Quorum Completion

Michael Leon Kazar

Introduction

This note describes a new replication strategy for maintaining databases in a distributed environment, called **quorum completion**. At least, I think it's new. Using this algorithm, we will provide build an application, called **ubik**, providing a simple abstraction: a set of disk files, replicated over a set of N servers.

Traditionally, this is accomplished by the use of atomic transactions @cite(Lampson). However, distributed atomic commit protocols are difficult to implement, are subject to blocking when failures occur during the commit operation, and have difficulty handling write operations when any servers are down.

This paper presents the design of a new atomic commit algorithm, called quorum completion, that remedies many of these problems. Specifically, our algorithm provides the following advantages. First, we require only an efficient and simple two-phase commit protocol to implement it. Second, our recovery process is non-blocking, once a majority of the servers recover. There is no window during which a crash will leave the database locked for an unbounded period as long as at least half of the server remain functional and connected by the network. Finally, our algorithm performs *write*, as well as *read*, transactions, as long as a majority of the servers remain operational. Its big disadvantage is that it allows at most one write transaction to run concurrently; although I believe this problem can be remedied, I do not indicate how to do so in this paper.

The remainder of this paper describes the implementation of the system. The system is implemented in two layers. The bottom layer handles synchronization site and quorum selection. The top layer provides the disk file abstraction and handles recovery within the new quorum. The top layer periodically calls the synchronization site election layer to determine in which type of quorum, if any, the server resides.

Basic Terms

The set of servers is divided into **quora**, such that every server is included in exactly one quorum. There are types of quora: **write** quora and **null quora**, corresponding to the operations allowed.

Each quorum is associated with a distinguished member of the quorum, called the synchronization site.

We will denote the quorum defined by synchronization site *A* at the time *T* as **Q(A,T)**. The discussion below will show that this is a well-defined definition. If we are not interested in the time a quorum is defined, the quorum will be denoted simply as **Q(A)**.

The Election Protocol and Quorum Definition

The election protocol elects a single synchronization site (or sync site) from those sites presently accessible within a network partition. A synchronization site *A* then defines a quorum by periodically contacting the sites that voted for its election. The quorum definition protocol guarantees one very useful invariant:

> For every time T, the intersection of Q(A,T) and Q(B,T), for two servers A != B, is empty.

In more detail, the election protocol is characterized by time constants, BIG, SMALL, RPCTIMEOUT and SKEW, such that

> BIG > SMALL
>
> BIG - SMALL > SKEW (the maximum clock skew in the system)
>
> SMALL > RPCTIMEOUT (the basic RPC timeout)

In steady state, a synchronization site sends beacon messages to all servers in its quorum every SMALL/2 seconds. This message contains two parameters: The first parameter is the the time T until which that the quorum is defined. The second is the type of quorum (null or write) that the synchronization site *A* was elected by. The receipt of such a beacon at a server *S* defines *S* as a member of Q(A) from the time the message is received until time T. Servers respond to a beacon with a response that is used to update the synchronization site’s value of T (its "last heard from" time).

The synchronization site computes its new value of T from the responses it received from its last beacon. When a normal server responds to a synchronization site’s beacon message, it guarantees that it hasn’t responded to another site’s beacon message in at least BIG seconds. The synchronization site then computes the length of time period during which theis guarantee ensures that the synchronization site is the only synchronization site elected by a write quorum, as follows. The synchronization site considers the last time it received a vote from each server in its quorum, and selects the earliest of these times. Call this time B. The synchronization site knows that between now and B+SMALL, the servers in its quorum can not vote for another synchronization site and still obey the restriction that they not vote for more than one synchronization site in any BIG second period. Thus the synchronization site can send a message to the entire quorum, guaranteeing the identity of the synchronization site until time T = B+SMALL.

