6.4.3 Quorum Voting

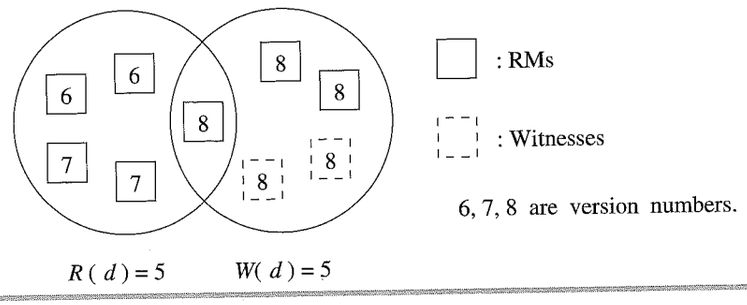
Read-quorum/write-quorum requires that each read operation to a replicated data object d must obtain a read quorum $R(d)$ of replica managers to perform the read. Similarly, each write operation needs a write quorum $W(d)$ to complete the write. This is a special case of the *semi-unanimous quorum voting* scheme for synchronization described in Chapter 4. Using the two quorums, conflicts between access operations on replicated data objects may be implicitly represented by an overlapping of the quorums. Let $V(d)$ be the total number of copies (votes) of object d. The overlapping rules for read and write are:

1. **Write-write conflict:** $2 * W(d) > V(d)$.
2. **Read-write conflict :** $R(d) + W(d) > V(d)$.

Once conflicts can be expressed, a standard two-phase locking or timestamp ordering protocol can be used to enforce one-copy serializability. To ensure that clients read the most recently *completed* update of data, a version number can be associated with each replicated data object. A read operation queries all $R(d)$ replicas. The value in the replica with the highest version number is chosen to be returned to the client. A write operation queries $W(d)$ replicas and adds 1 to the highest version number found: this is the new version number for writing to the replicas. Since the read and write quorums overlap with each other, a read is always returned with the most updated write. An example is shown in Figure 6.10, where $V(d)$ is 9 and both $R(d)$ and $W(d)$ are set to 5.

For most applications $R(d)$ is chosen to be smaller than $W(d)$. The special case of $R(d) = 1$ and $W(d) = V(d)$ represents the *read-one/write-all* protocol. If $W(d) < V(d)$, failures can be tolerated such as the *write-all-available*, but $R(d)$ must be larger than one. The *read-quorum/write-quorum* protocol is a good compromise between *read-one/write-all* and *read-one/write-all-available*. If $W(d)$ is much larger than $R(d)$, the data storage overhead for replicating data may be high. To reduce the storage requirement, the write-quorum may be extended to include *witnesses* (or *ghosts*). A witness does not carry a full replication of data. It maintains only the necessary information such as the version number and object identifier to participate in the write quorum. Two witnesses are shown in dash-line circles in Figure 6.10. The highest version number is 8.

FIGURE 6.10 A read-quorum/write-quorum with witnesses.



The quorum voting example shown assumes one vote per replica manager. Many generalized quorum voting schemes with variable vote weights have been proposed for different applications of replicated data. For example, heavier weights can be assigned to some preferred replica managers to enhance either performance or availability. Other issues about quorum voting are detailed in Chapter 12.

6.4.4 Gossip Update Propagation

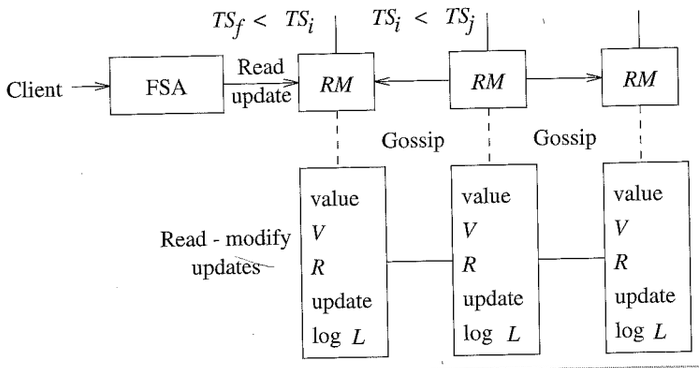
Many applications of data replication do not need as strong consistency as the one-copy serializability in *read-one/write-all-available* or the most recent update in the *read-quorum/write-quorum* protocol. If updates are less frequent than reads and ordering of updates can be relaxed, updates can be propagated *lazily* among replicas, which is similar to the dissemination of information by gossip in real life. This *read-one/write-gossip* approach is called a *gossip update propagation* protocol. In the gossip architecture, both read and update operations are directed by a file service agent (FSA) to any replica manager (RM). The FSA shields the replication details from the clients. Replica managers bring their data up to date by exchanging gossip information among themselves. The primary purpose of gossip architectures is to support high availability in an environment where failures of replicas are likely and reliable multicast of updates is impractical.

