# 9 Replication[1]

## 9.1 Introduction

**Replication** is the technique of using multiple copies of a server or a resource for better availability and performance. Each copy is called a **replica**.

The main goal of replication is to improve availability, since a service is available even if some of its replicas are not. This helps mission critical services, such as many financial systems or reservation systems, where even a short outage can be very disruptive and expensive. It helps when communications is not always available, such as a laptop computer that contains a database replica and is only intermittently connected to the network. It is also useful for making a cluster of unreliable servers into a highly-available system, by replicating data on multiple servers.

Replication can also be used to improve performance by creating copies of databases, such as data warehouses, which are snapshots of TP databases that are used for decision support. Queries on the replicas can be processed without interfering with updates to the primary database server. If applied to the primary server, such queries would degrade performance, as discussed in Section 6.6 on Query-Update Problems in two-phase locking.

In each of these cases, replication can also improve response time. The overall capacity of a set of replicated servers can be greater than the capacity of a single server. Moreover, replicas can be distributed over a wide area network, ensuring that some replica is near each user, thereby reducing communications delay.

## 9.2 Replicated Servers

### The Primary-Backup Model

To maximize a server's availability, we should try to maximize its mean time between failures (MTBF) and minimize its mean time to repair (MTTR). After doing the best we can at this, we can still expect periods of unavailability. To improve availability further requires that we introduce some redundant processing capability by configuring each server as two server processes: a primary server that is doing the real work, and a backup server that is standing by, ready to take over immediately after the primary fails (see Figure 9-1). The goal is to reduce MTTR: If the primary server fails, then we do not need to wait for a new server to be created. As soon as the failure is detected, the backup server can immediately become the primary and start recovering to the state the former primary had after executing its last non-redoable operation. Since we are primarily interested in transactional servers, this means recovering to a state that includes the effects of all transactions that committed at the former primary and no other transactions. For higher availability, more backup servers can be used to guard against the possibility that the primary and backup fail.

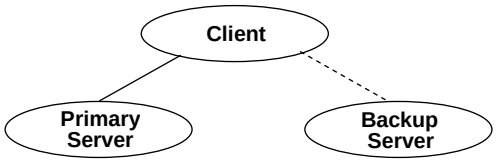



**Figure 9-1 Primary-Backup Model** The primary server does the real work. The backup server is standing by, ready to take over after the primary fails.

[1] This is a prefix of Chapter 9 of the forthcoming second edition of *Principles of Transaction Processing*, by Philip A. Bernstein and Eric Newcomer.

This technique is applicable to resource managers and to servers that run ordinary applications, such as request controllers and transaction servers. When a server of either type fails, it needs to be recreated. Having a backup server avoids having to create the backup server at recovery time.

To further reduce MTTR, each client connected to the primary server should also have a backup communication session with the backup server. This avoids recreating the sessions between the client and backup at recovery time. Since the time to recreate sessions can be quite long, even longer than the time to recover resource managers, this decreases (i.e., improves) MTTR.

In general, the degree of readiness of the backup server is a critical factor in determining MTTR. If a backup server is kept up-to-date so that it is always ready to take over when the primary fails with practically no delay, then it is called a **hot backup**. If it has done some preparation to reduce MMTR but still has a significant amount of work to do before it is ready to take over from the primary, then it is called a **warm backup**. If it has done no preparation, then it is called a **cold backup**.

As in the case of a server that has no backup, when the primary server fails, some external agent, such as a monitoring process, has to detect the failure and then cause the backup server to become the primary. The delay in detecting failures contributes to MTTR, so fast failure detection is important for high availability.

Once the backup server has taken over for the failed primary, it may be worthwhile to create a backup for the new primary. An alternative is to wait until the former primary recovers, at which time it can become the backup. Then, if desired, the former backup (which is the new primary) could be told to fail, so that the original primary becomes primary again and the backup is restarted as the backup again. This restores the system to its original configuration, which was tuned to work well, at the cost of some unavailability while the original primary is recovering after the former backup was made to fail.

When telling a backup to become the primary, some care is needed to avoid ending up with two servers both of which believe they're the primary. For example, if the monitor process gets no response from the primary, it may conclude that the primary is dead. But the primary may actually be operating. It may just be slow because its system is overloaded (e.g., a network storm is swamping its operating system), and it therefore hasn't sent an "I'm alive" message in a long time, which the monitor interprets as a failure of the primary. If the monitor then tells the backup to become the primary, then two processes will be operating as primary. If both primaries perform operations against the same resource, they may conflict with each other and corrupt that resource. For example, if the resource is a disk they might overwrite each other, or if the resource is a communications line they may send conflicting messages. Sometimes, the resource includes a hardware lock that only one process (i.e., the primary) can hold, which ensures that only one process can operate as primary at a time.

## Replicating the Resource

As we discussed in Chapter 7, a server usually depends on a resource, typically a database. When replicating a server, an important consideration is whether to replicate the server's resource too. The most widely-used approach to replication is to replicate the resource (i.e., the database) in addition to the server that manages it (see Figure 9-2). This has two benefits. First, it enables the system to recover from a failure of a resource replica as well as a failure of a server process. And second, by increasing the number of copies of the resource, it offers performance benefits when access to the resource is the bottleneck. For example, the backup can do real work, such as process queries, and not just maintain the backup replica so it can take over when there is a failure.