In other words, when a server responds to a beacon message, it implicitly guarantees that it has not responded to another site’s beacon message in at least BIG seconds. Thus, an individual server’s vote can not count in more than one synchronization server’s quorum simultaneously, since one synchronization site will stop counting server S in its quorum after it hasn’t heard from S in SMALL seconds, while S won’t be able to respond positively to another server’s beacon message until BIG seconds have passed, and, of course, BIG > SMALL + SKEW. Thus no server can vote for more than one synchronization site simultaneously.

Note that the receipt of a beacon message allows a server to operate under the assumption that it is a member of a particular quorum for a period of time (usually SMALL/2 seconds) without any communication with any other servers in the quorum, making this an efficient protocol.

If a server finds itself running without being a member of a quorum, it begins sending beacons to the other servers in the system, trying to get itself elected as synchronization site of a quorum. At this time, it claims to represent only a null quorum. To avoid endless thrashing around during the election process, a server will only vote for the lowest numbered server (ordered by some standard constant ordering) that it has heard from over some time period, and it may change its vote, if it hears from a "better" server. Of course, when changing a vote, it must follow the rule about delaying at least BIG seconds before voting for a new synchronization site, to avoid voting for multiple synchronization sites simultaneously. In our system, we use the "<" relation over the server internet address as our standard ordering.

Write Quorum Invariants

Once we have partitioned the servers into quora, we can postulate more invariants. These invariants are built on top of the concept of a quorum.

The invariants described in this section are concerned with the operations one can perform by virtue of being a member of a quorum of a certain size.

> There is at most one write quorum in the system at any one time.
>
> The write quorum is large enough so that any write quorum formed later will have to include at least one server from the last write quorum.

These two invariants are simply the invariants required by the quorum consensus algorithm @cite(Herlihy).

The Database

A ubik database consists of a number of files, each of which consists of a sequence of 8 bit bytes. The files are simply named by positive integers. Associated with the database is a 64-bit version number.

The version number associated with the database consists of an ordered pair: a synchronization site timestamp (called the **epoch**), and a **commit count** qualifying the timestamp; each field is 32 bits long. The synchronization site timestamp represents the Unix time (with 1 second resolution) that the current synchronization site first received enough votes to become a synchronization site. The commit count is a counter, initialized to 1 when a new synchronization site is elected, of the number of write transactions that committed in this database. It is incremented at the end of every write transaction.

The Quorum Completion Algorithm

So far, we have efficiently partitioned the set of servers into null and write quora. We now describe the quorum completion algorithm in detail.

There are several operations to describe: *read*, *write*, *propagated write*, *begin transaction*, *end transaction*, *recovery*, *beaconing* and *election*. Many of these are quite straightforward. Before examining the individual operations, assume that we have a copy of the database on each server, and along with the database, we store a database version number and an up-to-date flag.

Handling a **read operation** is quite simple. If a server is a member of a write quorum, and its database is up-to-date, then the read request is simply handled using the local database. If the server is a member of a write quorum, but it does not contain an up-to-date database, then the server returns an error code indicating that the answer should be available shortly. If the server is not a member of a write quorum, it returns an error code indicating that it can not handle the request.

Handling a **write operation** is quite similar. Again, the server must be a member of a write quorum, and must have an up-to-date database. In addition, the server must be the synchronization site of the quorum in order to handle a write request; if a write request arrives at a non-synchronization site, a distinctive error code is returned, notifying the client of the actual sychronization site’s identity. To handle a write request, the server locks the appropriate piece of data (possibly the entire database), makes the change in recoverable storage (in case the operation is aborted), and propagates the update to the other servers in the quorum by invoking their *propagated write* routine. If one of these propagated writes fails, the synchronization site must invoke the recovery procedure, and then continue to try to propagate the write to the remaining servers in the quorum. If the server is unable to propagate the request to a write quorum, the write operation (and thus the entire transaction) fails.

A **propagated write operation** is handled similarly to a write operation, except for the phase where the write operation is propagated to the other servers. The write operation changes the appropriate data in recoverable storage, and returns. The changes will be made permanent at commit time.