We will use the architecture illustrated in Figure 6.11 to demonstrate two simple implementations of the gossip update propagation protocol. The first assumes *overwrite* updates. The second uses *read-modify* updates, which depend on the current state of the data object. In the latter case an *update log* is used to record the history of updates so that data dependency can be observed. In the simplest implementation each RM i is associated with a timestamp TS_i which represents the last update of the data object. Each FSA also maintains a timestamp TS_f, which indicates the timestamp of its last successful access operation. Updates are *overwrite* and the group is not definitive, so no update log is necessary. The implementation can be described by actions taken by the FSA and RM when *read*, *update*, and *gossip* messages are presented to them.

Basic Gossip Protocol

- **Read:** TS_f of the FSA is compared with TS_i of the RM contacted. If $TS_f \leq TS_i$, the RM has more recent data, so its value is returned to the FSA and TS_f is set to TS_i. Otherwise, the FSA waits until the RM has been brought up to date through gossip. Alternatively, the FSA can try other RMs for more recent results.
- **Update:** The FSA increments TS_f. If $TS_f > TS_i$, the update is executed. TS_i is set to TS_f to reflect the new update. The RM can propagate the new knowledge at its own convenience by way of gossip. If $TS_f \leq TS_i$, the update has come too late or concurrent updates are being processed. Depending on the application, the FSA can either perform the overwrite (ignore the missing updates) or become more up to date by performing a read followed by the update. In either case timestamps must be advanced.

FIGURE 6.11 A gossip architecture.



- **Gossip**: A gossip message carrying a data value from replica manager j to replica manager i is accepted if $TS_j > TS_i$. The RM can repeat the gossip if so desired.

If the group is definitive so that all replica managers are known, vector timestamps can be used instead of single timestamps. The use of vector timestamps has a number of advantages that we will explain shortly. Since we are concerned only with updates of data objects, a vector timestamp can be composed of sequence numbers (or version numbers) representing the number of updates executed at each replica manager site. For example, the vector timestamp $TS_i =< 2, 3, 4 >$ shows that two, three, and four updates have occurred at RM_1, RM_2, and RM_3, respectively. A read operation with vector timestamp $TS_f =< 2, 2, 2 >$ pairwise and equal or less than TS_i is allowed to proceed. A TS_f of $< 2, 5, 5 >$ will need to wait, but we know more recent updates can be found at RM_2 and RM_3.

We used the concept of vector timestamps to implement causal order multicast in Section 4.1.5. Maintaining causal order is necessary for gossip propagation if updates are *read-modify*. The result of data multiplied by 2 and incremented by 1 is different from that when the order is reversed or an operation is missed. Read-modify updates are shorter messages. They are more efficient than using pairs of read and overwrite. Causal order gossip can be implemented by using vector timestamps and a buffering mechanism for orderly delivery of updates.

