# CSC 464, Assignment 5

This assignment is worth *six tokens* and
is due on *5 December at 11:59 pm*

The last incarnation of the bulletin board server will now include a distributed replication algorithm (plus a new application protocol for it). The new server can be based on the solution for either Assignment 3 or Assignment 4 at your discretion. If starting from the Assignment 3 solution however you must implement the full configuration file manipulation as described in Assignment 4 and further in this handout.

Note that the old application protocol as well as the manipulation of the bulletin board file are unchanged.

## 1 New Configuration

The configuration file for the new server contains two more definitions, as follows:

`RPORT` *rp*
`PEERS` $h_1$ $h_2$ $h_3$ $\cdots$

where *rp* is a string representing a positive number, and $h_i$, $i \geq 1$, are host names of IP addresses. The port *rp* is the port numbers on which the replica server (to be discussed below) listen to incoming clients. The `PEERS` line specifies the host names of the replica servers participating in the replica mechanism (see once more the next section).

Like before a missing line causes the respective variable to take a default value which is 9001 for `RPORT` and an empty list of peers for `PEERS`.

## 2 Replication Algorithm

Distributed algorithms are not something terribly different from the algorithms you already know; the difference consists in the fact that parts of the algorithms run on different machines, and so you need to implement network communication to put all the parts together. In addition, the unreliability and unpredictability of such a communication often introduce issues that need to be considered (by supplementary algorithmic steps). You are now ready to implement such a distributed algorithm.

Our particular distributed algorithm is a replicated database management system. You have a database to manage (the bulletin board file will do for this assignment); the database is so important that it is kept *replicated* on multiple servers running on multiple machines. There has to be a good probability that the content of the database is kept in sync on these servers at any given time.

This kind of servers, called *replica servers*, are multiprotocol: they use one protocol for the normal client-server interaction, and another for inter-server synchronization. This is also a good example of programs that are at the same time servers and clients.

The normal database commands are processed following a normal algorithm (the one implemented already for the original server). The synchronization between servers is initiated by the receipt of a `WRITE` or `REPLACE` command from some client, and is accomplished by using the *two-phase commit protocol*. This protocol is widely used in applications where data consistency is critical (see e.g., *Distributed Operating Systems and Algorithms*, by R. Chow and T. Johnson, Addison-Wesley, 1997). Specifically, the protocol ensures

as much as possible that data stored at each replica server are identical, even if this causes some data to be lost.

When using the two-phase commit algorithm, the server that received the `WRITE` or `REPLACE` command becomes the master (or coordinator), and the others become salves (or participants). In passing, note that the master becomes a client to all of the slaves.

As the name of the algorithm implies, it consists in two phases:

**Precommit phase** The master broadcasts to all the other servers (which reside on the hosts whose names are given in the configuration file) a "precommit" message. The servers which are available for synchronization acknowledge positively the message; the servers which are not ready (because of, e.g., a system failure) will send back a negative acknowledgment.

The master blocks (within some timeout) until it receives all the acknowledgments. If no acknowledgment comes until the master times out, a negative acknowledgment is assumed.

If there exists a negative acknowledgment, the master sends an "abort" message to the slaves and aborts the writing altogether (sending an `ERROR` message to the respective client). The slaves will abandon the whole thing if they receive a negative acknowledgment.

If on the other hand all the acknowledgments are positive, the second phase of the protocol is initiated.

**Commit phase** In the second phase, the master sends a "commit" message to all the slaves, followed by the data necessary for the current operation to proceed (namely the operation to be performed and its arguments, which you can send in the commit message itself or in a separate message).

Each slave then performs the corresponding operation. Each slave sends a positive acknowledgment back to the master if the (local) operation completed successfully; or a negative acknowledgment otherwise. The master performs the requested operation if and only if it has received positive acknowledgments from all the slaves. Upon the success of the local operation a "success" message is sent to all the slaves.

If there has been a negative acknowledgment from at least one slave, or if the master has been unsuccessful, then the master broadcasts a "not successful" message. Upon receipt of such a message, a slave "undoes" the writing process, in the sense that the effect of the writing is canceled.

To summarize, the two-phase commit protocol can be represented by two finite automata (one for the master and another for the slave), which are shown in Figure 1 (the conditions that enable the corresponding transitions are written in italics—and in blue if you read this document in colour, while the actions associated to the transitions are written in plain text).

