Bulletin Board Messages and Distributed Agreement: A CSC 590 Challenge

Stefan D. Bruda
stefan@bruda.ca

Read all *this handout carefully before you start working and make sure that you understand all the requirements. Indeed, all these requirements are reflected in the marking scheme.*

Contents

Introduction

This challenge consists of two phases. In the first phase you will construct a simple network server. The next phase will consists of convincing multiple such servers to work together. Read the document completely before starting any coding; everything in the document is part of the specification and it is your responsibility to ensure that you have implemented the whole server as specified.

The challenge is set up so that it gives you the opportunity to showcase your skills at system programming in a POSIX[1] environment. Indeed, you must implement the server as a UNIX service ("daemon") and you further must use the POSIX API provided by the UNIX standard C library. Therefore your server must be written in C or C++.

Throughout the handout we will refer to the following configuration parameters. How these parameters are obtained will be described in Section 3.1.

- *bp* is the port number for client-server communication (positive integer), see Section 1;
- *sp* is the port number for inter-server communication (positive integer), see Section 2;
- *bbfile* is the name of the bulletin board file that will be manipulated throughout this project (string), see Section 1;
- $\mathbb{T}_{\mathrm{max}}$ is the number of preallocated threads (positive integer), see Section 1.4;
- *peers* the list of peers participating in synchronization (possibly empty list of pairs host name–port number), see Section 2;
- *d* is a Boolean flag controlling the startup of the server, see Section 1.5;
- *D* is a Boolean flag controlling debugging facilities, see Sections 1.3 (last paragraph) and 3.2.

1 Phase 1: A Bulletin Board Server

Your first task is to construct a simple bulletin board server. The server accepts one-line messages from multiple clients, stores them in a local file, and serves them back on request. The name of the file is given by the parameter *bbfile*. Messages are identified upon storage by an unique number established by the server, and by the "sender" of the message (as provided by the `USER` command explained below).

The clients connect to our server on port *bp*. We also assume a production environment so that we implement concurrency control.

1.1 Application Protocol

For reasons of interoperability a network application communicates using a strict protocol (called the application protocol). The particular application protocol used by your server is outlined in this section. Fixed-width font represents text which is fixed for the given command or response, while parameters that may vary are shown in italics.

Every command and response consists of a single line of text. The server should handle any combination of the characters `'\n'` and `'\r'` as line terminator and should send back responses terminated by a single `'\n'`. You should be able to test your server using `telnet` as a client or indeed any other client capable of sending and receiving plain text.

Greeting

At the beginning of the interaction the server send the following text to the client that just connected:

`0.0` *greeting*

where *greeting* is some (possibly empty) message intended for human consumption. There is no particular format for the greeting text, but it is strongly suggested for this text to summarize the commands available to clients.

[1] In case you are wondering, POSIX originally came from "Portable Operating System Interface for uniX". As time went by the standard was adopted by other operating systems, so that the "for uniX" part is no longer pertinent. Thus many people claim that nowadays POSIX stands for "Portable Operating System Interface with an X added at the end for coolness".

`USER` *name*

This command establishes the user name of all the subsequent messages being posted by the respective client. The argument *name* is a string not containing the character `/`. Future messages posted by the respective client will be identified as posted by *name*. Normally, a client will send this command at the beginning of the session, but the server should handle the case in which this command is sent more than once during the interaction with a particular client, as well as the case when a client does not send a `USER` command at all (case in which the poster will be `nobody`).

The server response is the line

`1.0 HELLO` *name text*

where *text* is some (possibly empty) message intended for human consumption.

Whenever the user name contains unacceptable characters (including but not necessarily limited to `'/'`) or is otherwise incorrect the response of the server is:

`1.2 ERROR USER` *text*

where *text* is once more intended for human consumption and explains the issue encountered in processing the request.

`READ` *message-number*

This command asks for the message number *message-number*. In the event that this message exists on the bulletin-board, the server will send in response one line of the form

`2.0 MESSAGE` *message-number poster*`/`*message*

where *message* represents the requested message, prefixed by its poster (as identified by the `USER` command in effect at the time of posting).

If message *message-number* does not exist, then the server sends the line:

`2.1 UNKNOWN` *message-number text*

where *text* is a message for human consumption. If the server encounters an internal error while serving the request (e.g., the unavailability of the bulletin board file), then the following response is sent back to the client:

`2.2 ERROR READ` *text*

Again, *text* is an explanatory message with no particular structure.

