.5: Updating state dynamic voting

Example. Let us illustrate this approach by an example [JM87]. Suppose there are five nodes A, B, C, D, and E which have copies of a data d and which initially form one partition. After nine update operations on d, each of the five copies will have a version number of 9 (reflecting nine update operations), and an update sites cardinality of 5 (since there are five nodes in this partition). Now suppose that the communication network fails and partitions this group of nodes into two groups $\{A, B, C\}$, and $\{D, E\}$. If A gets another update operation, it will be able to communicate only with B and C. Since these three nodes form a majority partition, an update is performed on all three. After this operation, the state of the different copies is [JM87]:

	A	B	C	D	E
VN:	10	10	10	9	9
SC:	3	3	3	5	5

Since A can communicate only with B and C, after the operation, the version number of the majority group becomes 10, and the group's update sites cardinality becomes 3 (as only three nodes perform the update). The state of the other two nodes stay the same. Suppose that the majority partition further partitions, forming two groups $\{A, C\}$ and $\{B\}$, and A gets an update operation. With weighted voting, A cannot perform this operation, since $\{A, C\}$ together do not form a majority of the total number of nodes in the system. However, they do form a majority of the current copies (as there are three current copies). Hence, operations are allowed in this partition. After this operation, the state of the system will be [JM87]:

	A	C	B	D	E
VN:	11	11	10	9	9
SC:	2	2	3	5	5

Let us now consider the joining of groups. Suppose that the groups $\{B\}$ and $\{D, E\}$ join, forming a new group $\{B, D, E\}$. No operation should be allowed in this group, as no node in this group has the latest copy of the data, and allowing an operation will violate consistency (a read operation will be able to "see" old data). Let us see how dynamic voting handles this. Suppose that D wants to update a copy of the data by using dynamic voting. It will find M is 10, and the set I contains only B, and N is 5. Since the $|I|$ is not $N/2$, D is not allowed to update its state with others. If it were allowed, then after the update, B, D, E will have a version number of 10, and an update sites cardinality of 5, which will give $\{B, D, E\}$ a majority, thereby leading to an inconsistency. So, even though group $\{B\}$ has joined with $\{D, E\}$, each node is in the same state as it was before joining. Hence, for the purposes of performing operations, they are still treated as separate groups. If a request for an operation comes, say, at B, it cannot perform that operation, since the set I which it will form will contain only B, and N will be 3. So, the set I does not form a majority group and hence the operation cannot be performed. This preserves the mutual consistency of the data. It should be clear that if a group is formed by joining a group with the group that currently forms the majority group, the new group will also have a majority.

Dynamic Reassignment of Votes

Now let us discuss another dynamic technique by which the availability loss due to partitioning can be reduced. The approach of dynamic voting is to make the majority partition the set of total nodes in the system, so that it can handle further

partitions. A somewhat different approach is to change the votes of the node in the majority partition such that the loss of nodes is properly compensated and further partitioning can be handled. This is the approach of *dynamically reassigning votes* [BGS86].

The first approach for reassigning votes is the *overthrow technique*. In this technique, after a partition or a failure, for each node x outside the majority group, there will be one node in the majority group that supplants x, such that loss of x is not felt. Consider a system in which only one node x has been partitioned from the rest, and the votes of x are $v(x)$. Suppose that it is decided that the node a in the majority group is to supplant x. This decision can be made by simply ordering the nodes and selecting the highest node in the majority partition. The new votes of a have to be such that they cover the voting power of a before failure, plus the voting power of x, plus the increase in the majority caused by the increase in the total number of votes. If a increases its votes by $2v(x)$, the total number of votes will also increase by this amount, and the majority will increase by $v(x)$. Since the new votes of a provide the votes for x, as well as this increase in the majority, it can be shown that all the majority groups that used x can be formed by using a instead. The side effect of supplanting x is that node a becomes "more powerful," as it has more votes now and will be needed more often in forming majority groups.

The second method for dynamically reassigning votes is the *alliance technique*. In this, instead of supplanting a node x by one node, a group of nodes in the majority partition are used. If there are N nodes in the majority partition, a node x can be supplanted by assigning $\lceil (2v(x)/N) \rceil$ extra votes to each node. Since we can assign surplus votes, another way is to assign each node $2v(x)$ extra votes, or we can assign each node $v(x)$ votes, if the total number of nodes is more than two. Other distribution methods are also possible.

Now consider what happens when the node x rejoins the majority group. Clearly, something needs to be done, otherwise x will get "marginalized" and its votes will be reduced relative to the votes of other nodes. One way to avoid this is to have each node relinquish the extra votes it acquired because of x, after x rejoins. To implement this would require that each node keep track of how many votes it acquired when some node was excluded. Another method is to increase the vote of x when it rejoins. This method will continue to increase the vote assignment, and finally all nodes have to revert back to their original assignment to avoid making the size of the vote unmanageable.

Example. Consider a system with four nodes and initial vote assignment as $v(a) = 6$, $v(b) = v(c) = v(d) = 5$ [BGS86]. That is, the total votes are 21, and the majority is 11. Assume that node a gets disconnected from the rest, leaving $\{b, c, d\}$

as the majority group with 15 votes. Using the overthrow technique, assuming b is to get the extra votes, the new vote assignment will be:

$$v(a) = 6, v(b) = 17, v(c) = v(d) = 5.$$

The total votes are now 33, and the majority is 17. With the alliance technique, where each node gets $v(x)$ extra votes, the final vote assignment is:

$$v(a) = 6, v(b) = v(c) = v(d) = 11.$$

The total votes with this reassignment are also 39, and the majority is 20.