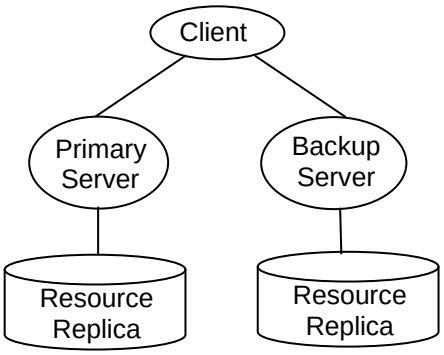
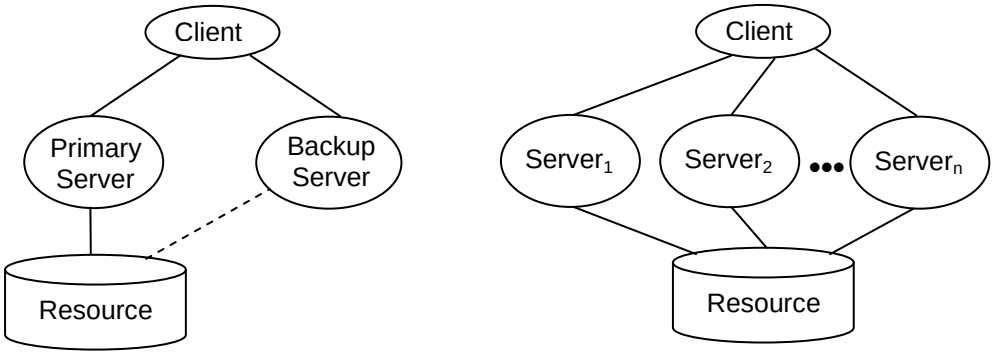



**Figure 9-2 Replicating a Server and its Resource.** Both the server and the resource are replicated, which enables recovery from a resource failure.

The main technical challenge in implementing this approach to replication is to synchronize updates with queries and each other when these operations execute on different replicas. This approach of replicating resources, and its associated technical challenges, are the main subject of this chapter, covered in Sections 9.3 - 9.1.

## Replicating the Server with a Shared Resource

Another approach is to replicate the server without replicating the resource, so that all copies of the server share the same copy of the resource (see Figure 9-3). This is useful in a configuration where processors share storage, such a storage area network. In a primary-backup configuration, if the primary fails and the resource is still available, the backup server on another processor can continue to provide service (see Figure 9-3a).




(a) Primary and Backup share the resource

(b) Many servers concurrently share the resource

**Figure 9-3 Replicated Server with Shared Resource.** In (a), the primary and backup server share the resource, but only one of them uses the resource at any given time. In (b), many servers share the resource and can concurrently process requests that require access to the resource.

This primary-backup approach improves availability, but not performance. If the server is the bottleneck and not the resource, then performance can be improved by allowing multiple servers to access the resource concurrently, as shown in Figure 9-3b. This approach, often called data sharing, introduces the problem of conflicts between transactions executing in different servers and accessing the same data item in the resource. One solution is to partition the resource and assign each partition to one server. That way, each server can treat the partition as a private resource and therefore use standard locking and recovery algorithms. If a server fails, its partition is assigned to another server, like in the primary-backup approach. Another solution is to allow more than one server to access the same data item. This solution requires synchronization between servers and is discussed in Section 9.1.

## *9.3 Replicating Servers and Resources*

### One-Copy Serializability

Replicas should behave functionally like non-replicated servers. This goal can be stated precisely by the concept of one-copy serializability, which extends the concept of serializability to a system where multiple replicas are present. An execution is **one-copy serializable** if it has the same effect as a serial execution on a one-copy database. We would like a system to ensure that its executions are one-copy serializable. In such a system, the user is unaware that data is replicated.

In a system that produces serializable executions, what can go wrong that would cause it to violate one-copy serializability? The answer is simple, though perhaps not obvious: a transaction might read a copy of a data item, say $x$, that was not written by the last transaction that wrote other copies of $x$. For example, consider a system that has two copies of $x$, stored at locations A and B, denoted $x_A$ and $x_B$. Suppose we express execution histories using the notation of Section 6.1.1, where $r$, $w$ and $c$ represent read, write, and commit operations (respectively) and subscripts are transaction identifiers. Consider the following execution history:

$$H = r_1[x_A]\ w_1[x_A]\ w_1[x_B]\ c_1\ r_2[x_B]\ w_2[x_B]\ c_2\ r_3[x_A]\ w_3[x_A]\ w_3[x_B]\ c_3$$

This is a serial execution. Each transaction reads just one copy of $x$; since the copies are supposed to be identical, any copy will do. The only difficulty with it is that transaction $T_2$ did not write into copy $x_A$. This might have happened because copy $x_A$ was unavailable when $T_2$ executed. Rather than delaying the execution of $T_2$ until after $x_A$ recovered, the system allowed $T_2$ to finish and commit. Since we see $r_3[x_A]$ executed after $c_2$, apparently $x_A$ recovered after $T_2$ committed. However, $r_3[x_A]$ read a stale value of $x_A$, the one written by $T_1$, not $T_2$.

When $x_A$ recovered, it should have been refreshed with the newly updated value of $x$ which is stored in $x_B$. However, we do not see a write operation into $x_A$ after $T_2$ committed and before $r_3[x_A]$ executed. We therefore conclude that when $r_3[x_A]$ executed, $x_A$ still had the value that $T_1$ wrote.

