Quorum Consensus Replication[1]

These are companion notes that should be complemented with the definitions, pseudo-code and examples that are included in the teaching slides (available at the course's web site).

Read and Write Thresholds

In the quorum consensus (QC) algorithm, we assign a non-negative weight to each copy of a register *x*. We then define a read threshold RT and write threshold WT for x, such that both 2WT and (RT + WT) are greater than the total weight of all copies of *x*. A read (or write) quorum of *x* is any set of copies of *x* with a weight of at least *RT* (or *WT*). The key observation is that each write quorum of *x* has at least one copy in common with every read quorum and every write quorum of *x*.

Basic Algorithm

In QC, the client front-end is responsible for translating Reads and Writes on data items into Reads and Writes on copies. A client front-end translates each *Write(x)* into a set of *Writes* on each copy of some write quorum of *x*. It translates each *Read(x)* into a set of *Reads* on each copy of some read quorum of *x*, and it returns to the application the most up-to-date copy that it read.

To help the client front-end figure out which copy is most up-to-date, we tag each copy with a version number, which is initially 0, and a *client-id* field (assuming each client is identified with a unique number). When the client front-end processes *Write(x)*, it determines the maximum version number of any copy it is about to write, adds one to it, and tags all of the versions that it writes with that version number and its *client-id*. Clearly, this requires reading all of the copies in the write quorum before writing any of them.

The version numbers measure how up-to-date each copy is. Each *Read* of a copy returns its version number along with its data value. The client front-end always selects a copy in the read quorum with the largest version number (there may be more than one such copy but they will all have the same value). If two different copies have the same version number but different *client-ids*, the client front-end selects the one with highest *client-id*.

The purpose of quorums is to ensure that *Reads* and *Writes* that access the same data item also access at least one copy of that data item in common. Even if some copies are down and are therefore unavailable to *Reads* and *Writes*, as long as there are enough copies around to get a read quorum and write quorum, client front-end can still continue to execute.

[1] Adapted from "Concurrency Control and Recovery in Database Systems" (Copyright © 1987 by Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman) and "Sharing memory robustly in message-passing systems" (in JACM 42, 1, Jan. 1995, 124–142, by Attiya, H., Bar-Noy, A. and Dolev, D.)

For a given object, the QC algorithm ensures that, when a client reads some data item *x* when no write to *x* is concurrently happening, the returned value will correspond to the value of the most recent write to *x*. This is because the read operation will receive values from a read quorum of *x*, while the previous write to *x* updated a write quorum of x. Since every read and write quorum have a nonempty intersection, the read will at least receive one value of the previous write (and possibly some outdated values from older writes). As the most recent write has a bigger version number, the read will choose that value.

Note that, if there is one or more writes to *x* taking place concurrently with the read, the return value can either be the value of the most recent completed write or the value of one of the concurrent writes. This is a problem, as a front-end that executes a sequence of reads concurrently with one or more writes can see an inconsistent sequence of values (e.g. first read returns the fresh value of the ongoing write; then second read receives responses from outdated replicas, thus returning the older value).

ABD Variant

A variant to the QC algorithm that avoids the previous pathological case consists of adding a *writeback* phase to every read. In this *writeback* phase, the front-end that has just read a given value *v* associated with a tag *t* will send a *Write(v, t)* request to all replicas and wait for WT *acks*.

It is easy to show that this additional phase ensures that any subsequent read will not observe from the past of *v*. Hence, the pathological case that we previously described is prevented.

Discussion

A nice feature of QC is that recoveries of copies require no special treatment. A copy of *x* that was down and therefore missed some *Writes* will not have the largest version number in any write quorum of which it is a member. Thus, after it recovers, it will not be read until it has been written at least once. That is, client front-ends will automatically ignore its value until it has been brought up-to-date.

Unfortunately, QC has some not so nice features, too. Except in trivial cases, a client front-end must access multiple copies of each data item it wants to read. Even if there is a copy of the data item at the client's site (i.e., the client is a replica too), the client front-end still has to look elsewhere for other copies so it can build a read quorum. In many applications, clients read more data items than they write. Such applications may not perform well using QC. One might counter this argument by recommending that each read quorum of *x* contain only one copy of x. But then there can only be one write quorum for *x*, one that contains all copies of *x*. This would lead us to the write-all approach, which we found was unsatisfactory.

A second problem with QC is that it needs a large number of copies to tolerate a given number of site failures. For example, suppose quorums are all majority sets. Then QC needs three copies to tolerate one failure, five copies to tolerate two failures, and so forth. In particular, two copies are no help at all. With two copies QC can't even tolerate one failure.

A third problem with quorum consensus is that all copies of each data item must be known in advance. A known copy of *x* can recover, but a new copy of *x* cannot be created because it could alter the definition of *x*'s quorums. In principle, one can change the weights of the sites (and thereby the definition of quorums) while the replicated system is running, but this requires special synchronization.