The **begin transaction operation** sets either a read or a write lock on the entire database. A write transaction sets its write lock on all servers, while a read transaction needs to set a read lock only on the server that will be handling the read transaction. If desired, either read or write locks can be set at a finer granularity than the entire database. However, because write database operations must update the database version number, concurrent write transactions are not permitted. Note that the use of more clever database version number formats might allow more simultaneous write operations.The **end transaction operation** is quite simple, as distributed commit operations go. If one is committing a read-only transaction, the commit operation is trivial: it just ends the transaction. The rest of this description concerns the commitment of a write transaction. Quorum completion uses a two-phase commit protocol. In the first phase, all servers are notified that they have the complete new database, and are asked to acknowledge receipt of this information. Piggybacked on this call is an update of the database version number (in particular, the commit count must be incremented). Before sending this acknowledgement, the servers must ensure that they have written the new database to stable storage; even after a crash, they will be expected to restore this version of the database. This is the first forced log-write of the transaction. No locks are released during this first phase, since one may not release any data before the transaction completes at a write quorum.

After a write quorum responds positively to this first phase, the second phase begins. This phase consists of the synchronization site notifying the servers in the quorum to release their locks. This phase requires no log forces, since locks need not be stored in stable storage.

Every synchronization site performs periodic **beaconing operations**. Periodically, a beacon message is sent to all of the other members of the quorum, essentially telling the quorum members that the quorum still exists until a certain time. This message also tells the members what type of quorum they are a member of (null or write), and thus what type of operations they are permitted to perform. Beaconing calls can fail, just as any others, in which case the synchronization site invokes the recovery procedure.

The **member change operation** is one of the more complex operations of quorum completion. This operation is initiated whenever a server is the synchronization site and has failed to communicate with a member of its quorum. During a member change operation, a site attempts to reestablish a quorum with a new set of members. If members are being deleted, the synchronization site's main responsibility is to ensure that no further changes to the database are performed until it is certain that the recently-lost member has not received a beacon in BIG seconds. At this point, the synchronization site can continue operation with a reduced quorum. If members are being added, no delay is necessary, but the synchronization site is responsible for ensuring an up-to-date database is sent to the new server. In neither case is a new synchronization site elected.

If the synchronization site is creating a new write quorum, after having not been a member of a write quorum for any period of time, it must compute a new quorum creation time, for use in writing database version numbers. After creating this quorum, it contacts all sites within the new write quorum and locates the most recent version of the database, ordering the databases by version number (comparing quorum creation time stamps first, and write operation commit counts after that). The new sychronization site fetches this database and declares it to be the new current database. It labels this database with a new version number using the quorum's new creation time as the first part of the version number, and any constant for the second.

If a non-synchronization site loses contact with its synchronization site, it invokes an **election operation**, after aborting any write transaction in progress. In this mode, the server is not a member of a quorum, and executes the procedure described above in the section entitled "Election Protocol and Quorum Definition." When this procedure is complete, a new synchronization site will have been selected, and a new quorum defined. The new synchronization site, if it has defined a write quorum, contacts all servers in the quorum, locates the most up-to-date database, fetches that database and distributes it to all of the other servers in the quorum.

The **local recovery operation** is performed every time that a server recovers from a crash. This operation consists of replaying the log so as to ensure that the database contained on the server is the lastest one that committed on that particular server.

The main **recovery operation** is perfomed by a newly-elected synchronization site when it first is elected by a write quorum. When recovering, the new synchronization site first polls all of the machines in the write quorum, determining the best database available in the write quorum (we'll see below that this guarantees that the new synchronization site will see the last database actually committed by a previous synchronization site). Then, the new synchronization site retrieves this database from the appropriate server, relabels it with a new database version number containing this newest synchronization site's epoch value, and then propagates this database to all of the servers in its write quorum. As we will see below, a crash during recovery is of little import as long as it leaves either the old or new database available at every site involved in the recovery operation.

Quorum Completion Example

This section traces the basic operation of the system through some common operations, as well as recovery.