Clearly, the behavior of H is not what we would expect in a one-copy database. In a one-copy database, $T_3$ would read the value of $x$ written by $T_2$, not $T_1$. There is no other serial execution of $T_1$, $T_2$, and $T_3$ that has the same effect as H. Therefore, H does not have same effect as any serial execution on a one-copy database. Thus, it is not one-copy serializable.

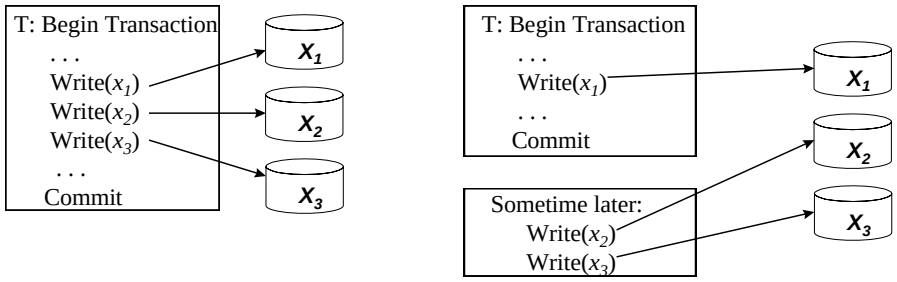
One obvious implication of one-copy serializability is that each transaction that writes into a data item $x$ should write into all copies of $x$. However, when replication is used for improved availability, this isn't always possible. The whole point is to be able to continue to operate even when some copies are unavailable. Therefore, the not-so-obvious implication of one-copy serializability is that each transaction that reads a data item $x$ must read a copy of $x$ that was written by the most recent transaction before it that wrote into any copy of $x$. This sometimes requires careful synchronization.

Still, during normal operation, each transaction's updates should be applied to all replicas. There are two ways to arrange this: replicate update operations or replicate requests. In the first case, each request causes one transaction to execute. That transaction generates update operations each of which is applied to all replicas. In the second case, the request message is sent to all replicas and causes a separate transaction to execute at each replica. We discuss each case, in turn.

### Replicating Updates

There are two approaches to sending a transaction's updates to replicas: synchronous and asynchronous. In the **synchronous** approach, when a transaction updates a data item, say $x$, the update is sent to all replicas of $x$. These updates of the replicas execute within the context of the transaction. This is called synchronous because all replicas are, in effect, updated at the same time (see Figure 9-4a). While this is sometimes feasible, it often is not, because it produces a heavy distributed transaction load. In particular, it implies that all transactions that update replicated data have to use two-phase commit, which entails significant communications cost.




(a) synchronous replication (b) asynchronous replication

**Figure 9-4 Synchronous vs. Asynchronous Replication.** In synchronous replication, each transaction updates all copies at the same time. In asynchronous replication, a transaction only updates one replica immediately. Its updates are propagated to the other replicas later on.

Fortunately, looser synchronization can be used, which allows replicas to be updated independently. This is called **asynchronous** replication, where a transaction directly updates one replica and the update is propagated to other replicas later on (see Figure 9-4b).

Asynchronous updates from different transactions can conflict. If they are applied to replicas in arbitrary orders, then the replicas will not be identical. For example, suppose transactions $T_1$ and $T_2$ update $x$, which has copies $x_A$ and $x_B$. If $T_1$ updates $x_A$ before $T_2$, but $T_1$ updates $x_B$ after $T_2$, then $x_A$ and $x_B$ end up with different values. The usual way to avoid this problem is to ensure that the updates are applied in the same order to all replicas. By executing updates in the same order, all replicas go through the same sequence of states. Thus, each query (i.e., read-only transaction) at any replica sees a state that could have been seen at any other replica. And if new updates were shut off and all in-flight updates were applied to all replicas, the replicas *would* be identical. So as far as each user is concerned, all replicas behave exactly the same way.

Applying updates in the same order to all replicas requires some synchronization. This synchronization can degrade performance, because some operations are delayed until other operations have time to complete. Much of the complexity in replication comes from clever synchronization techniques that minimize this performance degradation.

Whether synchronous or asynchronous replication is used, applying updates to all replicas is sometimes impossible, because some replicas are down. The system could stop accepting updates when this happens, but this is rarely acceptable since it decreases availability. If some replicas do continue processing updates while other replicas are down, then when the down replicas recover, some additional work is needed to recover the failed replicas to a satisfactory state. Some of the complexity in replication comes from ways of coping with unavailable servers and handling their recovery.

Replicas can be down either because a system has failed or because communication has failed (see Figure 9-5). The latter is more dangerous, because it may lead to two or more independently functioning partitions of the network, each of which allows updates to the replicas it knows about. If a resource has replicas in both partitions, those replicas can be independently updated. When the partitions are reunited, they may discover they have processed incompatible updates. For example, they might both have sold the last item from inventory. Such executions are not one-copy serializable, since it could not be the result of a serial execution on a one-copy database. There are two solutions to this problem. One is to ensure that if a partition occurs, only one partition is allowed to process updates. The other is to allow multiple partitions to process updates and to reconcile the inconsistencies after the partitions are reunited—something that often requires human intervention.


