from: http://web.mit.edu/6.033/2005/wwwdocs/quorum_note.html

Quorum Consensus

This note describes a quorum consensus algorithm that you may find useful in DP2. The algorithm was first described in:

```
    H. Attiya, A. Bar-Noy, and D. Dolev. "Sharing memory robustly in
message-passing systems." Journal of the ACM, 42(1):124-142, Jan. 1995
```

There are N replicas in a group. A replica group stores a collection of items; for example it might store files or file pages. Each item has an identifier and a state.

Each replica stores the state for each item, plus an extra piece of information: a version number. The version number is a pair

v1.val v2.val or (v1.val = v2.val and v1.cid v2.cid)

The idea is that if different replicas store different version numbers for an item, the state associated with a larger version number is more recent than the state associated with a smaller version number.

Reads go to a read quorum of size R and writes go to a write quorum of size W. Furthermore, we require that R+W N, i.e., read quorums always intersect with write quorums. This will ensure that read results always reflect the result of the most recent write (because the read quorum will include at least one replica that was involved in the most recent write).

For example, consider a group of 3 replicas. Then we have the following possibilities:

- R=3 and W=1. This improves performance for writes at the expense of reads, which is probably a bad idea since generally reads are more common than writes. In addition, this choice of quorums is bad because a write might happen at a single replica that then fails. If that replica were to lose its state, the outcome of the write would be lost. So generally we would like to have W>1.
- R=1 and W=3. This works very well for reads which is generally good since reads are common. But it is undesirable for writes because if one of the replicas is down or inaccessible, a write cannot complete until that replica recovers.
- R=2 and W=2. This choice is a good compromise compared to the R=1 and W=3 choice, since it increases the cost of reads in return for providing reasonable availability for writes.

How it works

Each client machine runs a client-side proxy that carries out the replication protocols. The proxy provides two operations that can be called by user code on the client machine:

write (x, s)
s <- read (x)

Here x is an identifier that indicates the item of interest (e.g., the name of a file). The write operation takes the new state that is intended to be stored for that item, and the read operation returns the current state of the item.

The write operation

A write requires either one or two phases. These phases require communication with a quorum, and the phase is complete when a quorum of responses has arrived at the client side.

1. The proxy sends a read request for the version number to all the replicas, and waits for replies from a read quorum. Then it takes the biggest version number , where c2 is its own client-id.
2. The proxy sends a write request containing the state and new version number to all the replicas and waits to receive acknowledgements from a write quorum. At that point the write operation is complete and the proxy can return to the user code.

If there are concurrent writers, the one choosing the largest version number will prevail. The protocol doesn't provide any synchronization; instead this would be done outside, e.g., by using locks.

The first phase can be avoided if the client knows what version number to provide without having to read it. This will be true if there is just one writer, i.e., one client modifies the item and all the other clients just read it. It can also be true if there is some way that writers coordinate to find out the proper version number without reading it.

Note that if the first phase is needed, it doesn't do any good to choose a small write quorum and a larger read quorum since completing a write will involve communicating with a read quorum.

The Read Operation

To perform a read requires either one or two phases.

1. The proxy sends a request to read the version number and state to all the replicas and waits for replies from a read quorum. If all version numbers it receives are the same, it returns the value to the user code and the protocol is complete.
2. Otherwise the proxy sends the largest version number and the associated value to all the replicas, and waits for responses from a write quorum. Once the quorum of acknowledgements has arrived, the proxy can return the value to the using code

The second phase of read is called the "write-back" phase. It is needed when the read is concurrent with a write, in order to ensure condition (b) below. It is also useful to handle the case of a client that fails in the middle of a write, since it effectively completes that write.

Usually, the write-back phase will not be needed. The reason is that write requests (in phase 2 of the write operation) are sent to all replicas and in the absence of failures all replicas will record the results of that write. Therefore, all the responses in phase 1 of the read operation will return the same version number.

Protocol at the replicas

In response to a read request for some item, the replica returns the value with its version number (or just the version number if this is what the client wants).

In response to a write request, if the new version number is greater than what the replica has stored, the replica overwrites with the new version number and new state. Otherwise it doesn't overwrite but instead just keeps what it has. In either case it returns an acknowledgement.

Discussion

This protocol provides atomicity.

1. It guarantees that a read will see a state at least as recent as that produced by the most recently completed write that completed before the read started.
2. It also guarantees that if some reader sees the results of a particular write, than any reader that starts after that reader finishes will also see a result at least that recent.

Another name for this condition is linearizability.

Recovery

This protocol has the property that when a replica fails (or is inaccessible because of a network problem), operations continue to work properly. If the needed quorum of replicas is available, operations complete; otherwise their execution will be delayed until the failure or inaccessibility is over.

Some care is required when a replica recovers: for the protocol to work properly it must be the case that a replica responds to requests only after it knows all modifications that it acknowledged before the failure.

One way to satisfy this requirement is to write the new state for a modified item to disk before sending the acknowledgement. Then on recovery the replica can continue to provide service immediately, provided its disk has not been trashed.

However, if there is a problem with the disk, or if the replica sent acknowledgements before writing to disk, then the replica must obtain all modifications it may have lost. It can do this by reading all items from the other replicas. This read doesn't require a write-back, since the next read done by a client will do the write-back.

The reason the replica must recover its state before responding to client requests is that otherwise it would count in a quorum but the information it is sending might be stale.

For example consider a three-replica system in which both read and write quorums consist of two replicas.

Now consider the following scenario:

- Client C1 writes
- Replica R1 fails and forgets this value (i.e., at R1, x reverts to).
- Replica R1 recovers and immediately begins responding to requests without recovering any lost state.

At this point a read might complete with a quorum consisting of replicas R1 and R3, providing the user with the state s0, which is wrong.

Note that if the read had happened while R1 was failed, there wouldn't be a problem because in this case the read would require responses from R2 and R3, and therefore it would return the value s1. Note also that the problem will not occur if R1 recovers its state before responding to client requests.

The exact mechanisms whereby a replica detects that its state is bad, and the way it determines exactly which items it needs to read from other replicas, are beyond the scope of this note.