CS 464/564, Assignment 4

This assignment can be solved in teams of no more than *four students*,
is worth *eight tokens* and
is due on *18 April at 11:59 pm*

You now have an opportunity to build a relatively complex production ready, distributed application.

We deal with the same bulletin board server. The old application protocol as well as the manipulation of the bulletin board file are unchanged. You should start your work from some known good solution for Assignment 3 (your solution or my solution). If you want to start from your solution you should probably wait for your marks to make sure that your solution works reasonably well. I will not explicitly test the old behaviour, but any issue observed during testing will be penalized, no matter whether it comes from the old or the new code.

We first add the necessary features over the existing solution to make it a production server. These features are a complete startup (daemonization) process and a re-configuration mechanism. In addition, the bulletin board server will now include a distributed replication algorithm (plus a new application protocol for it).

1 New Configuration

The configuration file for the new server contains the following additional definitions (in addition to the existing ones):

`FOREGROUND` *fg*
`PDEBUG` *pd*
`RPORT` *rp*
`PEER` *h*`:`*p*

All these new definitions are optional. The Boolean flags *fg* and *pd* can be specified either as `0` or `1`, or as `false` or `true`, and default to `false` (or `0`). Setting *fg* to `true` causes the server to remain attached to the controlling tty, see Section 2. Setting *pd* to true enables debugging information for the replication algorithm, see Section 4.

The string *rp* represents a positive number and defaults to 9001. It represents the port number on which the replica server (to be discussed in Section 4) listens to incoming clients.

The string *h* is either a host name or an IP address in dotted decimal notation. The string *p* represents a port number (positive integer), so that the combination *h*`:`*p* defines a TCP/IP communication end point and specifies a peer participating in the replication mechanism (see once more Section 4). Multiple lines `PEER` are allowed in the configuration file, and all the communication end points thus defined should be added to the list of peers for the respective server. You can assume that the current server does not appear in its list of peers. If no `PEER` line is present then the list of peers for the current server is empty.

2 Startup

We attempt to construct a production grade server, so our server must behave like a civilized daemon i.e., detach from the terminal by default, close unnecessary descriptors, use a PID file (i.e., a lock file that also

stores the PID of the currently running server) to ensure that only one copy runs at any given time, and so on. It is your responsibility to implement all the required startup features as given in the textbook and lecture notes.

In particular, the server should move to the "safe" directory where it should look for (and create) any file which is not given using an absolute path, including but not limited to the PID file but excluding the configuration file and the bulletin board file (which should be opened as specified on the command line and in the configuration file, respectively). For the purpose of this assignment the safe directory must be the subdirectory `run` of the root directory of your submission, and must be created if it does not exist. Obviously this defies the whole purpose of a PID file, but you will still need to go through the motions and at the same time it will make testing (and marking) easier.

When the flag *fg* is set the server must remain in foreground and attached to the controlling tty.

Verbosity switches ("`-v`") are not required but are recommended (they already exist if you base your work on my solution, though you my want to add more debugging options).

During normal operation messages particular to the current server are sent to the `run/bbserv.log` log file. When *fg* is set, all these messages (and possibly more debug messages) are printed to the standard output stream.

3 Dynamic reconfiguration

If the server receives the `SIGHUP` signal then it restarts, in the sense that all the idle threads quit, and all the active threads close connection and quit immediately after the current request (if any) has been completed. Once the only running thread is the monitor thread (if it exists at all), the server reads again the configuration file, re-configures itself accordingly, $\mathbb{T}_{\text{incr}}$ client handling threads are preallocated (where $\mathbb{T}_{\text{incr}}$ is the possibly new value just read from the configuration file), and the server resumes normal operation.

The server also reacts to the signal `SIGQUIT`. When such a signal is received the server terminates gracefully, in the sense that it does the same thing as when receiving `SIGHUP` but instead of re-reading the configuration file and restarting it just terminates. When *fg* is set the same graceful termination should happen upon pressing Ctrl-C in the controlling tty (you will need to find out the corresponding signal by yourself).