Suppose now that node c separates from the group $\{b, c, d\}$. If no vote reassignment was done, then the group $\{b, d\}$ will have a total of 10 votes, which is not a majority. However, with reassignment by the overthrow technique, this group will have a total of 17+5=22 votes, which forms the majority. In the reassignment by alliance technique, the votes of this group are 11+11=22, which also forms the majority. As we can see, in this situation, without reassignment, no group will have a majority, but with reassignment, one group does have a majority.

Besides dynamic voting and dynamically reassigning votes, many other methods have been proposed that try to adapt the voting system to changes in the system caused by failures. The *missing writes* approach tries to improve the read performance of weighted voting [ES83]. In weighted voting, if r is 1 (and w is equal to the number of nodes), read performance is good, but then updates are stopped as soon as one node fails. If r (and w) are a majority, then the read performance is poor. In this approach, the system works in two modes. In the normal mode for a read operation, only one copy is read (i.e., r is 1), and for a write operation, all copies are updated. If some copy cannot be accessed in a write operation, then the system runs in the failure mode. In the failure mode, the missing-writes approach works like majority voting; both read and write operations require a majority. By doing this, the performance of read operations is improved when the system has no failures. Again, this method works only if nodes fail or if node and communication failures can be distinguished.

The method of *voting with ghosts* tries to improve the write availability of weighted voting [RT88]. In this approach, if a node fails, a *ghost* is started within the same group as the failed node, and is assigned the same number of votes as the failed node. A ghost is a process without storage. Therefore it cannot reply to a read request, and so does not participate in a read quorum. Ghosts are allowed to participate in a write quorum. The write operation has an additional restriction in that it must contain at least one non-ghost copy. Since a ghost is not allowed to take part in a read quorum, the intersection of a read quorum with a write quorum (recall that with weighted voting, a read quorum must intersect with every write quorum) can

only contain non-ghost copies. This ensures that the read operation will get the latest data. When a node comes back up, it replaces the ghost only after it has obtained the latest data. This can be done by the ghost of the node acquiring a read quorum and then reading the latest copy and installing it on the recovered node.

Another technique is to use the *regeneration of objects* [PNP88]. In this method *configuration data* about the replicated data is kept in a directory, which is itself replicated. If a node fails, its effect is not felt in a read operation. However, if in a write operation the required quorum is not obtained, then new copies of the data are created on different nodes and the configuration data is updated. In this manner, the degree of replication is maintained and the change in location of replicas is reflected by properly maintaining the configuration data. The method of *available copies* [BG84], and *voting with witnesses* [Par86] are similar.

7.4.3 Vote Assignment

In a voting method, an update is performed on a group of nodes. The groups of nodes that can be formed for an operation are such that they always intersect with each other. This ensures that if an update is to be performed in a group, then no other group is performing updates. This also ensures that there is a write-write mutual exclusion. The strength of a voting algorithm lies in the fact that it is able to support mutual exclusion without communication between nodes. A majority voting strategy defines a set of groups of nodes, such that each group has a majority of votes.

The performance of a voting system, particularly when network partitioning occurs, will clearly depend on the assignment of votes to different nodes. Consider the example of a system consisting of four nodes: a, b, c, and d. If we assign one vote to each node, then the possible groups for performing an update are:

$$\{\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}.$$

That is, these are the groups whose votes add up to a majority of votes (i.e., at least three out of four nodes). Now consider an assignment that gives a two votes and the rest one vote each. The majority is still three votes, so the groups that have a majority are:

$$\{\{a, b\}, \{a, c\}, \{a, d\}, \{b, c, d\}\}.$$

This set of groups is clearly preferable to the previous grouping, as the system can continue to perform updates (at least in one group) in all the groups of the previous grouping, as well as some more. For example, if after a partition, the groups are $\{a, b\}$ and $\{c\}$, then the latter set of groups will allow updates to be performed in one group while the former will not allow any updates.

This example clearly shows that the reliability of a voting-based system depends critically on the vote assignment. If partitions occur, vote assignments will determine whether updates are allowed or not. We say that a voting system has *halted* if no operation can be performed on any node, that is, there is no group that has a majority of votes. A system is *up* otherwise. The goal of vote assignment is to maximize the steady-state probability that the system is up. Here we will discuss some approaches to assigning votes in a majority voting system.

Using Coteries

Here we discuss an approach to vote assignment by enumerating the possible groups that can be formed by majority voting [GB85]. By a vote assignment we get a set of groups (where each group is a set of nodes), such that each group has a majority of votes. In addition, the intersection of any two groups is non-null. Such a set of groups is called a *coterie* [GB85]. Formally, if U is the set of nodes that compose the system, then a set of groups S is a *coterie* if:

1. $G \in S$ implies that G is not empty and is a subset of U.
2. If $G, H \in S$, then G and H must have at least one node in common.
3. There are no $G, H \in S$ such that $G \subset H$.

The second condition is called the *intersection property*, and the last condition ensures *minimality*. For example, if $\{a, b\}$ is a group, then clearly $\{a, b, c\}$ will also form a group and is not included in the coterie.

A coterie R is said to *dominate* another coterie S, if for each group in S, there is a group in R that is its subset. If there is no coterie that dominates a coterie S, then S is *nondominated (ND)*. If R dominates S, then clearly R is preferable over S, and consequently we want to avoid considering it while deciding the vote assignment. In the example given above, the second coterie dominated the first one (which is why it was superior). Hence, for vote assignment purposes, we only need to consider ND coteries. For the remainder of the discussion, a coterie will mean ND coteries.

Now let us discuss vote assignments. Let U be the set of nodes in the system. A *vote assignment* is a function $v : U \rightarrow N$ (N is nonnegative integers) and $v(a)$ is the number of votes assigned to a node a. For a vote assignment v, the total votes $TOT(v)$ is the sum of votes for all the nodes in the system, and the majority $MAJ(v)$ is defined as $\frac{TOT(v)}{2} + 1$, if $TOT(v)$ is even, and as $\frac{TOT(v)+1}{2}$ if $TOT(v)$ is odd. The set of *majority groups* (i.e., groups, each having a majority of votes) of a vote assignment v is Z, where:

$$Z = \{G | G \subseteq U \text{ and } \sum_{a \in G} v(a) \geq MAJ(v)\}.$$

The minimal elements of Z (i.e., those groups having no subset in Z) form a coterie, and are called a *coterie corresponding to* v.