As shown in Figures 6.11 and 6.12, each replica manager is associated with two vector timestamps, V and R, and an update log L. V carries the timestamp of the current value of the object. R represents the replica manager's knowledge of update requests in the group. For example, $R =< 2, 3, 4 >$ at RM_1 indicates that exactly two updates originated from RM_1 and to the best of its knowledge three and four updates were initiated at RM_2 and RM_3, respectively. This estimated information is obtained through gossip. It is computed by *merging* (pairwise maximum) two Rs when exchanging gossip. The update log L is a list of update records collected by the RM. Each record in the log consists of a unique identifier r for the update u, in addition to other essential information such as

the operation type and value. Record r and update u are denoted by vector timestamps. u is the timestamp issued by the FSA for an update operation. r is made unique for update u by RM_i by taking the corresponding u and replacing the ith component of the vector timestamp u with the ith component of R. We summarize the causal order gossip protocol as follows:

Causal Order Gossip Protocol

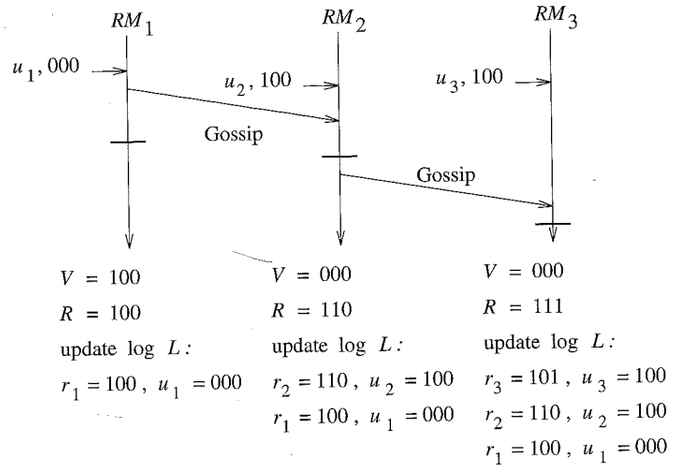
- **Read**: The FSA assigns its vector timestamp TS_f to the query q. If $q \leq V$, the RM returns the stable object value to the FSA. The FSA advances its timestamp by merging q with V. Otherwise the FSA either waits for the object to become up to date or contacts other RMs.

- **Update**: The FSA assigns its vector timestamp TS_f to the update u. It is compared with the current timestamp V of the object. If $u < V$, the update comes too late and is rejected by the RM since causal order needs to be enforced. The FSA contacted is out of touch. A read should have preceded the update. Under the normal situation, u is either greater than or equal to V or concurrent with V, meaning that neither $u < V$ nor $u > V$ (using vector comparison discussed in Section 3.4.3). In any case, the update is accepted. The replica manager RM_i increments the ith component of R by 1 indicating one more update has been processed at the replica site. A unique identifier r is generated and placed with the update operation u in the log L. If $u = V$, the update operation is executed. The timestamp V is advanced by merging with r. As a result of this execution, some records in the log may become *stable*, meaning that they satisfy the stable condition, $u \leq V$, and are ready to be executed. The stable condition implies that all operations that are supposed to *happen before* u have been committed. If more than one record satisfies this condition, their execution must be performed in causal order. This is easy to do because their vector timestamps reflect causal ordering. Other update operations wait in the log for the stable condition, which will be changed when the RM receives gossip that may contain executable updates. Updates that have been executed are marked so that they will not be repeated. Executed updates are not deleted immediately since they need to be propagated to other RMs.

- **Gossip**: A gossip message from RM_j to RM_i carries RM_j's vector timestamp R_j and log L_j. R_j is merged with the vector timestamp R_i in RM_i. The log L_i in RM_i is joined with L_j except for those update records with $r \leq V_i$. Records with $r \leq V_i$ must have been accounted for by RM_i. The aggregate of logs is a *true* gossip in that it contains not only the knowledge of one RM but also that of many others.

Figure 6.12 illustrates the execution of three update operations using update logs for causal ordering. All vector timestamps are initialized to $< 0, 0, 0 >$. Update $u_1 = < 0, 0, 0 >$ is directed to RM_1. It is accepted since u_1 is not less than V_1. R_1 is incremented to $< 1, 0, 0 >$. An update record of r_1 and u_1 is generated and placed in log L_1. It is executed and V_1 is updated to $< 1, 0, 0 >$. Meanwhile, L_1 which contains one executed update, is gossiped to RM_2. Updates u_2 and u_3 come to RM_2 and RM_3 separately.

FIGURE 6.12 A causal order gossip scenario.