(a) Replica3 fails

(b) Network connection to Replica3 fails

**Figure 9-5 Node and Communications Failures.** Replica1, Replica2, and Replica3 are connected by a network. In (a), Replica3 fails. In (b), the connection to Replica3 fails. Replica1 and Replica2 cannot distinguish these two situations, yet the system's behavior is quite different.

Circumventing these performance and availability problems usually involves compromises. To configure a system with replicated servers, one must understand the behavior of algorithms used for update propagation and synchronization. These algorithms are the main subject of this chapter.

## Replicating Requests

An alternative to sending updates to all replicas is to send the *requests* to run the original transactions to all replicas (see Figure 9-6). To ensure that all of the replicas end up as exact copies of each other, the transactions should execute in the same order at all replicas. Depending on the approach selected, this is either slow or tricky. A slow approach is to run the requests serially at one replica and then force the requests to run in the same order at the other replicas. This ensures they run in the same order at all replicas, but it allows no concurrency at each replica and would therefore be an inefficient use of each replica's resources.

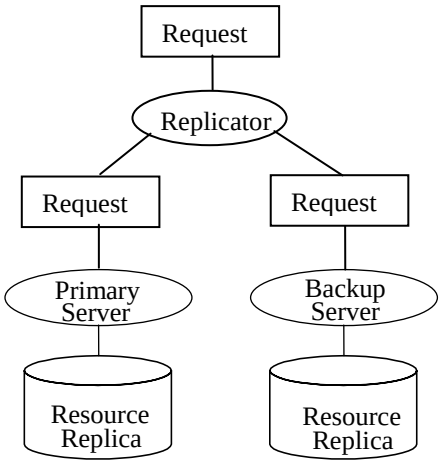



**Figure 9-6 Replicating Requests** Each transaction runs independently against each replica. In both cases, conflicting updates must be applied in the same order against all replicas.

The trickier approach is to allow concurrency within each replica and use some fancy synchronization across replicas to ensure that timing differences at the different replicas don't lead to different execution orders at different replicas. For example, in Digital's Remote Transaction Router (RTR), a replicated request can be executed at two or more replicas concurrently as a single distributed transaction. Since it runs as a transaction, it is serialized with respect to other replicated requests (which also run as transactions). It therefore can execute concurrently with other requests. Transaction synchronization (e.g., locking) ensures that the requests are processed in the same order at all replicas. As usual, transaction termination is synchronized using two-phase commit. However, unlike ordinary two-phase commit, if one of the replicas fails while a transaction is being committed, the other continues running and commits the

transaction. This is useful in certain applications, such as securities trading (e.g. stock markets), where the legal definition of fairness dictates that transactions must execute in the order they were submitted, so it is undesirable to abort a transaction due to the failure of a replica.

Replicating updates is a more popular approach than replicating requests, by far. Therefore, we will focus on that approach for the rest of this chapter.

## *9.4 Single-Master Primary-Copy Replication*

### Normal Operation

The most straightforward, and often pragmatic, approach to replication is to designate one replica as the primary copy and to allow update transactions to read and write only that replica. This is the primary-backup technique as illustrated in Figure 9-1. Updates on the primary are distributed to other replicas, called **secondaries**, in the order in which they executed at the primary and are applied to secondaries in that order (see Figure 9-7). Thus, all replicas process the same stream of updates in the same order. In between any two update transactions, a replica can process a local query.

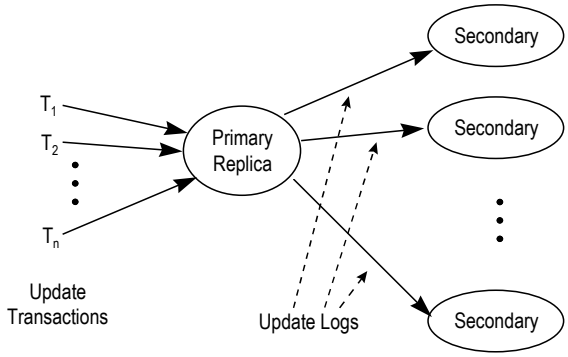



**Figure 9-7 Propagating Updates from Primary to Secondaries.** Transactions only update data at the primary replica. The primary propagates updates to the secondary replicas. The secondaries can process local queries.

One way to propagate updates is by synchronous replication. For example, in a relational database system, one could define an SQL trigger on the primary table that remotely updates secondary copies of the table within the context of the user's update transaction. This implies that updates are propagated right away, which may delay the completion of the transaction. It also means that administrators cannot control when updates are applied to replicas. For example, in some decision support systems, it is desirable to apply updates at fixed times, so the database remains unchanged when certain analysis work is in progress.