Now we can define what similar vote assignments are. Two vote assignments are *similar* if, and only if, their corresponding coteries are the same. Hence, in a system with an odd number of nodes, a vote assignment that gives 1 vote to each node is similar to another vote assignment that gives 3 votes to each node.

It has been shown that if the number of nodes in the system is five or less, then for all possible coteries there is a vote assignment. That is, for any coterie S, there is a vote assignment v whose corresponding coterie is S. Hence, one possible method for selecting the best vote assignment (for a system with five or fewer nodes) is to first enumerate all possible coteries. An algorithm for enumerating coteries for a system is given in [GB85]. Once the coteries are enumerated, select the coterie that yields the best reliability for the given system, that is, select the one in which the probability of the system being operational is the highest. Finally determine the vote assignment that corresponds to this coterie.

To determine the vote assignment that corresponds to a given coterie, the coterie can be converted into a set of linear equations. Since each group in the coterie must have a majority of votes, for each group G we get, $\sum_{a \in G} v(a) \geq MAJ(v)$. If a solution exists, then it can be determined by solving these inequalities. If no vote assignment exists that corresponds to a coterie, then the inequalities will lead to a contradiction.

If the number of nodes is more than five, then there exist coteries with no vote assignment corresponding to them (though there exists an MD voting that corresponds to it [CAA90b]). Hence, selecting a vote assignment by selecting a corresponding coterie will not be straightforward. Furthermore, the number of different possible vote assignments in a system with n nodes is 2^{n^2} [GB85], and the number of coteries generated is more than this according to the enumeration algorithm. Hence, for large systems, the approach of enumerating all the coteries for evaluation is of limited use for selecting vote assignments.

Heuristics for Vote Assignment

Another approach to selecting the vote assignments is not to try for the very best, but to use some heuristics that may suggest the best vote assignment or a vote assignment that is close to the best. Here we describe some heuristics that have been proposed [BG87]. A vote assignment is *uniform* if each node gets the same number of votes, and if the number of nodes is odd. This means that each node is assigned one vote. If the number of nodes is even, then one node has two votes, while the rest have one vote each. A *singleton assignment* is one in which all the votes are assigned

to a single node, that is, one node has one node and the remaining nodes have zero votes. A singleton assignment makes the system similar to a primary site system.

Heuristic I. For each node i, multiply its reliability by the sum of reliabilities of the links incident upon i. Round the number obtained, and this number represents the votes assigned to the node i. If the total number of votes is even, then the node with most votes gets one extra.

Heuristic II. For each incident link to node i, multiply its reliability by the reliability of its other end-point node. These products are summed and finally multiplied by the reliability of i to get the votes assigned to i. Rounding and making the total number of votes odd is done in the same manner as in Heuristic I.

Heuristic III. This heuristic attempts to find a cluster of nodes that is most likely to exist, and assign votes only to it. A link is considered *weak* if the product of its reliability and the reliability of one of its end points is less than 0.5. Weak links are first removed, which may partition the network. From this reduced graph, the biconnected components are determined. A biconnected component of a graph is a subgraph in which there are at least two paths from any node to any other node, that is, removal of one node cannot cause the subgraph to become disconnected. The biconnected components are assigned votes separately using Heuristic II. Out of these, the assignment that maximizes the probability that the system is up is taken as the vote assignment.

Example. Let us illustrate these heuristics by an example [BG87]. Consider the graph shown in Fig. 7.6(a), which also shows the reliabilities of the nodes and links. For this graph, Heuristic I will result in the assignment:

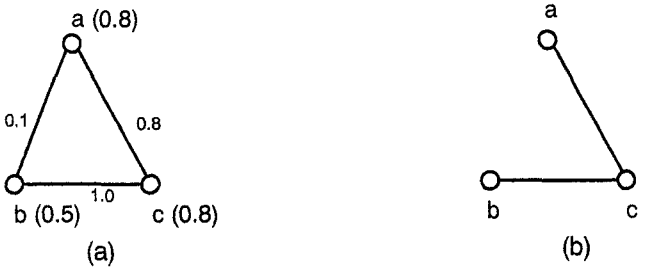


Figure 7.6: An example

v(a) = round(0.8(0.1+0.8)) = 1
v(b) = round(0.8(0.8+1.0)) = 1
v(c) = round(0.5(0.1+1.0)) = 1

Heuristic II will give:

$$v(a) = \text{round}(0.8(0.5*0.1+0.8*0.8)) = 0$$
$$v(b) = \text{round}(0.8(0.8*0.8+1.0*0.5)) = 1$$
$$v(c) = \text{round}(0.5(0.1*0.8+1.0*0.8)) = 0$$

Heuristic III will first produce the transformed graph, which is shown in Fig. 7.6(b). This graph has three biconnected components, each with a single node. The three assignments for these three biconnected components are therefore $< 1, 0, 0 >$, $< 0, 1, 0 >$, and $< 0, 0, 1 >$. The second and the third assignments will result in a system reliability of 0.8, and so any of these can be chosen.

As enumerating all possible vote assignments is difficult for a system consisting of six or more nodes, it is hard to evaluate the effectiveness of these heuristics for such systems. For a system with five or fewer nodes, best vote assignment can be determined by enumerating the coteries, and then determining the reliability of this assignment. In experiments, it has been found that for such systems, one of the heuristics provides the best possible reliability [BG87]. Also, for larger systems, in many cases the heuristics give a vote assignment which offers a reliability close to the best-known (through simulation) reliability.