`WRITE` *message*

This command sends *message* to the server for storage. The server will store the message into the bulletin board file as a line of the form:

message-number`/`*poster*`/`*message*

where *message-number* is a unique number assigned by the server, and *poster* is the poster of the message as specified by a previous `USER` command issued by the respective client (`nobody` if no `USER` command has been issued by that client). Upon successful storage, the server returns the following message to the client:

`3.0 WROTE` *message-number*

When an error occurs during the storage process, the server responds with this message instead:

`3.2 ERROR WRITE` *text*

The receipt of such a response must guarantee that no message has been written to the bulletin board.

`REPLACE` *message-number*/*message*

This command asks the server to erase message number *message-number* and replace it with *message* (which will be assigned the same message number as the original). The poster is also changed to the current poster as identified by a previous `USER` command (`nobody` if no such command has been issued).

The server response is identical to the response to a `WRITE` request, plus

`3.1 UNKNOWN` *message-number*

when message number *message-number* does not exist in the bulletin board file (case in which no message is added to the file).

`QUIT` *text*

Signals the end of interaction. Upon receipt of this message the server sends back the line

`4.0 BYE` *some-text*

and closes the socket. The same response (including the socket close) is given by the server to a client that just shuts down its connection. The server always closes the socket in a civilized manner (by shutting down the socket before closing it).

1.2 Performance and Other Implementation Requirements

Your server must be robust, in the sense that no message shall be lost when the server is terminated, except possibly a message that is currently being written to disk. The bulletin board file should be considered too large to be kept completely in memory.

Your server must also be efficient, in the sense that it must not rewrite the whole bulletin board file upon the receipt of each and every message. It should use the file system as little as possible (within the robustness requirements above).

Now it is also the time to think about and implement a mechanism for rolling back the most recent transaction (write or replace). This is going to be used in the second part of the assignment.

You must build a concurrent server, which is able to serve many clients simultaneously. You must provide an implementation based on POSIX threads, using concurrency management and thread preallocation (as explained in Section 1.4).

1.3 The Bulletin Board File

The bulletin board file specified in the configuration file or on the command line must be created if nonexistent and must be re-used as is otherwise (rather than being overwritten). Message numbers are assigned by the server in sequence, according to the order in which the `WRITE` requests have been received. No two messages can have the same number in any bulletin board file. In particular, if the server is started on an existing file it should inspect the file on startup and make sure that any new message written to the file has an associated number that does not conflict with existing message numbers.

The access control to the bulletin board file follows the *readers/writers paradigm*, a common scheme in operating systems. Simultaneous reads of the bulletin board file by different threads of execution must be allowed. However, *no other operation* (read *or* write) is allowed when a thread writes to the bulletin board file. Of course, if a write request is issued while several read operations are in progress, the write will have to wait until the current reads complete. Note that this mechanism is slightly more complicated than the normal file locking. You may want to use a structure (that records the number of current read and write operations) protected by a critical region and also a condition variable to enforce this restriction. Inefficient implementations of this access control mechanism will be penalized. *Do not use file locking for implementing the access control since this method will not work as expected.*

Testing the fine access Obviously, testing the correct implementation of access control is primarily your responsibility. No matter how you choose to perform your tests however, the following method of testing *must* be available in your submission: Whenever the parameter D is `true`: (a) suitable messages marking the beginning and the end of a read or write operation should be provided to the standard output, and (b) each read and write operation should be artificially lengthened by different amounts of time (recommended values: 3 seconds for read operations and 6 seconds for write operations) using suitable `sleep()` statements inside the respective critical regions. I will let you figure out by yourself how is this helpful in debugging access control.

1.4 Concurrency Management

The server must use preallocated threads. The number of threads to be preallocated is $\mathbb{T}_{\mathtt{max}}$, so that $\mathbb{T}_{\mathtt{max}}$ is also a limit on concurrency.