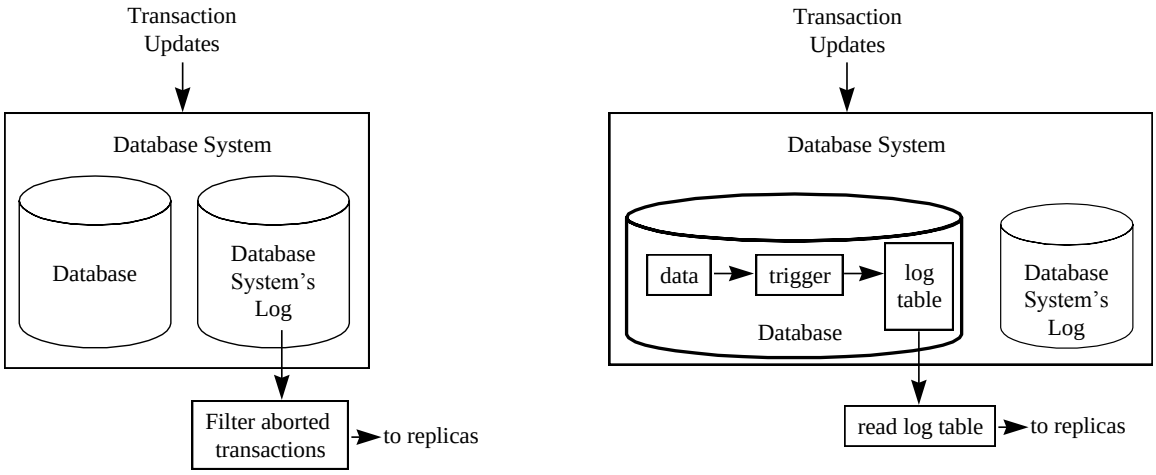
Currently, the more popular approach is asynchronous replication, where updates to the primary generate a stream of updates to the secondaries which is processed after transactions on the primary commit. For database systems, the stream of updates is often a log. The log reflects the exact order of the updates that were performed at the primary, so the updates can be applied directly to each secondary as they arrive.

The log can be quite large, so it is worth minimizing its size if possible. One technique is to filter out aborted transactions, since they do not need to be applied to replicas (see Figure 9-8a). This reduces the amount of data transmission and the cost of processing updates at the replica. However, it requires that the primary not send a log record until it knows that the transaction that wrote the record has committed. This introduces additional processing time at the primary and delay in updating the secondary, which are the main costs of reducing the data transmission. Another technique is to send only the finest granularity data that has changed, e.g., fields of records, rather than coarser-grain units of data, such as entire records.

Rather than using the database system's log, some relational database systems capture updates to each primary table in a log table that is co-located with the primary table (see Figure 9-8b). One approach is to have the system define an SQL trigger on each primary table that translates each update into an insert on the

log table. Periodically, the primary creates a new log table to capture updates and sends the previous log table to each secondary where it is applied to the replica. This approach to capturing updates can slow down normal processing of transactions, due to the extra work introduced by the trigger.




(a) Propagating updates from the database system's log

(b) Propagating updates from a log table generated by database triggers

**Figure 9-8 Generating Update Streams for Replicas** An update stream for replicas can be produced from (a) the database system's log or (b) a log table produced by triggers.

Another approach is to post-process the database log to create the stream of updates to the replicas. This avoids slowing down normal processing of transactions as compared to the trigger approach. In fact, the log can be sent to a different server, where the post-processing is done.

Some systems allow different parts of the database to be replicated to different locations. For example, the primary might contain a table describing customers and other tables describing orders. These tables are co-located at the primary, since many transactions require access to all of these tables. However, the customer table may be replicated at different servers than the order tables. To enable this, the log post-processing splits each transaction's updates into two transactions, one for the customer table and one for the order tables, and adds them to separate streams, one for the customer replicas and one for the order replicas. If there is also a replica that contains all of the customer and orders information, then the log post-processor would generate a third stream for that replica, with all of the updates of each transaction packaged in a single transaction in the stream.

Given this complex filtering and transaction splitting, often a "buffer database" is used to store updates that flow from the primary to secondaries. Updates that are destined for different replicas are stored in different areas of the buffer database. This allows them to be applied to replicas according to different schedules.

Some systems allow application-specific logic to be used to apply changes to replicas. For example, the application could add a timestamp that tells exactly when the update was applied to the replica.

Although primary-copy replication normally does not allow transactions to update a replica, there are situations where it can be made to work. For example, consider an update transaction that reads and writes a replica using two-phase locking. Suppose it keeps a copy of all the values that it read, which includes all of the data items that the transaction wrote. When it is ready to commit, it sends the values of data items that it read along with values that it wrote to the primary. Executing within the context of the same transaction, the primary reads the same data items that the transaction read at the secondary, setting locks as in normal two-phase locking. If the values of the data items that it reads at the primary are the same as those that the transaction read at the secondary, then the transaction applies its updates to the primary too and commits. If not, then it aborts. This is essentially an application of the optimistic concurrency control technique described in Section 6.8.

Most database systems offer considerable flexibility in configuring replication. Subsets of tables can be independently replicated, possibly at different locations. For example, a central office's Accounts table can be split by branch, and the accounts for each branch are replicated at the system at that branch. As the number of replicated tables grows, it can be rather daunting to keep track of which pieces of which tables are replicated at which systems. To simplify management tasks, systems offer tools for displaying, querying, and editing the configuration of replicas.

The replication services of most database systems work by constructing a log stream or log table of updates and sending it to secondary servers. This approach was introduced in Tandem's (now HP's) Non-Stop SQL and in Digital's VAX Data Distributor in the 1980's. Similar approaches are now offered by IBM, Informix (now IBM), Microsoft (SQL Server), Oracle and Sybase. Within this general approach, products vary in the specific features they offer: the granularity of data that can be replicated (a database, a table, a portion of a table); the flexibility of selecting primaries and secondaries (a server can be a primary server for some data and a secondary for others); how dynamically the configuration of primaries and secondaries can be changed; and facilities to simplify managing a large set of replicas.