The server should ignore all the signals not explicitly mentioned above.

4 Replication Algorithm

Distributed algorithms are not something terribly different from the algorithms you already know; the difference is the fact that parts of the algorithm 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 are given a database to manage (the bulletin board file for this assignment), which is kept *replicated* on multiple servers running on multiple machines (often referred to as *peers*). Consistency between the database copies must be maintained at all cost. This means that a request for modifying the database (`WRITE` or `REPLACE` in our case) must fail if it cannot be effected by all the peers.

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.

A client connects to one of the replica servers and its requests are processed following a normal algorithm (the one implemented already for the original server). In particular, a `READ` request is served by the respective server alone, without any server-to-server communication. 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 aims to ensure 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 slaves (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 of 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 the master fails to receive an acknowledgment from some slave within the prescribed timeout, then a negative acknowledgment from that slave 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 process if they receive an abort from the master. None of the copies of the bulletin board file are modified.

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 *unsuccessful* message. Upon receipt of such a message, a slave rolls back the writing process, in the sense that the effect of the writing is canceled.

The end result is that all copies of the bulletin board file are modified in the same way, either according to the operation requested by the client or not at all. The master then responds to the client with either a `3.0` message (if the operation was successful) or a `3.2` message (if the bulletin board file was unchanged).

The two-phase commit protocol can also be represented by two finite automata (one for the master and another for the slave) as shown in Figure 1 (the conditions that enable the corresponding transitions are written in italics and in blue, while the actions associated to the transitions are written in plain text). The start stare of both automata is "Idle".

Note that you are responsible for designing the application protocol for the two-phase commit algorithm. Give serious thought to this design, preferably before starting coding so that your protocol is robust and unambiguous. Note that the specification of the application protocol for the two-phase commit algorithm is a compulsory part of the documentation (see below).

When the *pd* flag is set all the messages exchanged over the network (between the server and the clients as well as during the two-phase commit synchronization) must be sent to the log file (or to the terminal if the server is attached).