1.5 Startup and Reconfiguration

Whenever the configuration file and the command line (see Section 3.1) results in a value of `true` for d the server performs the following startup sequence:

1. Binds to the port numbers as specified and performs any necessary initialization of the data structures.
2. Sets an appropriate umask.
3. Installs appropriate signal handlers for all the signals.
4. Leaves the current process group.
5. Closes all file descriptors and re-opens them as appropriate. In particular console output is redirected to the file `bbserv.log` located in the current working directory.
6. Detaches from the controlling tty.
7. Puts itself into background.
8. Check and writes into the PID file `bbserv.pid` located in the current working directory.

Whenever d is `false` the following steps above are not performed: 4, 5, 6, and 7.

The server reacts to the `SIGQUIT` and `SIGUP` signals as follows: It closes all the master sockets, terminates all the preallocated threads immediately after the current command (if any) is completed, and then closes all the connections to all the clients. If the signal received is `SIGQUIT` then the server terminates. If on the other hand `SIGHUP` is being handled, then the server re-reads the configuration file, applies the changes (if any) and restart the normal operation. However, any change of d is disregarded. Also note that as opposed to the normal startup (see Section 3.1) this time the (new) parameters re-read from the configuration file take precedence over the command line, including the list of peers (which is set exclusively according to the new configuration file).

2 Phase 2: Data Replication

We are now ready to implement a replicated database management system. Since we have it already we will use the bulletin board file as our database. This file is now kept replicated (and synchronized) on multiple servers.

Each server receive a list of the other servers that are participating in the synchronization. This is the list *peers* as specified in the configuration file and/or on the command line (see Setion 3.1). Each element in the list consists of a host name and a port number.

In addition, each server listens on port *sp* for incoming requests for synchronization.

2.1 Synchronization

The fact that this is a replicated database is transparent to the clients. All the peers are equally capable of serving clients as described in the previous section. The only perceptible difference is a possible delay in the response of a request to write or replace a message. Any `USER`, `READ`, or `QUIT` request is served locally as before. The synchronization between servers is initiated by the receipt of a `WRITE` or `REPLACE` command, and is accomplished using the *two-phase commit protocol*. This protocol is widely used in applications where data consistency is critical. 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 of two phases:

Precommit phase The master broadcasts to all the peers (in the list *peers*) 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. A negative acknowledgment is assumed if no acknowledgment comes until the master times out.

If there exists a negative acknowledgment, the master sends an "abort" message to the slaves and aborts the writing altogether (sending a `3.2 ERROR` message to the respective client). The slaves will abandon the whole process 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.

If the process is successful then a suitable `3.0 WROTE` message is sent to the client, otherwise a `3.2 ERROR` message is sent instead.

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, while the actions associated to the transitions are written in plain text).

2.2 Application Protocol

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. The protocol must be fully described in a file named `protocol2pc.txt` included in your submission.

In addition, whenever D is `true` your server must print to the standard output all the messages exchanged with its peers (sent and also received). Please identify in such a printout what are the messages that have been sent and what are the messages that have been received.