## Failures and Recoveries

This primary-copy approach works well as long as the primary and secondaries are alive. How do we handle failures? Let us work through the cases.

### *Secondary Recovery*

If a secondary replica fails, the rest of the system continues to run as before. When the secondary recovers, it needs to catch up processing the stream of updates from the primary. This is not much different than the processing it would have done if it had not failed; it's just processing the updates later. The main new problem is that it must determine which updates it processed before it failed, so it doesn't incorrectly reapply them. This is the same problem as log-based database recovery that we studied in Chapter 7.

If a secondary is down for too long, it may be more efficient to get a whole new copy of the database rather than processing an update stream. In this case, while the database is being copied from the primary to the recovering secondary, more updates are generated at the primary. So after the database has been copied, to finish up, the secondary needs to process that last stream of updates coming from the primary. This is similar to media recovery, as in Chapter 7.

### *Primary Recovery with One Secondary*

If the primary fails, recovery can be more challenging. One could simply disallow updates until the primary recovers. This is a satisfactory approach when the main goal of replication is better performance for queries. In fact, it may be hard to avoid if rich filtering and partitioning of updates is supported. Since different secondaries receive a different subset of the changes that were applied to the primary, secondaries are often not complete copies of the primary. Therefore, it would be difficult to determine which secondaries should take over as primary for which parts of the primary's data.

If a goal of replication is improved availability for updates, then it is usually not satisfactory to wait for the primary to recover, since the primary could be down for awhile. So if it is important to keep the system running, some secondary must take over as primary. This leads to two technical problems. First, all replicas must agree on the selection of the new primary, since the system cannot tolerate having two primaries—this would lead to total confusion and incorrect results. Second, the last few updates from the failed primary may not have reached all replicas. If a replica starts processing updates from the new primary before it has received all updates from the failed primary, it will end up in a different state than other replicas that did receive all of the failed primary's updates.

We first explore these problems in a simple case of **database mirroring** where there are only two replicas, one primary and one secondary. This case is fairly common in practice. Many database products offer a high availability feature that uses this technique. This feature is usually distinct from the functionally rich, primary-copy replication feature described above.

So suppose there are only two replicas, and the secondary detects that the primary has failed. This failure detection must be based on timeouts. For example, the secondary is no longer receiving log records from the primary. And when the secondary sends "are you there?" messages to the primary, the secondary receives no reply from the primary. However, these timeouts may be due to a communications failure between the primary and secondary, similar to the one shown in Figure 9-5, and the primary may still be operating.

To disambiguate the case of primary failure from primary-secondary communications failure, an external agent is needed to decide which replica should be primary. A typical approach is to add a "watchdog" process, preferably on a different machine than the primary and secondary. The watchdog sends periodic "are you there?" messages to both the primary and secondary. If the watchdog can communicate with both the primary and secondary, but they cannot communicate with each other, then the watchdog notifies both the secondary and primary of this fact. It tells the secondary to fail, since it can no longer function as a replica. And it tells the primary to create another secondary, if possible. If the watchdog can communicate only with the secondary, then it tells the secondary that it believes the primary is down. If the secondary too is unable to communicate with the primary, then it can take over as primary. In this case, if the primary is still operational but is simply unable to communicate with the watchdog, then the primary must self-destruct. Otherwise, the old primary and the new primary (which was formerly the secondary) are both operating as primary. In summary, if the secondary loses communications with the primary, then whichever replica can still communicate with the watchdog is now the primary. If neither replica can communicate with the watchdog or with each other, then the neither replica can operate as the primary. This is called a **total failure**.

Suppose that the primary did indeed fail and the secondary has been designated to be the new primary. Now we face the second problem: the new primary may not have received all of the committed updates performed by the former primary before the former primary failed. One solution to this problem is to have the primary delay committing a transaction's updates until it knows that the secondary received those updates. The primary could wait until the secondary has stored those updates in stable storage, or it could wait only until the secondary has received the updates in main memory. If the system that stores the secondary has battery backup, then waiting until the updates are in main memory might be reliable enough. In either case, we're back to synchronous replication, where the updates to the replica are included as part of the transaction. This extra round of commit-time messages between the primary and secondary is essentially a simple two-phase commit protocol. The performance degradation from these messages can be significant. The choice between performance (asynchronous replication) and reliability (synchronous replication) depends on the application and system configuration. Therefore, database products that offer database mirroring usually offer both options, so the user can choose on a case-by-case basis.

## Primary Recovery with Multiple Secondaries

In the next three subsections, we look at the more general case where there are multiple secondaries.

### *Majority and Quorum Consensus*

Suppose there are multiple secondaries, and a secondary detects the failure of a primary. This could be the result of a communication failure that partitions the network into independent sets of functioning replicas. In this case the primary could still be operational, so the set of replicas that doesn't include the primary must not promote one of the secondaries to be a primary. The same problem can arise even if the primary is down. In this case, we do want to promote one of the secondaries to become primary. But if there are two independent sets of replicas that are operating, each set might independently promote a secondary to be the primary, a situation that we want to avoid.