The example above seems to suggest that singleton or uniform assignments may frequently offer the best reliability. It has been shown that for homogeneous systems where the reliability of nodes is the same and is greater than 0.5, if the links are perfectly reliable (i.e., have a reliability of 1.0), then uniform assignment is better than any nonuniform assignment [BG87]. It has also been shown that if the system has two nodes, and the links are perfect, then both uniform and singleton assignments yield the same reliability. And for a larger system with perfect links and a node reliability greater than 0.5 (but less than 1.0), in general, uniform assignment is better than singleton assignment. Another heuristic has been proposed in [Kum91b], which uses a randomized algorithm for vote assignment.

An Integer Programming Approach

Another approach for assigning votes is based on integer programming. For a network, first the top k groups are determined (perhaps using simulation), where k is a parameter and is chosen so that most of the likely groups are included. These groups represent the most likely partitions in case of failures. With these k groups, the vote assignment problem is formulated as an integer programming problem, with

the objective of minimizing the average number of operations that cannot be satisfied due to partitioning.

The criteria used for selecting the top k groups is the average number of operations that are performed by the system per unit of time in a group. For a group, the rate of arrival of requests is the sum of arrival rates at the nodes in the group. The average number of operations performed in a group per unit of time is the product of the rate of arrival of requests and the fraction of time (in the life of the system or a long period of time) for which this group exists. For integer programming, we have to select k groups that have the highest number of operations performed per unit of time. The simplest way of determining these groups is through simulation, using the steady-state probabilities for nodes/links to be up, as is done in [Ven92].

With the top k groups known, the goal is to assign votes in such a manner that these groups have a majority of votes. It is easy to observe that it may not always be possible to assign a majority of votes to all the groups. Hence, the aim is to assign votes in such a way that the groups getting a majority of votes have the maximum average operations performed in them. This can be formulated as an integer programming problem. Three different formulations have been proposed in [Ven92]. We will describe one formulation here. Let n be the number of nodes in the system, and T_i be the average number of operations performed per unit of time in a group G_i.

The problem formulation is such that we get a vote assignment which assigns a majority of votes to a subset of k groups for which the sum of T_i is the maximum. Let X_i be a binary variable which is zero if group G_i has a majority of votes, and is one otherwise. By defining X_i in this manner, we can formulate this as a minimization problem. The integer programming model is:
Minimize:

$$\sum_{i=1}^{k}(T_i * X_i)$$

subject to:

$$\frac{1}{2}(\sum_{j=1}^{N} v(j) + 1) - \sum_{k \in G_i} v(k) \leq d * X_i (i = 1, \ldots, k)$$

$$X_i = 0 \text{ or } 1 \ (i = 1, \ldots, k).$$

Since $X_i = 0$ represents the ith group getting the majority of votes, $T_i * X_i$ will represent the average number of operations lost due to the ith group not getting a majority of votes. The objective function is trying to minimize the sum of the average number of operations lost due to the groups not getting a majority of votes.

The first term in constraint (for a group G_i) is $1/2(\sum_{j=1}^{N} v(j+1)$ which represents the majority of votes. The second term $\sum_{k \in G_i} v(k)$ represents the votes assigned to group G_i.

Case 1. When X_i is zero, the constraint will become

$$\frac{1}{2}(\sum_{j=1}^{N} v(j) + 1) - \sum_{k \in G_i} v(k) \leq 0$$

which makes sure that the ith group will have votes greater than or equal to the majority of votes.

Case 2. When X_i is one, the constraint becomes

$$\frac{1}{2}(\sum_{j=1}^{N} V_j + 1) - \sum_{k \in G_i} V_k \leq d.$$

The value of d should be such that when X_i is one, whatever the votes assigned to group G_i are, the constraint is still satisfied. As the value on the left-hand side will be, at the most, the majority of votes (since votes are positive), the value of d is chosen so that it is greater than the possible majority of votes needed to satisfy the constraint. This also means that the constraint allows a majority of votes to be assigned to group G_i when X_i is one. We have to make sure that whenever X_i is one, the ith group will not have the majority of votes. Since the objective function is trying to minimize $T_i * X_i$, integer programming will always try to assign all X_i's as zero. According to Case 1, $X_i = 0$ means that G_i gets the majority of votes. If it is not possible to assign the majority of votes to all the groups, then some X_i's will have to take the value one. The objective function will select a subset out of the k groups so that the sum of T_i for these groups is the maximum, and it will assign the corresponding X_i as zero. The constraint will force these groups to get a majority of votes. All the other groups which have corresponding X_i as one will not get a majority of votes, since the objective function would have assigned X_i as zero if it could have gotten the majority of votes. To ensure that the constraint is satisfied even when G_i does not have a majority (and votes assigned to group G_i can even be zero), d must be greater than or equal to the majority of votes.

Example. Let us consider a three-node fully connected network, with each component having a reliability of 0.9. Let the arrival rates of requests at all the nodes be 200 per unit of time and let $k = 3$. The top 3 partitions are {1,2,3}, {1,2}, and {2,3} (the group {1,3} is equally likely to {1,2} or {2,3}). The fraction of total simulation time for which these groups are present is (found through simulation) approximately

0.7, 0.08, 0.08, respectively. The average number of operations performed in each of the groups per unit of time is therefore 420, 32, and 32, respectively. The above formulation will result in (choosing d to be 500):
Minimize:

$$420X_1 + 32X_2 + 32X_3$$

subject to:

$$-V_1 - V_2 - V_3 - 1000X_1 \leq -1$$

$$-V_1 - V_2 + V_3 - 1000X_2 \leq -1$$

$$V_1 - V_2 - V_3 - 1000X_3 \leq -1 \ .$$

$$X_1, X_2, X_3 = 0 \text{ or } 1$$

Solving this will result in a vote assignment of (0,1,0). This vote assignment gives a majority of votes to all the three groups ({1,2,3},{1,2},{2,3}). Since node 2 is present in all the groups, it has given all the votes (1 vote in this case) to node 2, which will give a majority of votes to all the above groups.