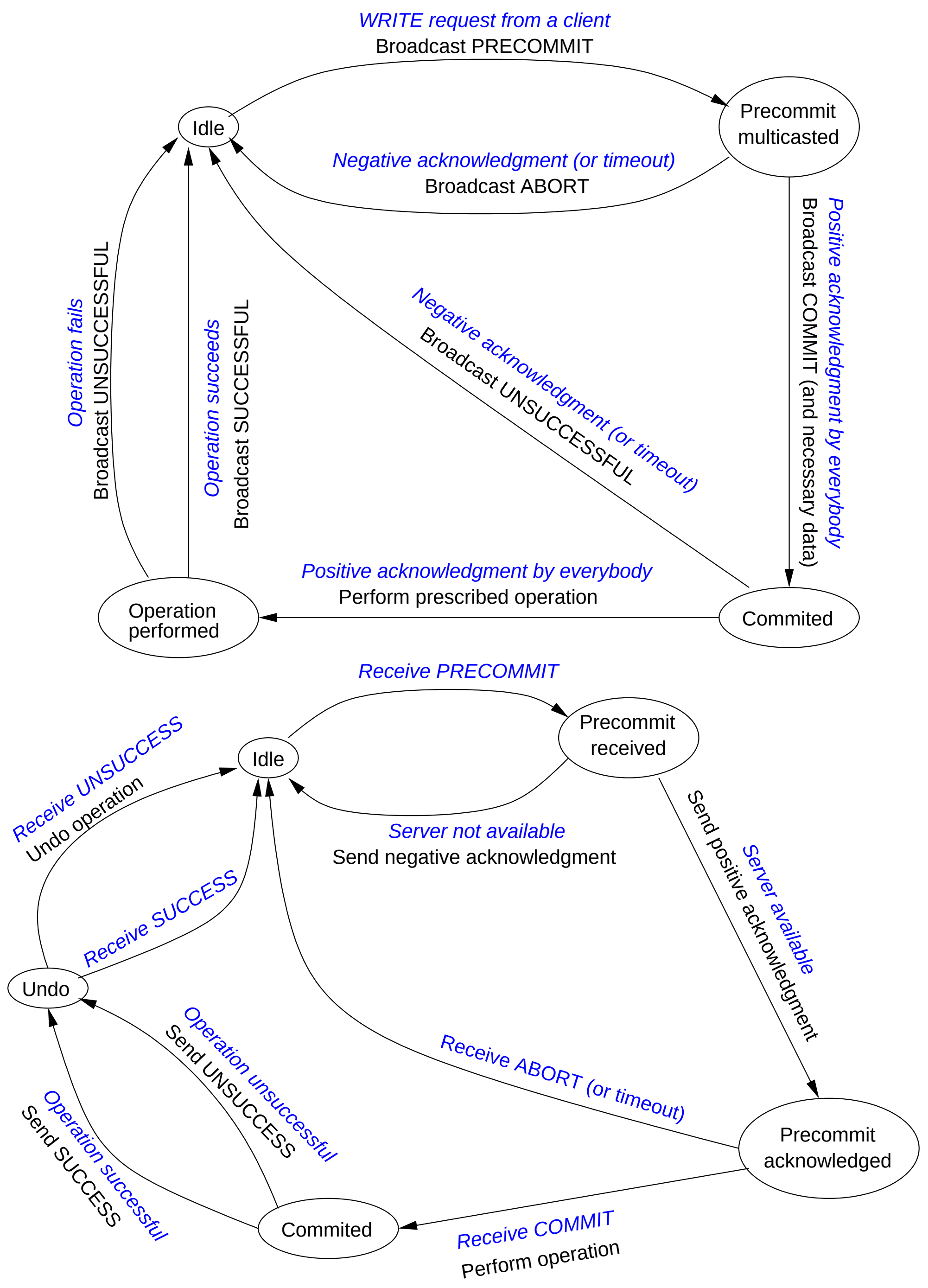

Figure 1: The master (above) and slave (below) in the two-phase commit protocol.

3 Implementation and Testing

Most of the implementation requirements have already been outlined earlier. The only major thing left to specify is how are the configuration parameters obtained. We also include a short discussion on debugging considerations.

3.1 Configuration

Upon startup, our server reads a configuration file. This file contains pairs consisting of a variable name and a value for that variable, separated by an = character (with no blanks). The configuration file includes (in no particular order) the following definitions:

```
THMAX=𝕋max
BBPORT=bp
SYNCPORT=sp
BBFILE=bbfile
PEERS=peers
DAEMON=d
DEBUG=D
```

where $\mathbb{T}_{\max}$, *bp*, and *sp* are strings representing positive numbers, *bbfile* is a file name, *D* and *d* are Boolean values and can be specified using either the numbers `0` and `1` or the strings `false` and `true`. The list of peers *peers* consists of strings separated by blanks. Each such a string has the form `host:port`, where `host` is a host name (DNS name or IP in dotted decimal notation[2]) and `port` is a positive number. You can assume that the list of peers does not include the current server. You do not need to validate the peers on startup. Instead you only need to try to connect to the peers when a synchronization is needed; any malformed peer will be handled by the synchronization protocol in the precommit phase.

Most of the lines in the configuration file are optional; each missing line causes the respective variable to take a default value, as follows: 20 for $\mathbb{T}_{\max}$, 9000 for *bp*, 10000 for *sp*, an empty list for *peers*, `true` for *d*, and `false` for *D*. The only mandatory data is *bbfile*; if the server cannot obtain the file name from either the configuration file or the command line (see below) then it must refuse to start (with a suitable error message printed to the standard output). The server never modifies the configuration file, even if it is missing or incomplete.

