

# CS138 Programming Assignment 3: Quorum

---

*Assignment Out:* February 26, 2008  
*Assignment Due:* March 13, 2008 (11:59 pm)

---

## 1 Introduction

Now that you have a replicated system that keeps all of its members up to date, it's time to think about data consistency. If several clients simultaneously send conflicting updates to two or more different servers, or attempt to add data with the same key, your database should take it in stride and provide a consistent view of the data.

## 2 The Assignment

You're going to add a feature to SDDB – the ability to make “quorum” updates. In a quorum update, instead of updating the database on a node directly, it sends a “quorum request” to the other nodes. When a majority of nodes in the system have agreed to update the database, an authoritative update is propagated and every node updates its database at that time.

Your system should return only data that has been successfully integrated into the entire database. If a quorum is in progress, the data that is being decided upon should not be visible in the database.

Note that it may sometimes be impossible to commit an update - for example, if there are only two servers in the system and each is given a conflicting version of the same update, neither will be able to acquire a majority. What is important is not that each change be accepted by the system, but that it maintains a consistent state across all servers. Keep in mind that if a new update is requested with no conflict after the above example takes place, it should pass the quorum and commit the update.

Finally, our usage of ‘update’ is ambiguous, and may refer to either an ‘add’ (adding a new key to the database) or an update (modifying an existing key in the database).

## 3 The Design

For this assignment, you will be modifying the structure of your tree that you set up in SDDB. In SDDB, each node contacted the root directly in order to propagate updates. You should now

change this to a more mature system where each node propagates messages directly to its parent and/or its children. This allows for a more distributed architecture and reduces a single point of failure. This will become especially important in the next project, where you will make your database more fault-tolerant.

In order to implement the quorum semantics, your nodes will have to be prepared to create quorums and participate in quorums, and also interact with the newly-added election numbers. The following sections explain the semantics of each.

### 3.1 Election Numbers

While participating in a quorum, you will have to place your vote. The mechanism that provides these semantics are election numbers. We have updated the `DBData` class to include this information. Each key in the database now stores both the datum and an "election number" representing the version of the datum. You should interact with it in the following way:

1. A node will vote 'yes' on a quorum if the relevant datum has the same election number as the quorum election number. Otherwise it will vote 'no'.
2. If a node votes 'yes' on some quorum, it should increment the relevant datum's election number.
3. If a node votes 'no' on some quorum, it should not change the datum's election number.

You should think about why this ensures that when two quorums occur concurrently, each node will vote 'yes' on at most one of them. Come see the TAs on hours if you are unsure. The helpsession will present various examples to illustrate this mechanism.

### 3.2 Creating a Quorum

There are multiple steps to creating a quorum. When a node wants to perform a quorum update (meaning the `quorum_put` method was invoked), it will do the following:

1. Create a quorum request.
2. Send it to all of its children and the parent.
3. Listen for responses (votes) from all the nodes and maintain a tally for the vote.
4. If more than half the nodes vote 'yes', it should perform an authoritative update (similar to how it was done in SDDB), which will commit the change throughout the tree. Finally, it should inform the client who initiated the quorum by invoking the `SuccessListener.success()` method.

5. If half or more of the nodes vote ‘no’, it should inform the client who initiated the quorum by invoking the `SuccessListener.failure()` method.

### 3.3 Participating in a Quorum

When a node receives a request to perform a quorum update from another node, it must contact the creator of the quorum and place its vote (either ‘I agree to this update’ or ‘I disagree with this update’). The ‘Election Numbers’ section above explains how a node decides whether to vote ‘yes’ or ‘no’ on a quorum. Informally, the node votes ‘yes’ if it hasn’t already voted on a conflicting quorum.

Regardless of what vote is placed, the node will propagate the request to its parent and children. However, it is important that a node will not vote in the same quorum twice. There are multiple ways to ensure this (for example, not sending a quorum back to the node that sent it to you, or keeping track of all quorums the node has participated in), and the decision of which method to use is up to you.

## 4 Requirements

- **Modify SDDB** - You should clone your SDDB implementation and modify it such that it adheres to the new `put` and `get` interfaces. Essentially, the changes merely change the signatures to put in and get out `Serializable` rather than exposing the `DBData` class to the client (as it should not need to know about this). This is not directly related to this assignment, but makes the design much cleaner. *DO THIS FIRST.*
- **Implement the Interface** - You must implement the new `DDBQuorum` interface which is provided. The method `quorum_put` should send a quorum request to decide whether to commit the put. The original `put` method should still perform an authoritative put under the assumption that there will be no contradiction.
- **Multiple Puts** - If two or more quorum puts with the same key are simultaneously initiated by different nodes, at most one of them should be committed by any server. All reads thereafter from any server should return the same data from the key.
- **Commit Responses** - When a quorum is successful and the given update is committed, the client should be informed via the given `SuccessListener`’s `success` method. Similarly, when it is determined that the update cannot be committed, the `failure` method should be invoked.

This handout is purposefully vague on implementation details, beyond the bare minimum to explain the assignment. Think the project through before diving in, and come talk to the TAs if you have any questions or want to discuss your ideas. There are several ways to do the assignment, some better than others.

## 5 Code Exchange

We have provided you with a new interface to implement called `DDBQuorum`. It provides a method `quorum_put` which should create a quorum to decide whether to commit the given datum to the database. `quorum_put` has an additional argument, the `SuccessListener`, which is a callback to the user when the result of the quorum is determined. If the quorum is successful, the `success` method is invoked, and if the quorum fails, the `failure` method is invoked. We have also provided you with a new version of `DBData` which has election numbers.

You should clone your `sddb` directory source directory, and then copy the code from `/course/cs138/asgn/quorum` into it, overwriting conflicting files.

As usual, you should write a README to document notable bugs and features, as well as other information that you think the TAs should know.

To hand in quorum, run the following script from the directory containing your source code:

```
/course/cs138/bin/cs138_handin quorum
```

## 6 Extra Credit

- **Weighted Quorum** - Instead of seeking a numerical majority of active servers for your quorum, you use a weighted majority. Each server has a weight between 0 and 1, such that the sum of all servers' weights is 1. To approve an add or update, only servers whose weights add up to .5 or more are needed. When a new server joins the system, it starts with a weight of 0 and asks to split the weighting of the first other server it finds.
- **Deletion Quorums** - Implement deletions and extend your quorums to apply to these as well.

Feel free to create your own extra credit projects, but be sure to run your idea past a TA beforehand if you want to be sure to get points for your additional work.