One simple way to ensure that only one primary exists is to statically declare one replica to be the primary. If the network partitions, the partition that has the primary is the one that can process updates. This is a feasible approach, but it is useless if the goal is high availability. If the primary is down, each partition has to assume the worst, which is that the primary is really running but not communicating with this partition. Thus, neither partition promotes a secondary to become primary.

A more flexible algorithm for determining which partition can have the primary is called **majority consensus**: a set of replicas is allowed to have a primary if and only if the set includes a majority of the replicas (see Figure 9-9). Since a majority is more than half, only one set of replicas can have a majority. Moreover, each partition can independently figure out if it has a majority. These are the two critical properties of majorities that makes the technique work.

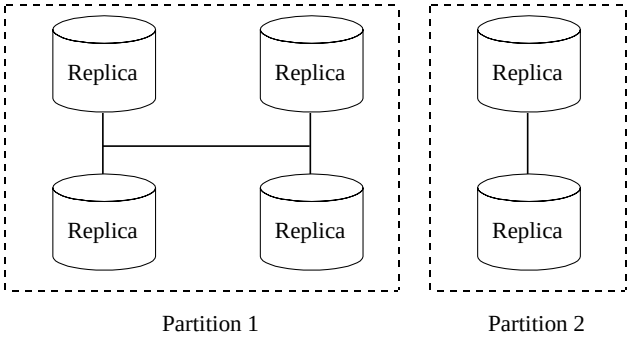



**Figure 9-9 Majority Consensus** Partition 1 has a majority of the replicas and is therefore allowed to process updates. Partition 2 may not process updates.

Majority consensus is a generalization of the watchdog technique we described for database mirroring. The watchdog adds a third process to the mix. Two communicating processes comprise a majority. Thus, whichever partition has at least two communicating processes is allowed to have the primary: either the existing primary and secondary if the watchdog is down; or the watchdog plus whichever replica(s) it can communicate with. By convention, if the watchdog can communicate with the primary and secondary but the latter cannot communicate with each other, then the secondary is told to fail.

Majority consensus does have one annoying problem: it does not work well when there is an even number of copies. In particular, it is useless when there are just two replicas, since the only majority of two is two, that is, it can operate only when both replicas are available. When there are four replicas, a majority needs at least three, so if the network splits into two groups of two copies, neither group can have a primary.

A fancier approach is the **quorum consensus** algorithm. It gives a **weight** to each replica and looks for a set of replicas with a majority of the weight, called a **quorum** (see Figure 9-10). For example, with two replicas, one could give a weight of two to the more reliable replica and a weight of one to the other. That way, the one with a weight of two can be primary even if the other one is unavailable. Giving a weight of two to the most reliable replica helps whenever there is an even number of replicas. If the network partitions into two groups with the same number of copies, the group with the replica of weight two still has a quorum.

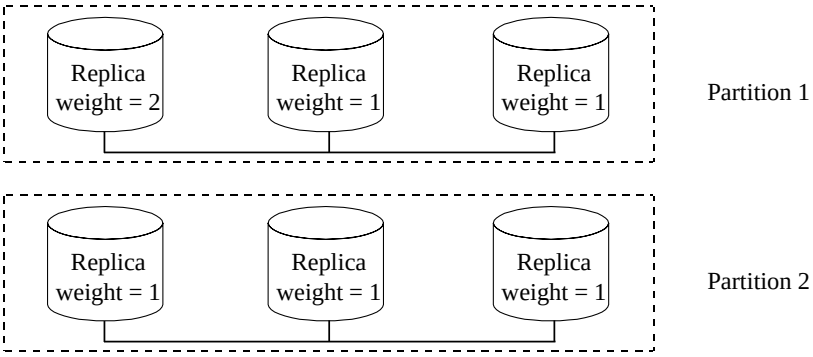



**Figure 9-10 Quorum Consensus** Partition 1 has a total weight of 4, which is more than half of the total weight of 7. It therefore constitutes a quorum and is allowed to process updates.

## *Reaching Consensus*

During normal operation, the set of operational replicas must agree on which replicas are up and which are down or unreachable. If a replica loses communication with one or more other replicas, then the operational

replicas need to reassess whether they still have a majority. (For the purpose of this discussion, we'll assume majority consensus, not quorum consensus.) In fact, the non-operational replicas that are up also need to do this when they reestablish communications with a replica, since this replica may be the one they need to reach a majority. After some group of replicas is established as having a majority, that group can choose a primary and ensure that all replicas in the group have the most up-to-date state.

To discuss the details, we need some terminology: The **replica set** is the set of all replicas, including those that are up and down. The **current configuration** is the set of operational replicas that are able to communicate with each other and comprise a majority.

An algorithm that enables a set of processes to reach a common decision is called a **consensus algorithm**. In this case, that common decision is agreement on the current configuration by a set of operational replicas. Given our problem context, we'll call the participants replicas instead of processes. But the algorithm we describe works for general consensus, not just for deciding on the current configuration.

One problem with such consensus algorithms is that multiple replicas may be trying to drive a common decision at the same time. It's important that different replicas don't drive the replicas towards different decisions. Another problem is that the system may be unstable, with replicas and communications links failing and recovering while replicas are trying to reach consensus. There's not much hope in reaching consensus during such unstable periods. However, once the system stabilizes, we do want the algorithm to reach consensus quickly.