The default configuration file is called `bbserv.conf` and resides in the current directory. The name can be overridden by the command line option `-c` whose argument specifies (using an absolute or relative path) the configuration file to be used for the respective session.

All the options provided in the configuration file can be overridden using command line switches as follows:

- `-b` overrides (or sets) the file name *bbfile* according to its argument
- `-T` overrides $\mathbb{T}_{\max}$ according to its argument
- `-p` overrides the port number *bp* according to its argument
- `-s` overrides the port number *sp* according to its argument
- `-f` (with no argument) forces *d* to `false` or `0`
- *-d* (with no argument) forces *D* to `true` or `1`

[2]The dotted decimal notation specifies an IP address as a sequence of 4 bytes separated by dots. Each byte is a decimal number (and so ranges from 0 to 255). As an example 207.162.99.36 is the current IP address of `linux.ubishops.ca`.

Any non-switch argument is further interpreted as a peer specification (and so must have the form `host:port` as explained above).

If some option is specified in both the configuration file and by a switch on the command line then the command line version takes precedence. The non-switch arguments on the other hand are *additions* to the peers specified in the configuration file (meaning that they will be added to the list *peers*).

Command line switches must be ideally specified in any order and any combination (ideally they will be parsed using `getopt()`). However, at a minimum the following order must be supported: `-b`, `-f`, `-d`, `-t`, `-T`, `-p`, `-s`, `-c`, followed by the non-switch arguments. Note that all the switches are optional (meaning that any of them may be missing).

3.2 Debugging

Most (if not all) real-life servers contain debugging facilities, which serve two purposes. For one thing, concurrent applications are notoriously difficult to debug, so debugging facilities must be provided to facilitate this process. Secondly, if a server misbehaves in the wild the respective system administrator will typically enable these debugging facilities to identify and correct the problem if possible, or else provide meaningful data to the developer if the problem cannot be resolved locally.

In our case the debugging facilities are controlled by the parameter *D*. The only such facilities that are mandatory are the ones related to debugging the concurrent access to the bulletin board file and exposing the peer-to-peer communication (see Sections 1.3 and 2.2). Increasing the overall quantity of the messages produced to the standard output in the presence of this flag is highly recommended. I do not have actual suggestions what events to identify this way, but I am sure that throughout the development you will find the need to print out information in order to debug your server; these messages should be suppressed during normal operation, but I suggest that you re-enable them in the presence of the flag *D*.

3.3 Reference and Staging Computing Environments

Your server is expected to interact with the standard `telnet` client as well as any other client that sends and receives text. You should therefore be able to handle clients that terminate lines of text with the usual `'\n'` but also with the sequence `"\r\n"` (which is the case of `telnet`). Note that in fact according to the protocol specification (see Section 1.1) you should be able to handle any combination of `'\n'` and `'\r'` as line terminator.

The reference computing environment for this challenge is `linux.ubishops.ca`, meaning that this is the environment under which I will compile and run your solutions. This environment is a Linux machine which provides recent version of GCC (currently 11.2.0, subject to change) and GNU make (currently 4.3).

This being said, a well written solution should build and run on any reasonably current POSIX system with a standard C/C++ compiler (including but not necessarily limited to GCC and LLVM) and a non-ancient version of make (GNU or otherwise). You can easily set up such an environment using any current Linux distribution (make sure that you install the development packages) or Mac OS (with Xcode as well as the Xcode command line tools installed). Such an environment can also be created in Windows, but I am not able to provide advice on this matter since it exceeds my expertise. However, it is still a good idea to test out your solution on `linux.ubishops.ca` before submitting (just in case).

In addition to `linux.ubishops.ca` the following machines will become available for you sometime in the week of 15 May to test your work:

```
10.18.0.21  10.18.0.22  10.18.0.23
10.18.0.24  10.18.0.25  10.18.0.26
10.18.0.27  10.18.0.28  10.18.0.29
```

These machines are only accessible from `linux.ubishops.ca` (meaning that you will have to connect to `linux.ubishops.ca` first and then ssh into the machines from there). The machines do not feature any development environment, so that you must build your work on `linux.ubishops.ca` before running it on these machines.

Your credentials for these machines will be the same as for `linux.ubishops.ca` on 15 May; make sure that your account on `linux.ubishops.ca` works and you know your user name and password for that account by then. The machines are however independent, so a change of password on `linux.ubishops.ca` will not propagate to any other machine, and changing your password on one machine will not propagate to the others. The home directories are also independent from each other.