Note that you are responsible for designing the application protocol for the two-phase commit algorithm. Give some thought to this design, preferably before starting coding so that your protocol is robust and unambiguous.

# 3 What to submit

Submit the sources for your server plus appropriate documentation. The strongly preferred documentation format is *plain text*, though PDF is also acceptable. Also provide a makefile whose default target produces an executable named `rbbserv` and residing in the root directory of your submission.

Make sure that you provide an appropriate tests suite. Pay of course special attention to the features that have been added, but also make sure that the old features still work.

Put everything in a directory, zip it, and submit the result by email.

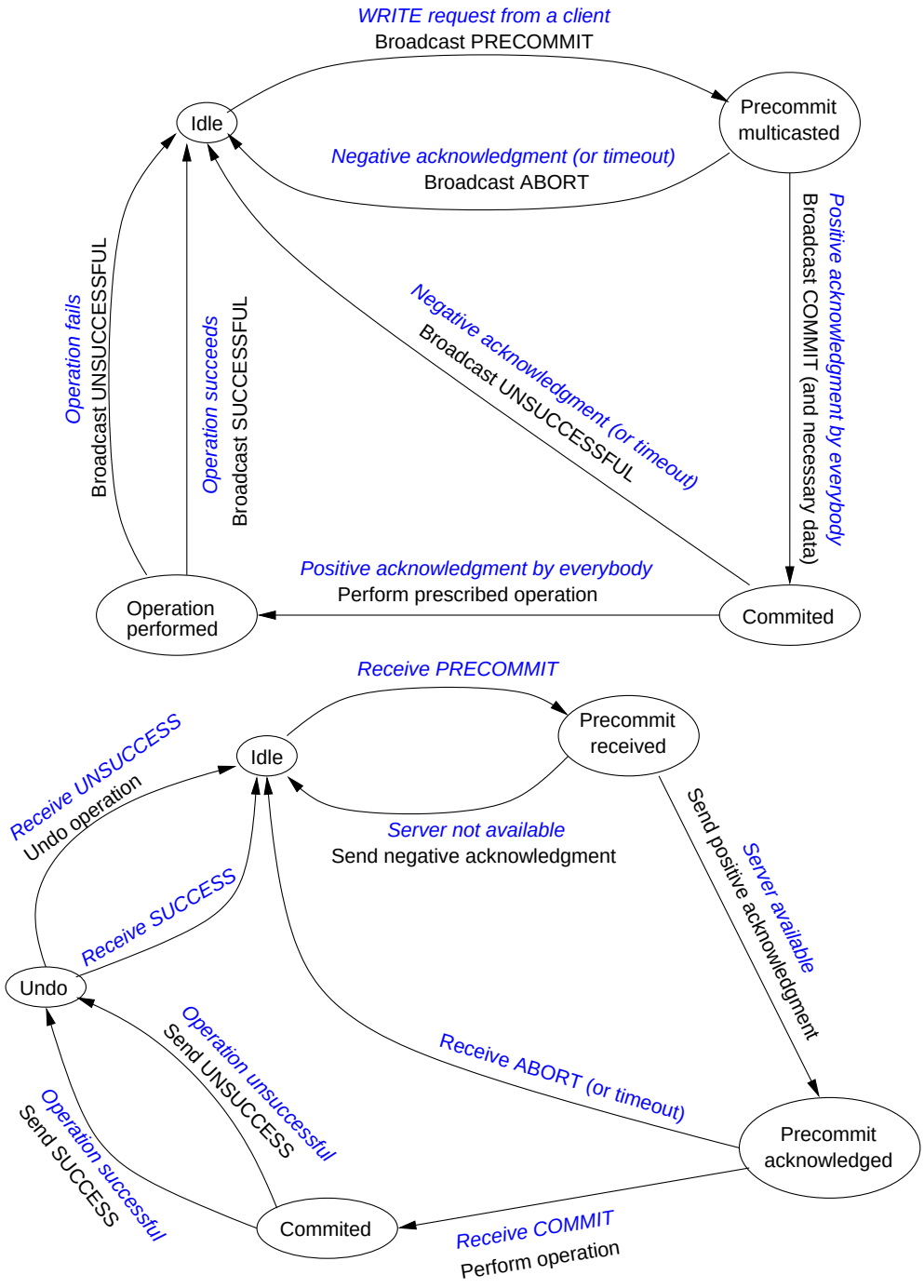


Figure 1: The master (above) and slave (below) in the two-phase commit protocol.