7.5 Degree of Replication

We have discussed various techniques to manage replicated data. So far, we have focused only on the correctness of approaches, that is, the replica management algorithm should ensure a single-copy view. However, the goal of replication is to increase reliability and availability by keeping the data accessible even when failures occur in the system. It is clear that the reliability of a system will generally increase as the number of replicas, or the *degree of replication*, increases, since more replicas will be able to mask more failures. However, it is not clear how the degree of replication will affect system availability.

Since the algorithms for replica management involve overhead which usually increases as the number of replicas increases, it is unlikely that the system availability will continue to increase as the degree of replication increases. There are two opposing forces for the degree of replication: the increase in reliability as the degree of replication increase tends to increase availability, whereas the increase in overhead tends to decrease availability. In this section, we will study the effect of the degree of replication on the system availability, and the optimum degree of replication that maximizes the system availability. We will study the two most popular techniques: the primary site approach and majority voting.

7.5.1 Primary Site Approach

We first discuss the effect of the degree of replication on the availability of the primary site approach. The discussion is based on the models proposed in [HJ89b, HJ89a]. Consider a network of N homogeneous nodes, each with a lifetime that is exponentially distributed with the mean $\frac{1}{f}$. The sites are linearly ordered. At the start, the first site is chosen as the primary site; all the other sites are specified as the backups. The state of the primary site is periodically checkpointed on all the backups with the rate c, and the time required for a checkpoint is $\frac{1}{h}$. The service time for an operation on the data is exponentially distributed with the mean $\frac{1}{\mu}$, and the inter arrival time of requests for operations in the system is exponentially distributed with the mean $1/\lambda$. If the primary site fails, all operations after the last checkpoint are first redone by the new primary site in order to reach a consistent state. Let $\frac{1}{r}$ be the time required for recovery. $1/r$ is assumed to be much smaller than $1/f$. A failed site is repaired and it rejoins the system after it is repaired. The repair service is exponentially distributed with the rate δ.

It is assumed that the checkpointing cost is exponentially distributed, and the primary site also checkpoints the state on itself. Let b be the expected cost of checkpointing by the primary on a backup. The primary site checkpoints its status on its backups, one at a time. Assuming that the cost of establishing a checkpoint is the same at each node, we get the total cost of checkpointing as $\frac{1}{h} = b \times N$.

The availability of this system, α, is defined as the fraction of time that the system is available for serving user requests. The system can be in one of the two states: the normal state or the idle state.

- Normal state: the state in which at least one site is working. In this state, the system performs three kinds of activities: operations, recovery, and checkpoints.
  - Operations: the primary site is available for serving user requests. Let T_N be the random variable representing the total time that the system is operating.
  - Recovery: the system is recovering from the primary node failure. Let T_R be the random variable representing the total recovery time of the system.
  - Checkpoints: the primary site is checkpointing its status on the backups. Let T_C be the random variable representing the total checkpointing time.
- Idle state: the state in which all sites have failed. In this state, all sites are

waiting to be repaired and no services can be provided. Let T_I be the random variable representing the total idle time.

The availability α is the fraction of time that the system is available for user requests. If O is the expected overhead, then $\alpha = 1 - \frac{O}{L}$. The overhead occurs because checkpointing, recovering, and repairing are needed to make the system operational. The expected overhead within the period L is given by $O = E(T_I) + E(T_R) + E(T_C)$, where $E()$ represents the expected value of a random variable. To compute the overhead within a large period L, we need to compute the expected idle time, checkpointing time, and recovery time.

$E(T_I)$. The system is idle when all sites have failed. Let p_0 be the probability that no site is alive. Since we assume that both the lifetime and the repair time of a site are exponentially distributed, we can represent the system by a Markovian model, where the state of the system is the number of working sites. For this model, we have $p_0 = 1/\sum_{k=0}^{N}(\frac{\delta}{f})^k \frac{1}{k!}$. Therefore,

$$E(T_I) = \frac{L}{\sum_{k=0}^{N}(\frac{\delta}{f})^k \frac{1}{k!}}.$$

It should be noted that the larger the N, the smaller is the p_0.

$E(T_R)$. To compute the total recovery time, we first determine the number of failures during the period L. Since the system is in the normal state for $(1 - p_0)L$ seconds and the mean time between failures is $\frac{1}{f}$, the total number of failures during the period L is given by $L(1 - p_0)f$. Hence, the expected total time spent for recovery is:

$$E(T_R) = \frac{L(1 - p_0)f}{r}.$$

$E(T_C)$. The primary site checkpoints its state at a rate of $1/c$, and no checkpointing occurs during recovery. Hence, the expected number of checkpoints during L is: $c.((1 - p_0)L - \frac{L(1-p_0)f}{r})$. Therefore, the total time that the system is in the checkpointing state is:

$$E(T_C) = \left(\frac{(1 - p_0)L - \frac{L(1-p_0)f}{r}}{1/c}\right) \times \frac{1}{h}.$$

From the above equations, the overhead can be computed as follows:

$$O = p_0 L + \frac{L(1 - p_0)f}{r} + \left((1 - p_0)L - \frac{L(1 - p_0)f}{r}\right) \cdot \frac{c}{h}.$$

The availability of the system is given by:

$$\alpha = (1 - p_0)(1 - \frac{f}{r})(1 - \frac{c}{h}).$$

The three terms in the expression for availability reflect the contribution of the three overhead-generating activities on the system. If the idle time of the system (reflected by p_0) or the checkpointing cost $(1/h)$ or the recovery cost $(1/r)$ increases, the availability decreases.