Note that these machines are weak (1 CPU core and 1 GB of RAM each). It is probably a good idea to try to balance their use (e.g., if you see too many people logged into one move to another), but as long as your program runs reasonably well there is not reason to worry.

The machines are to be regarded as disposable, meaning that they can be reset to their initial form at any time and without warning. While in principle they will not be reset until the end of the term, this cannot be guaranteed, so make sure that all your data is backed up elsewhere.

This all being said, note that the testing of your server does not need a multi-machine environment. You can also run multiple replica servers on the same machine but out of different directories and bound to different port numbers. It is however a good idea to test your implementation on multiple machines at least once in order to eliminate any possible bugs that do not manifest themselves on a single-machine run.

Be aware that you will be working in a multi-user environment and so it is reasonable to assume that each machine will run multiple instances of the server. It is therefore a wise idea to choose unique port numbers. I recommend that you use 8000 + your user ID as port number for service to clients, and 9000 + your user ID as port number for synchronization. The following two shell commands will print out these numbers when run on `linux.ubishops.ca` or any of the staging machines:

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

Even better, you can provide appropriate port numbers directly on the command line of your server along the following line:

```
./bbserv -p $((8000+$(id -u))) -s $((9000+$(id -u))) -b bbfile
```

Note that if you use the staging machines then you should restrict yourself to port numbers between `9000` and `11000` since other port numbers are blocked at the firewall level.

4 Submission

Submit the source code for your server plus appropriate documentation. Note that a description of the application protocol for synchronization is a compulsory part of the documentation.

4.1 Presentation

Your submission must contain the following:

1. *Your working code.*
2. *A suitable makefile.* Building your code must produce an executable named `bbserv` in the root directory of your submission. The build process must complete successfully when using GNU make and GCC by simply typing `make` at the shell prompt in the root directory of your submission. I recommend that you build a simple makefile by hand, but generated makefiles are also fine (as log as they work in the reference environment).

   Also provide the target `clean` that removes all the object, executable, log, and any other unnecessary files, thus restoring the submission to its pristine condition, ready to be built and run.
3. *Documentation* is an integral part of your submission and should include:

- A brief "user guide" documenting any deviations from, or additions to the specified human interface or protocols (as the case may be). Do not repeat the specification, just document deviations and/or additions. Also provide descriptions (or references thereof) to all the algorithms used by your code which were not given in the handout. Proper credit should be given to any code that you did not write and/or that existed before you started coding for this challenge. Describe here any non-obvious implementation solution you came up with. If you did some smart coding by all means document your smartness (otherwise I might miss it).
- Suitable comments in the code, outlining the functionality of all the pieces of code and referencing the section on algorithms and functionality (above).
- The file `protocol2pc.txt` (see Section 2.2).

   The documentation (except the content of `protocol2pc.txt`) can be included in the README file (see below) if its size does not justify a separate document.

   All the documentation should be provided in *plain text*; no other format is acceptable.

4. *A README file*. This file must be called `README` (all capitals and with no extension) and must be plain text. The first line must contain the name of all the collaborators separated by commas. The second line must contain a list of email addresses again separated by commas. Exactly all the email addresses on this line will be used for sending feedback on your submission; you can specify as many email addresses as you wish.

   The remainder of the file is free form and contain any information you may wish to convey. In particular I recommend to include here the documentation (see above) instead of having it in a different file.

4.2 How to Submit

Your solution should be handed in electronically, as *one* directory which in turn contains your work. Do not create any kind of archive; simply submit the directory itself. The name of the submitted directory need *not* follow any particular format.

The following submission method is compulsory; work submitted in any other way will not be marked. Log into `linux.ubishops.ca` and submit using the following command:

```
submit csc590 dir
```

where *dir* is the directory you want to submit. You can specify it by either a relative or an absolute path. Also note the capitalization of "cs590," which must be observed.

Example Let's say that your work (code, documentation, etc.) is in the directory `~/networking/bbserv`, and that the current working directory is `~/networking`. You should then type *one* of the following commands:

```
submit cs590 ~/networking/bbserv
submit cs590 bbserv
submit cs590 ./bbserv
```

Type `man submit` for more details.