Servers in this system can be in one of three states with respect to a newly-received operation. A server may not be a member of a sufficiently large quorum to handle an operation, in which case it returns a distinctive error code to this effect. A server may be a member of a write quorum, but the system might be recovering and this server might not yet contain an up-to-date database. In this case, a server might choose to wait and retry the operation; it should get an up-to-date database in a bounded time period. A server may be a member of a write quorum, and have an up-to-date database, in which case the server may handle read operations. Finally, a server may be the synchronization site for a write quorum, and have an up-to-date database, in which case it may handle write operations.

In more detail, let us assume we have just started up N servers with empty databases. At this point, all of the correctly-functioning servers start periodically sending beacons to all other servers; most stop sending beacons as soon as they hear from a site lower in the canonical ordering (e.g. with a lower network address). A newly-booted server must wait BIG seconds before responding to beacon messages to guarantee that a previous incarnation of the server did not vote for another synchronization site; afterwards, the server may vote for the lowest numbered site from which it has received a beacon.

As soon as a sync site has received enough votes to declare a quorum, it does so, declaring in its periodic beacons the type of quorum (null or write) that voted for it. After declaring a write quorum, a server contacts the servers in its quorum and determines the latest available version of the database. It updates its local copy of the database from this version, updates the database's version number, and distributes this database to the other servers in the quorum. As soon as the sync site has sent a database to a server, it marks that server as up-to-date. As long as the server remains in this quorum, it will continue to receive updates keeping it up-to-date.

Any server containing an up-to-date database in a write quorum may handle read requests from its database locally. The sync site is the only site that can handle write requests, and then only when it has an up-to-date local database.

When a sync site receives an update request, it propagates the request to all of the other servers in the write quorum. When a sync site receives a commit request, it sends this request to all of the other servers. If, due to site or network failures, the server is no longer a member of a write quorum, it must fail the write operation by returning an error response to its caller. The caller will not be able to determine whether the operation really succeeded or not until after a new write quorum is elected.

Finally, the synchronization site sends unlock messages to all of the servers in the quorum. Any server that misses this message will keep the appropriate part of the database locked until it realizes, BIG seconds later, that it is no longer a member of a write quorum, at which point its entire database becomes inaccessible.

Note that as soon as a write quorum is elected, the correct version of the database is determined to be the latest version anywhere in the write quorum, and any other database in the system, including possibly more recent data, is discarded. However, any discarded data could not be part of a committed transaction, since such a transaction would have to have succeeded in a write quorum, in which case it would have appeared in one of the servers of the initial write quorum, and would thus have been a candidate database in the initial determination of the most up-to-date database for the newly-created write quorum.

Local Atomic Commit

So far, we haven't talked about how commit is handled locally on each server. One has the usual choice of single site commit protocols:

Logging. One may use logging to allow uncommitted operations to be backed out, or partially completed committed operations to be completed. We will assume the use of logging in the remainder of this paper.

Careful update. One may use careful update to ensure that a crash during the commit operation leaves the database in a consistent state.

Correctness Sketch

This section sketches out proofs of several useful properties of the quorum completion algorithm. There are several points we will demonstrate in this section. Some are quite straightforward to demonstrate, others are quite subtle.

1. Database version numbers always increase with time.

2. No two databases ever have the same database version number unless they have identical contents.

3. Transactions are executed in a serializable manner.

4. After a crash, the next synchronization site gives out the most recently-committed data.

5. When no crash has occurred, all reads at time T read the data committed at the last transaction committed before T.

6. If a read is completes at time T, it will not see any data not committed by time T.

Formal proofs of the above are out of the question, but here are informal sketches of the above lemmas.

First, no two servers are synchronization sites for a write quorum simultaneously. Furthermore, if server A is synchronization site at T1, and B is synchronization site at T2, then T2-T1 >= SKEW, since A has votes of at least K sites until E1 and none of these sites can vote for B until E1 + SKEW, so B can not get a majority until T2 > E1 + SKEW. Because synchronization sites label their databases with the time at which they became synchronization sites, and since SKEW represents the maximum clock skew in the system, it is clear that if B becomes synchronization site after A, then B will label its database with a larger version number than did A. This shows property 1.