If a failure occurs, all the operations performed since the last checkpoint have to be redone. As the number of operations performed between the two checkpoints is λ/c, on an average $\lambda/2c$ operations need to be redone. Hence, it takes an average of $\frac{\lambda}{2\mu c}$ time to redo the requests since the last checkpoint. The recovery cost, $\frac{1}{r}$, is equal to $\frac{\lambda}{2\mu c}$. If the number of sites is fixed, the optimal checkpoint rate is determined by differentiating this expression with respect to c and setting the result to zero. From this we get:

$$\frac{\lambda}{2\mu c^2} - \frac{1}{fh} = 0.$$

Solving this equation, we get the optimal checkpoint rate as $c^* = \kappa\sqrt{h}$, where κ is given by $\kappa = \sqrt{\lambda f/2\mu}$. Using this value of the checkpoint interval in the expression for availability, we get the best availability with respect to checkpoint interval. Using this, we can study the effect of degree of replication and find the optimal value of N such that α is maximized. The availability of the system with c^* can be written as:

$$\alpha = (1 - p_0)\left(1 - f(\frac{\lambda\sqrt{Nb}}{2\mu\kappa})\right)(1 - \kappa\sqrt{Nb}).$$

As N increases, p_0 decreases, thereby increasing availability. On the other hand, increasing N increases the overhead of checkpointing, which reduces the availability. Hence, there is some value of N which maximizes α.

It is difficult to get the optimum degree of replication which maximizes α by differentiating the above equations and solving for N. A numerical computation is needed to search the optimum value of N. We illustrate it by an example [HJ89a].

Example. A fault tolerant system has N identical nodes, each with a mean lifetime of 600,000 seconds (approximately 1 week). The mean service rate of the system is 5 transactions per second ($\mu = 5$), the total arrival rate is 3 transactions per second ($\lambda = 3$), and the mean repair time is 50,000 seconds ($1/\delta$ =50,000). When the primary server is performing a checkpoint, it takes 500 milliseconds to copy its status to a backup server (b=0.5). The availabilities are shown in Table 7.2. From the table, we

No. of Nodes	Availability
1	0.92215
2	0.98684
3	0.99559
4	0.99719
5	0.99746
6	0.99742
7	0.99729
8	0.99713
9	0.99698
10	0.99682

Table 7.2: An example

can see that this optimal N is 5. Increasing N beyond 5 will decrease the availability. However, the availabilities with $N = 2$ and $N = 3$ are only 1% less than the optimal availability. Hence, $N = 2$ and $N = 3$ also provide near optimal availability, and may be chosen due to practical considerations.

7.5.2 Majority Voting

Now we consider the issue of degree of replication and its effect on the availability of a majority voting system, based on the model proposed in [HJ93]. Consider a system consisting of N identical nodes, each having a replica of the data and with a mean lifetime of $1/f$. The average time to perform an operation on replicated data is exponentially distributed with the mean $\frac{1}{\mu}$. It is assumed that the node failures do not cause the network to partition. When a node fails, it is immediately sent to a repair server. After repair, the node rejoins the system as a working node. Assume that there is only one on-line repair server which uses the first-come-first-serve (FCFS) service discipline to repair failed nodes, and whose repair time is exponentially distributed with the mean $\frac{1}{\delta}$.

The system can perform operations as long as there are a sufficient number of working nodes in the system. Let M be the minimum number of nodes which must be working in order for the system to provide services. The threshold of M depends on the voting algorithm used in the controller and the type of operation (read or update). If the system uses a majority voting algorithm, M is equal to $\lceil \frac{N}{2} \rceil$. If the number of working nodes is less than M, the system cannot provide services and a total failure occurs. When a total failure occurs, all activities are suspended and the system has to be repaired before any service can be resumed. The system has the following two states:

Normal state: In this state, there are a sufficient number of working nodes to provide normal services. Let T_N be the random variable representing the time the system is in the normal state. T_N can also be interpreted as the life of the system. Let $E(T_N)$ represent the expected value of T_N.

Failure state: In this state, the system encounters a total failure. All services are suspended. Assume that negligible requests arrive when the system is in this state. Let T_F be the random variable that represents the time the system is in the idle state.

To compute the $E(T_N)$ and the $E(T_F)$ of the system, define the system state as the number of working nodes. The system is in state i when there are i nodes working at that moment. With this, the state $M-1$ represents a total failure of the system; in this state, all activities are suspended except for repairing the failed nodes. Define $t(i, j)$ as the expected time for the system starting at state i and first visiting state j.

There are two components in $t(i, i-1)$. First, the system has to stay in state i for s_i time units. Then, a transition occurs. It may go to state $i+1$ or to state i. If it goes to state $i-1$, it reaches the goal. On the other hand, if it goes to state $i+1$, it has to take $t(i+1, i)$ time for the system to go back to state i again and then it takes another $t(i, i-1)$ time for the system to reach state $i-1$. The probability of going to state $i+1$ from state i is $\frac{\delta}{i \cdot f + \delta}$. This can finally be written as a recurrence equation (the details are given in [HJ93] and are omitted here):

$$t(N, N-1) = \frac{1}{Nf},$$
$$t(i, i-1) = \frac{1}{if} + \frac{\delta}{if} \cdot t(i+1, i) \text{ for } i = N-1, \cdots, M.$$

The mean time to failure (MTTF) of the system is the duration between the time that the system starts services and the time that the system fails. And the mean time to repair of the system is the duration between the time that the system fails and the time that the system is working again. From the definitions, the MTTF of the system is equal to $E(T_N)$, and $E(T_N)$ is the expected time for the system to travel from state M to state $M-1$. Hence, we have $E(T_N) = t(M, M-1)$, or:

$$\text{MTTF} = E(T_N) = t(M, M-1).$$

Using the above recurrence equation, $t(M, M-1)$ can be determined. It is hard to obtain a closed-form solution from the recurrence equation. The recurrence equation can easily be solved numerically to determine $E(T_N)$. The mean time to repair (MTTR) is $E(T_F) = \frac{1}{\delta}$.