Re-submissions and late submissions Re-submissions within the due date are fine (just issue the above command as many times as you wish). Note however that only the most recent submission will be kept. Late submissions will be accepted with a 20% penalty per day late as long as the `submit` program does not say otherwise.

5 Grading

The marking process will assume a production server. Such a server should not deviate from the application protocol and generally should behave as specified. Most tests are automated and so silly mistakes may result in substantial loss of marks (so do not make them).

Marking is based on semi-automated tests carried on a fresh build of your code. Your code will receive only a cursory inspection to ensure that you have observed the implementation limitations (if applicable), and to establish code ownership. Bad programming practice and dirty code will not impact your grade unless they impact the execution of the program, or are conspicuous enough to be easily noticed in said cursory code inspection.

I will build and run your program on a couple of batches of standardized testing scenarios. The first batch will run your server with an empty list *peers* (thus testing Phase 1) and the second batch will run multiple instances of the server (thus testing Phase 2). I will be using my own configuration file copied over to your directory. All of this is mostly automated. It is therefore crucial to follow the build process and command line to the letter. Note in particular that the ideal handling of the command line switches is to accept them in any order, but at the minimum they should be accepted in the order in which they are specified in Section 3.1.

As a consequence of this marking scheme your program *must* build exactly as specified and must accept input exactly as specified. Any deviation from the build specification or run time input requirements (command line arguments, files, etc) will result in the submission not being marked at all and so receiving zero marks. I will *not* debug or fix during marking build errors or run time errors caused by not observing the given specification; trivial mistakes can be very costly so don't make them. If in doubt ask for assistance *before* submitting your work.

Penalties will be applied on all deviations from the specified behaviour, including but not necessarily limited to incorrect implementation of the application protocols, incorrect concurrent behaviour, busy-waiting loops, and deadlocks. Note in particular that in network applications any communication should have an associated timeout (to eliminate the possibility of deadlocks caused by unreliable communication).

In addition to all of the above be aware that identical or nearly identical submissions by two (or more) teams will receive (all of them) an automatic grade of 0. Further action according to the departmental and university guidelines for academic dishonesty may be pursued.

Marking scheme The various aspects of your submission will have the following weight in the final grade:

- Basic client-server communication: 30%
- Concurrent access to the bulletin board file: 20%
- Peer-to-peer synchronization: 30%
- Timeouts, busy loops (or rather lack thereof), deadlock avoidance, etc.: 10%
- Signal handling: 10%

6 Resources

It is your responsibility to become familiar with POSIX and network programming. The resources suggested below will probably accomplish this, but I am sure that many more resources exist out there.

The UNIX implementation of the TCP/IP network stack is Berkley sockets. This, together with other standard APIs were described in CS 464/564. This course also offered a presentation of programming techniques for concurrent and distributed applications. Sections of particular interest include the lecture notes (overhead slides and also recorded video lectures) and the code samples. The latter section contains in particular an archive with various TCP utilities which you are welcome to use (and will most likely make your life easier).

On top of this, a very useful resource is the *manual pages*. On any UNIX machine with the developer tools installed you can type

```
man function
```

for a detailed description of *function* (which could be a system or library function but also some other programming construct). Sometimes a command like the above can produce the wrong manual page. For example

```
man getopt
```

will produce the manual page of the UNIX command `getopt` rather than the library function with the same name. This happens because manual pages are split into numbered sections and by default the first hit is returned. Given that `getopt` is a shell command it has a manual page in Section 1, while the manual pages for library function (which is presumably what you are looking for) are described in Section 3. To obtain the correct manual page in such a case you must specify the section number (3 in our example) explicitly as the first argument. Therefore the command that will give you the manual page of the function `getopt` is

```
man 3 getopt
```

Note that shell commands are described in Section 1, system functions (OS API) are described in Section 2, and standard C library functions in Section 3. It is sometime difficult to realize right off the bat which function is a system function and which is a library function; in such a case the only recourse would be to try both sections. Occasionally you may need to go to Section 7 which is the rest of the Linux programmer's manual. As an example signals are described in the manual page `signal` residing in Section 7.

An excellent introduction into network and system programming in UNIX is *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version* (by Douglas E. Comer and David L. Stevens, Prentice Hall, 2001). This is actual the basis of the lectures mentioned at the beginning of this section.