They must have read data from RM_1 earlier since they have the timestamp $< 1, 0, 0 >$. u_2 arrives at RM_2 before the gossip. Its timestamp is larger than $V_2 = < 0, 0, 0 >$, so it is placed in log L_2 after R_2 has been updated to $< 0, 1, 0 >$ and r_2 has been assigned $< 1, 1, 0 >$. The arrival of the gossip from RM_1 adds r_1 to L_2 and advances R_2 to $< 1, 1, 0 >$. Similarly, u_3 is accepted by RM_3. A record of $r_3 = < 1, 0, 1 >$ and $u_3 = < 1, 0, 0 >$ is put into L_3. When the gossip arrives from RM_2, r_2 and r_1 are added to log L_3. R_3 merges with R_2 to become $< 1, 1, 1 >$. At the instance shown in the diagram, r_1 in L_2 becomes stable and is ready for execution. The execution of r_1, in turn, makes r_2 ready. Following the same procedure, the causal execution sequence will be r_1, r_2, and r_3 at RM_3. The same execution sequence will be observed by all three RMs when RM_3 sends gossip to RM_1 and RM_3.

The example shows the basic gossip update propagation concept. Other issues such as garbage collection of the logs and optimization of message count are presented in Chapter 12.

6.5 SUMMARY

This chapter presents the fundamental concepts and design issues for the implementation of distributed file systems. A distributed file system is characterized by how dispersion and multiplicity of files and users are made *transparent*, how *caching* and *replication* are supported to enhance performance and availability, what *file sharing semantics* is assumed, and whether the system is *scalable* and *fault tolerant.*

We show how file access and location transparencies can be achieved by using a hierarchical file naming structure and a file system mounting protocol. A distributed file system consists of a set of file servers that collectively perform transparent file services. A critical design factor of a DFS is whether the file servers should be *stateless* or *stateful.* Stateless file servers are robust and simple to implement. However, many file applications require management of the state information.

File sharing is a result of *overlapping* (space multiplexing) and/or *interleaving* (time multiplexing) file access operations to the same files. We describe two unique cases of file sharing due to overlapping: use of *client caches* to enhance file access time and *replication* of files to improve file availability. In both cases, data replicas exist and the primary implementation issue is the maintenance of data consistency for achieving replication transparency. We also show two unique examples of file sharing due to interleaving: *session* and *transaction* file accesses. Both access methods involve interleaved read and write operations to the same files. The main implementation consideration is to prevent interference among the sharing processes to achieve concurrency transparency.

Based on different objectives for file sharing, we discuss three common file sharing semantics models: *Unix*, *session*, and *transaction* semantics models. The Unix semantics model and its variations can be implemented by using the *write-through, write-back, write-invalidate*, and *write-update* cache coherence protocols. The choice of a cache coherence protocol is a trade-off between the implementation efficiency and the consistency requirement of the protocol. The session semantics model assumes that file sharing is infrequent for some applications. Thus, the model ignores the problems of sharing within a session and shifts the burden of coordinating file sharing between sessions to the application users.

For other applications (particularly database applications), file sharing is essential and also frequent. The transaction semantics model for these applications requires a concurrency control protocol to maintain the ACID semantics for concurrent transactions. We illustrate three common approaches for concurrency control in the context of a distributed transaction processing system. *Two-phase locking* is a pessimistic concurrency control protocol that achieves serializability by restricting the operations in a transaction. The *timestamp ordering* approach is a less pessimistic approach that allows operations in a transaction to proceed freely and resolves the serializability using timestamps only when conflicts exist. The *optimistic* concurrency control protocol ignores conflicts completely during the execution phase of a transaction. The result of the execution is committed only if consistency can be validated in the validation phase. We outline and discuss the pros and cons of each of the three concurrency control approaches.

Files can be replicated to allow for concurrent accesses and to tolerate failures. For many applications, a strong consistency requirement of the replicas is not necessary as long as data can be updated orderly. We show three replicated data management protocols for such applications, the *primary copy*, *quorum voting*, and the *gossip update* protocols. Since data replication implies the use of the notion of a *group*, we will revisit the topic of replicated data management in the context of group communication in Chapter 12.