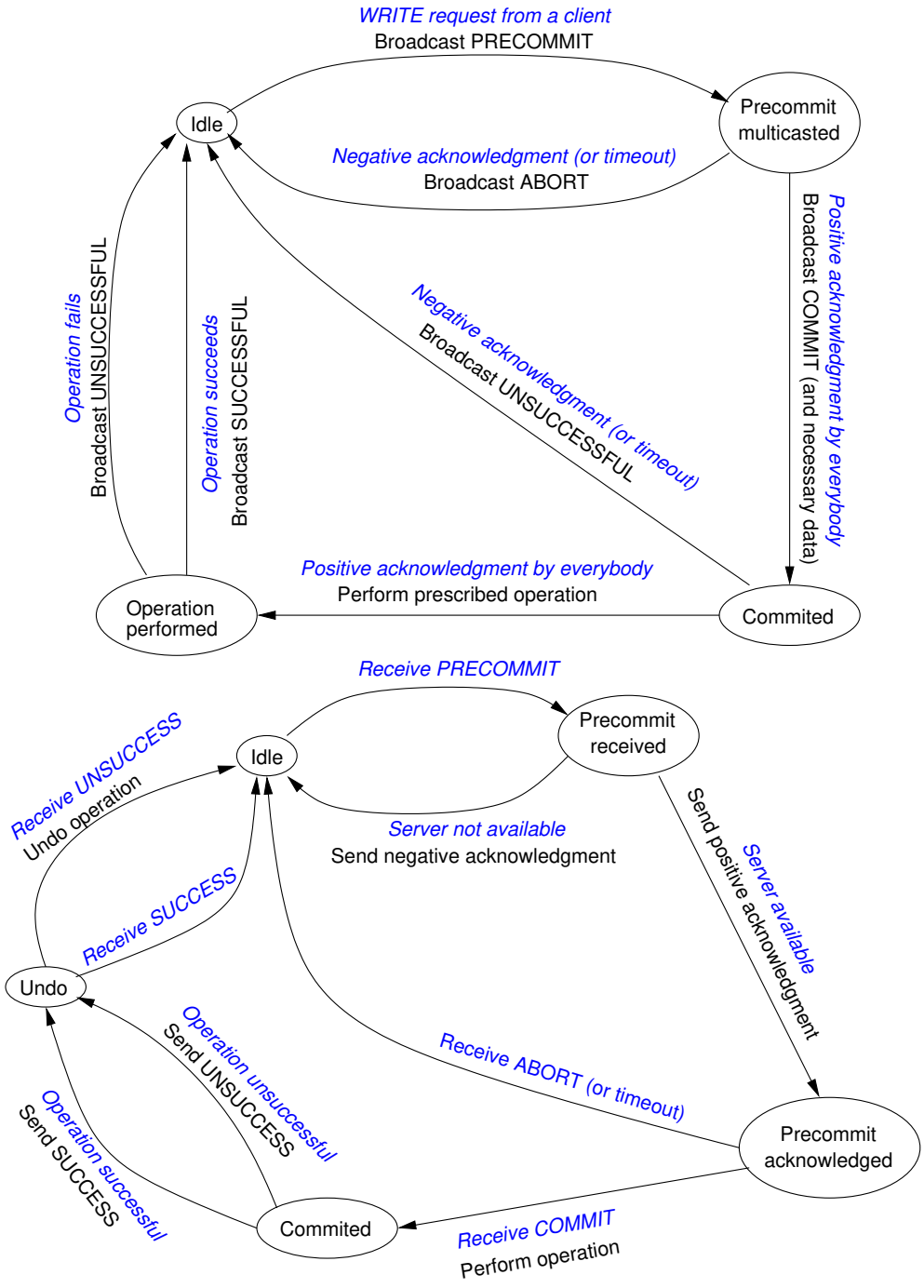

Figure 1: The master (above) and slave (below) in the two-phase commit protocol.

5 What to submit

Submit the source code for your server plus a makefile and appropriate documentation as specified on the course's Web site. The default target in your makefile should produce an executable named `rbbserv` and residing in the root directory of your submission.

Jut like in the previous assignment, the executable should not accept any command line argument except the optional name of an alternate configuration file. You are free to accept command line switches if you find them useful, but they should all be optional.

Your server should accept an empty list of peers, meaning that no `PEER` line is specified in the configuration file. When this happens the server should work just like it did in the previous assignment, operating on its own bulletin board file and not synchronizing that file with anybody else.

In addition to the usual requirements you *must* provide a full description of the application protocol for your two-phase commit implementation. This description must be provided in a separate, plain text file named `replproto.txt` residing in the root directory of your submission.

A full description of an application protocol must include a detailed and complete specification of all the message exchanged between the participants as well as the acceptable sequencing of these messages. An example of such a description is Section 2 of the Assignment 3 handout. Note however that the sequencing presented there is trivial and so implicit in the description. In the two-phase commit protocol description the sequencing should be presented in detail, possibly using finite automata as the ones shown in Figure 1.

Like before, you can test your server using the client you built for the previous assignment, or build your own, more user friendly client. You can also use `telnet` but be aware that telnet sends both a carriage return (`'\r'`, also displayed as `^M` by some editors) *and* the normal line feed (`'\n'`) at the end of each line. Note that I will primarily use `telnet` during marking.

Use the J118 machines for testing multiple replica servers. I recommend that you use 8000 + your user ID as port number for service to clients (like before), and 9000 + your user ID as port number for replica communication. The following two shell commands will print out these numbers when run on `linux.ubishops.ca` or the lab machines:

```
echo $((8000+$(id -u)))
echo $((9000+$(id -u)))
```