If SKEW is greater than the clock resolution, then the above also shows that no two databases ever have the same database version number if they are written by different synchronization sites, since they have different epoch fields. This demonstrates the difficult part of property 2. For those transactions executed without any change in the synchronization site, one simply notes that every write transaction increments the commit count field in the database version number. Thus we've shown property 2.

Transaction are executed in a serializable manner by holding the required type of lock over the transaction from the time that the specified piece of data is first referenced, until the time that the transaction commits at a write quorum (for write transactions) or at a single site (for read transactions).

After a crash, the new synchronization site does not give out any data until it has polled at least a write quorum's worth of servers, and determined the latest database version available from any of those machines. If a transaction has completed in the previous write quorum, then that transaction wil have modified at least a majority of the servers' databases, and, when the new synchronization site contacts a majority of the existing servers, it is guaranteed to contact at least one server that participated in the last synchronization site's commit operation successfully. The new synchronization site will then take this latest database, label it with its own epoch, and propagate it to the new write quorum.

On the other hand, if the last transaction never reached the point where a commit was issued, then none of the servers participating in the previous synchronization site's transaction will have performed a local commit operation, and all of these servers will have aborted the transaction when they recovered their database after losing contact with the previous synchronization site.

There remains another state, wherein a commit was issued for a particular transaction, but the synchronization site did not manage to contact all of the machines within a write quorum before it crashed. In this case, the decision whether to commit or abort the transaction is not made until the next synchronization site is elected by a write quorum. In this case, after being elected by a write quorum, the new synchronization site chooses to commit the last transaction if the new write quorum contains at least one server from the last synchronization site's write quorum that managed a local commit, and aborts the transaction if none of the servers in the new quorum contain the committed data.

Finally note that it is possible for a crash to occur during the new synchronization site's recovery procedure. In this case, aside from servers containing the previous transaction's database labeled with the previous synchronization site's database version number, a number of servers may also contain the exact same database labeled with the latest synchronization site's database version for it. However, reexecuting the recovery procedure for a new synchronization site in this environment does not cause any problems, since a majority of the servers contain the correct database, and all of the correct (committed) databases are labeled with larger database version numbers than all of the old databases. This demonstrates property 4, and property 6 in the case where a crash occurs before the trasnaction is committed.

Showing property 6 when no crash occurs is quite easy, since aborts are never performed by a server when no crash occurs. Since all updated data is locked until the transaction commits at at least a write quorum, there is no opportunity to read uncommitted data.

Finally, when a read is issued at time T, while a write transaction commits at time $T_c < T$, then this read reads the committed data, since either the previous write transaction either completed at all sites in the quorum, in which case the read issued at T must go to a server that committed the trasnaction at T_c, or the read was sent to a site that never completed the previous write transaction. In this last case, the synchronization site delays the commit response until the missing server has gone for BIG seconds without hearing from the synchronization site. At this time, the synchronization site knows that the missing site has not heard from a synchronization site in at least SMALL seconds, and is thus not willing to give out any data to read or write transactions. Thus a read issued at $T > T_c$ will not read any data from a site that failed to participate in the last transaction's commit protocol. Thus, we have shown property 5: no uncommitted data can be read by a transaction.

Intermission

At this point, we have described a generic system for performing read and write operations on a replicated database. We next describe the actual procedural interface to the ubik system. This is the interfaced used by programmers writing an application using the reliable, replicated disk abstraction.

The Ubik Procedural Interface

See the companion file ubik.vdoc for the ubik programmer’s interface.

Logging in Ubik

In the ubik system, we propose to use new-value only logging. Data changes will be kept in virtual memory until commit time.

Some attention must be paid to recording truncate operations, since they do not result in changes to the virtual memory state, but this should be a relatively minor implementation complexity.