The availability of the system, α, is defined as the probability that the system is available to processing requests at any time. As described in the previous section, the system is cyclic, having a normal state and a failure state. Therefore, the probability that the system is available to processing requests is the portion of time that the system is in the normal state. In other words, the availability can be represented as:

$$\alpha = \frac{E(T_N)}{E(T_N) + E(T_F)} = \frac{t(M, M-1)}{t(M, M-1) + \frac{1}{\delta}}.$$

The computation of $t(M, M-1)$ is specified in the recurrence equation above.

From the definition of system availability, it is clear that system availability is optimized when the MTTF is maximized. The MTTF is a function of the degree of replication, N. For a majority-consensus voting system, the total number of non-failure states is $\frac{N+1}{2}$. Therefore, when N increases, the total number of non-failure states increases. That is, there are more failures that can be tolerated (degree of fault tolerance). In this case, increasing N tends to increase the MTTF. But, on the other hand, the failure transition rates between states also increase as N increases. For example, the transition rate from state N to state $N-1$ is Nf. Hence, increasing N increases the failure transition rate from state N to state $N-1$. Similarly, increasing N increases the transition rates from state $N-1$ to state $N-2$, from state $N-2$ to state $N-3$, etc. In this case, increasing N tends to decrease the MTTF. In other words, increasing N increases the degree of fault tolerance but also increases the failure transition rates. Therefore, there is an optimal N such that the availability of the system is maximized.

The value of the optimal N depends on the ratio of the life of a node and the repair time of the repair server. If the ratio is very large and if N is small, the degree of fault tolerance becomes a dominant factor. In this case, increasing N increases the degree of fault tolerance and, hence, improves the system availability. The system availability increases as N increases until N becomes too large and the failure transition rates become dominant. At that point, any further increase of N results in the decrease of system availability. This threshold point (the optimal degree of replication) usually occurs at a large N when the repair time is small (compared to the life of a node). On the other hand, if the ratio is small and N is small, the failure transition rates become the dominant factor. Consequently, increasing N degrades the system availability. Therefore, the optimal degree of replication in this case is a small number.

Example. Consider a majority-voting system with each node having a mean life of 30 days. The optimal degrees of replication of the system for different repair times are numerically computed. Table 7.3 gives the optimal degrees of replication for

various repair times. From the table, we find that the optimal N for system availability

Repair time (day)	Optimal N	Optimal MTTF	Optimal α
1	15	4794	0.999791
2	9	213	0.990735
3	5	85	0.965909
4	5	57	0.93429
5	3	45	0.9
10	3	30	0.75
15	3	25	0.625

Table 7.3: Effect of degree of replication in majority voting

is 15 when the repair time is 1 day. As expected, the optimal N decreases as the repair time increases. If the mean repair time is greater than 5 days, the optimal N is always 3 (since we are considering a majority-voting system, the minimum degree of replication is 3).

Note that when the repair server takes 15 days to repair a failed node, the MTTF of the system is shorter than that of a node (25 days *vs.* 30 days) and the availability of the system is smaller than that of a node (0.625 *vs.* 0.667). In other words, when the repair time is 15 days, using the voting approach for fault tolerance does not improve the MTTF and availability of the system.

7.6 Summary

In this chapter, we have discussed the problem of making data resilient to failures in the system. That is, masking node and communication failures in the distributed system to users in order to perform operations on data. Resiliency is supported by replicating the data on multiple nodes. Since the goal is to mask failure, and replication is a technique to support this, replication should be "hidden" to the users of the data. That is, even with replication, it should appear that there is a single copy of the data in the system, and one-copy serializability is maintained. There are two basic approaches for managing replication. First is the *optimistic approach*, where no restriction is placed on access to data. Consequently, if partition occurs, the copies of the data object in different groups may become divergent. The second approach is called the *pessimistic approach*, in which access to replicated data is controlled such that inconsistency never results, and single-copy serializability is preserved.

The goal of an optimistic approach is to resolve any inconsistencies that may arise due to unrestricted processing during partitioning. This attempt to resolve the inconsistencies is made after the groups rejoin, and is not always successful.

stem availability

ıal α
791
735
909
429
)
5
25

)ting

decreases as the
the optimal N is
ninimum degree

node, the MTTF
d the availability
her words, when
lerance does not

ilient to failures
ın the distributed
ıcy is supported
ıask failure, and
den" to the users
ɜ is a single copy
d. There are two
approach, where
ccurs, the copies
second approach
lata is controlled
y is preserved.
tencies that may
tempt to resolve
ways successful.

We have discussed one particular method based on *version vectors* for resolving this inconsistency. With each copy of a file, a version vector is associated, which represents the number of updates, originating at different nodes, which are performed on the copy. If a version vector of a group dominates the version vector of another, it means that one group has seen a subset of updates of another group and the inconsistency is resolved easily. If no version vector dominates, it represents a conflict situation. Conflict resolution is left to the user.

We have discussed three pessimistic approaches for replica control: the primary site approach, the state machine approach, and voting. In the *primary site approach*, one of the sites having the data is designated as primary, while the others are designated as backups. All requests for the operations on the data are sent to the primary site. The primary periodically checkpoints the state of the data on the backups. If the primary fails, one of the backups becomes the primary and starts performing the operations. The single-copy serializability is preserved, since all the requests for operations on the data are performed at a given time by only one node. If communication failures cause the network to get partitioned, then the partition that has the primary site is able to perform the operations on the data, while the others cannot access the data. This requires that the nodes which are alive must be able to distinguish between node failures and network partitions.