There are several variations of algorithms to reach consensus, but they all have a common theme, namely, that there's a unique identifier associated with the consensus, that these identifiers are totally ordered, and that the highest unique identifier wins. We will call that identifier an **epoch number**. It identifies a period of time, called an **epoch**, during which a set of replicas have agreed on the current configuration, called an **epoch set**. An epoch number can be constructed by concatenating a counter value with the unique replica identifier of the replica that generated the epoch number. Each replica keeps track of the current epoch number $e$ in stable storage.

During stable periods, the epoch set with largest epoch number is the current configuration. During unstable periods, the actual current configuration may differ from the current epoch set. The goal of the consensus algorithm is to reach agreement on a new epoch set with associated epoch number that accurately describes the current configuration.

Suppose a replica R is part of the current configuration, which has epoch number $e_1$. If R detects that the current configuration is no longer valid (because R has detected a failure or recovery), R becomes the leader of a new execution of the consensus algorithm, which proceeds as follows:

1. R generates a new epoch number $e_2$ that is bigger than $e_1$. For example, it increments the counter value part of $e_1$ by one and concatenates it with R's replica identifier.
2. R sends an **invitation** message containing the value $e_2$ to all replicas in the replica set.
3. When a replica R′ receives the invitation, it replies to R with an **accept** message if R′ has not accepted another invitation with an epoch number bigger than $e_2$. R′ includes its current epoch number in the accept message. Moreover, if R′ was the leader of another instance of the consensus algorithm (which is using a smaller epoch number), it stops that execution. Otherwise, if R′ has accepted an invitation with an epoch number bigger than $e_2$, it sends a **reject** message to R. As a courtesy, it may return the largest epoch number of any invitation it has previously accepted.
4. R waits for its timeout period to expire (to ensure it receives as many replies as possible).
    a. If R receives accept messages from at least one less than a majority of replicas in the replica set, then it has established a majority (including itself) and therefore has reached consensus. It therefore sends a **new epoch** message to all of the accepting replicas and stops. The new epoch message contains the new epoch number and epoch set. When a replica receives a new epoch message, it updates its epoch number and the associated list of replicas in the epoch set and writes it to stable storage.

b. Otherwise, R has failed to reach a majority and stops.

Let's consider the execution of this algorithm under several scenarios. First, assume that only one leader R is running this algorithm. Then it will either receive enough accept messages to establish a majority and hence a new epoch set. Or it will fail to reach a majority.

Suppose a leader $R_1$ fails to establish an epoch set. One reason this could happen is that $R_1$ may be unable to communicate with enough replicas to establish a majority. In this case, $R_1$ could periodically attempt to re-execute the algorithm, in case a replica or communication link has silently recovered and thereby made it possible for $R_1$ to form a majority.

A second reason that $R_1$ may fail to establish an epoch set is that another replica $R_2$ is concurrently trying to create a new epoch set using a higher epoch number. In this case, it is important that $R_1$ not re-run the algorithm right away with a larger epoch number, since this might kill $R_2$'s chance of getting a majority of acceptances. That is, it might turn into an "arms race," where each replica re-runs the algorithm with successively higher epoch numbers and thereby causes the other replica's consensus algorithm to fail.

The arms race problem notwithstanding, if $R_1$ fails to establish an epoch set and, after waiting awhile, receives no other invitations to join an epoch set with higher epoch number, then it may choose to start another round of the consensus algorithm. In the previous round, if it received a reject message with a higher epoch number $e_3$, then it can increase its chances of reaching consensus by using an epoch number even higher than $e_3$. This ensures that any replica that is still waiting for the result of the execution with epoch number $e_3$ will abandon waiting and choose the new, higher epoch number instead.

### *Establishing the Latest State*

After the current configuration has been established as the epoch set, the primary needs to be selected and all of the replicas in the current configuration have to be brought up to date. If the epoch set includes the primary from the previous epoch, then there is no need to switch primaries. Moreover, the primary has the most up-to-date state. So to ensure the secondaries are up-to-date, the primary simply needs to determine the state of each secondary and send it whatever updates it is missing.

If the epoch set does not include the primary from the previous epoch, then a new primary must be selected. The choice of primary may be based on the amount of spare capacity on its machine (since a primary consumes more resources than a secondary) and on whether it has the latest or a very recent state (so that it can recover quickly and start accepting new requests).

The latest state can be determined by comparing the sequence numbers of the last message received by each secondary from the previous primary. The one with highest sequence number has the latest state and can forward the tail of the sequence to other secondaries that need it. After a replica receives that state, it acknowledges that fact to the primary. After the new primary receives acknowledgements from all replicas in the epoch set, it can start processing new transactions. The new primary should then start off with a message sequence number greater than that of the largest received by any secondary in the previous epoch.

This recovery procedure relies on the majority consensus rule, which ensures that each epoch includes at least one replica that was in the epoch set of the previous epoch. These are the only replicas that know the latest state. This is why the new primary must wait for all replicas to acknowledge receipt of the latest state. Otherwise, if it starts processing transactions and fails before some secondaries are up-to-date, there's a risk that the next epoch won't have any up-to-date replicas in its epoch set.

This argument is predicated on the use of synchronous replication between the primary and secondaries. Otherwise, as with database mirroring, the last few transactions committed by the primary before it failed might not be known to any of the secondaries that are part of the next epoch. The same tradeoff between performance and reliability that we discussed for database mirroring applies here as well.