The second approach we discussed utilized *active replicas*. This is also called the *state machine approach*. In this approach, all replicas of the data are kept simultaneously active. A request for an operation is sent to all the replicas, and any replica can service the request. To ensure that the replicas are mutually consistent and that one-copy serializability is maintained, all the requests for operations must be sent to all the replicas, and the different requests must be processed by the different nodes in the same order. This ensures mutual consistency. One way to support the properties of mutual consistency and one-copy serializability is to use *atomic broadcast*. A request for an operation is atomically broadcast to all nodes having the copy of the data. Atomic broadcast ensures that the different requests are processed by different nodes in the same order. This approach works only for node failures.

The third approach we discussed was *voting*. In voting, for performing a read or an update operation on the data, the requesting node has to first get a "quorum" of votes from the nodes. The group of nodes that give the requester the quorum is such that a group performing the read operation always has a node with the latest update, and the read-write and write-write mutual exclusion property is supported. That is, when a group is performing a read or an update operation, no other group can perform an update operation. This property provides the single-copy view of the replicated data. Since voting treats all failures uniformly, it can handle both site failures and network partitioning and preserve data consistency. Voting algorithms

can be static or dynamic. In static voting, all parameters for voting are fixed, while in dynamic approaches, some parameters may be changed as failures and recoveries take place in the system.

We have discussed some static voting algorithms. The first one is weighted voting. In this, each copy of the object is assigned certain votes. A node wishing to perform a read or a write operation must collect a read quorum of r votes or a write quorum of w votes, respectively. The quorums are such that w is greater than half of the total number of votes and $r + w$ is greater than the total number of votes. These restrictions ensure that any two write quorums intersect and any read and write quorums intersect. One of the nice features of this approach is that it treats node failures and communication failures uniformly; the weighted voting strategy works even if the network partitions. A generalization of this approach that can reduce communication overhead is *hierarchical voting*. In this, nodes are logically organized in a hierarchy and quorums are collected at each level for an operation.

We discussed two different dynamic methods for voting. In *dynamic voting*, an operation is allowed if it has a quorum of a majority of the previous majority partition. In other words, if a partition occurs, the majority partition is taken to be the "system." If a further partition occurs, a majority of the nodes of this system are needed for an operation. On rejoining of groups, a node may update itself if it can communicate with some member of the last majority group. The second approach that we discussed is *dynamic reassignment of votes*. In this approach, if nodes fail, the votes to the nodes are reassigned such that the effect of failed nodes is compensated. One method of reassignment is to assign twice the votes of nodes that are not in the majority partition to nodes in the majority partition. This way, the effect of nodes that are in the minority partition is compensated.

The performance and reliability of voting systems depend considerably on the votes assigned to a node. We have discussed the problem of vote assignment, and have discussed three approaches to it. The first one is based on *coteries*, in which all possible groups that can be formed by vote assignment are assigned. This approach is only useful for small systems. The second approach is the use of heuristics, and we discussed a few different heuristics for vote assignment. The third approach we discussed considers the most likely partitions and then formulates the vote assignment problem as an integer programming problem.

These three replica-control approaches mask replication and failures by different techniques. The primary site approach masks replication by routing all requests to one site. The active-replicas approach manages replicas by ensuring that all sites get the requests in the same order, and hence all replicas are always equivalent, and capable of servicing any request on the data. The voting approach masks replication by ensuring mutual exclusion between groups performing the operations. The first

two approaches
primary site app
can be distinguis
mutual exclusio
groups. Hence,
can handle both.

Finally, we d
we continue to b
so will the cost
optimum degree
the optimal degr

We have ass
is, we focused o
by the use of re
there can alway
a situation is ca
be needed after
be performed, t
will have the m
node to fail. W
the next chapte
performed by th
for recovering f

Problems

1. What ad

2. Suppose
replica o
At time
are being
what con

3. One met
primary
on stable
uses two
checkpo

two approaches work well when only sites can fail, but no partitioning occurs. The primary site approach can also be used under network partitioning, if partitioning can be distinguished from site failures. Voting is a general approach which supports mutual exclusion of operations without communication between nodes of different groups. Hence, it treats both node failures and network partitioning uniformly, and can handle both.

Finally, we discussed the issue of degree of replication. Clearly, in no system can we continue to benefit by increasing replication. Though the reliability may increase, so will the cost of managing the replicas. Hence, it is likely that there may be some optimum degree of replication. We have discussed analytic models for determining the optimal degree of replication for the primary site approach and weighted voting.

We have assumed throughout the chapter that some nodes are always alive. That is, we focused on masking the failure of nodes and communication failures entirely by the use of replication. Hence, the issue of recovery was not discussed. However, there can always be situations where all the nodes in the network may fail. Such a situation is called *total failure*. In such situations, some recovery activities will be needed after the nodes recover from failure. However, before any recovery can be performed, the most recent replica has to be obtained. The node that failed last will have the most recent replica, and hence, this requires that we determine the last node to fail. We will discuss this problem of determining the last process to fail in the next chapter. Most of the recovery activities from a total failure will clearly be performed by the last node to fail, after it recovers. We will not discuss the methods for recovering from total failure.

Problems

1. What additional problems does data replication introduce?

2. Suppose there are four nodes — A, B, C, and D — in a system, each with a replica of a file f. Suppose node B fails at time T_1, separating A from {C, D}. At time T_2, B comes back up and reconnects the network. If version vectors are being used, under what conditions will there be no conflicts, and under what conditions there will be conflicts?

3. One method for supporting the primary site approach is to have only the primary site perform the operations, but periodically checkpoint information on stable storage, which is accessible to backups. Suppose the primary site uses two-phase locking for concurrency control. What information should be checkpointed on the stable storage and when?