The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2022
Assignment 4 (version 1.0)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 4:00pm Friday 3 June, 2022

Introduction

The goal of this assignment is to further develop your C programming skills, and to demonstrate your understanding of networking and multithreaded programming. You are to create two programs. One – `dbserver` – is a networked database server that takes requests from clients, allowing them to store, retrieve and delete string-based key/value pairs. The other – `dbclient` – is a simple network client that can query the database managed by `dbserver`. Communication between the `dbclient` and `dbserver` is done using HTTP requests and responses, using a simple RESTful API. Advanced functionality such as authentication, connection limiting, signal handling and statistics reporting are also required for full marks.

The assignment will also test your ability to code to a particular programming style guide, to write a library to a provided API, and to use a revision control system appropriately.

Student Conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository, and do not allow others to access your computer - you must keep your code secure.

You must follow the following code referencing rules for all code committed to your SVN repository (not just the version that you submit):

Code Origin	**Usage/Referencing**
Code provided to you in writing **this semester** by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, posted on the discussion forum, or shown in class).	May be used freely without reference. (You must be able to point to the source if queried about it.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3)	May be used freely without reference. (This assumes that no reference was required for the original use.)
Code examples found in man pages on `moss`.	May be used provided the source of the code is referenced in a comment adjacent to that code.
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has not been shared or published.	
Code (in any programming language) that you have taken inspiration from but have not copied.	
Other code – includes: code provided by teaching staff only in a previous offering of this course (e.g. previous A1 solution); code from websites; code from textbooks; any code written by someone else; and any code you have written that is available to other students.	May **not** be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail). Copied code without adjacent referencing will be considered misconduct and action will be taken.

Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and they cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you are not able to adequately explain your code or the design of your solution and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Specification – `dbclient`

The `dbclient` program provides a commandline interface to allow access to a subset of the `dbserver`'s capabilities, in particular it permits only the setting and retrieving of key/value pairs. It does not support `dbserver` authentication, or deletion of key/value pairs. **Note that you will need to read the dbserver specification below also to fully understand how the programs communicate.**

`dbclient` does not need to be multithreaded.

Command Line Arguments

Your `dbclient` program is to accept command line arguments as follows:

```
./dbclient portnum key [value]
```

- The `portnum` argument indicates which localhost port `dbserver` is listening on – either numerical or the name of a service.
- The `key` argument specifies the key to be read or written.
- The `value` argument, if provided, specifies the value to be written to the database for the corresponding key. If `value` is not provided, then `dbclient` will read the value from the database.
- Any additional arguments are to be silently ignored.

`dbclient` Behaviour

If insufficient command line arguments are provided then `dbclient` should emit the following message (terminated by a newline) to `stderr` and exit with status 1:

```
Usage: dbclient portnum key [value]
```

If the correct number of arguments is provided, then further errors are checked for in the order below.

Restrictions on the key value

The key value must not contain any spaces or newlines. If it does then `dbclient` should emit the following message (terminated by a newline) to `stderr` and exit with status 1:

```
dbclient: key must not contain spaces or newlines
```

Connection error

If `dbclient` is unable to connect to the server on the specified port of localhost, it shall emit the following message (terminated by a newline) to `stderr` and exit with status 2:

```
dbclient: unable to connect to port N
```

where `N` should be replaced by the argument given on the command line.

`GET`ting key/value pairs

If no `value` argument is specified, then `dbclient` shall send a `GET` HTTP request to the `dbserver` for the specified `key` in the public database. The HTTP request will look like this:

```
GET /public/<key> HTTP/1.1

```

where `<key>` is replaced by the requested database key. Note that the request (first) line must be terminated by a carriage-return line-feed (CRLF) sequence (CRLF = `\r\n`) and be followed by a similarly-terminated blank line. There is no body part necessary in the request.

If the server returns a 200 (OK) response, then `dbclient` shall emit the returned value string to `stdout` (with a terminating newline), and exit with status 0.

If the server returns any other response (e.g. 404 Not Found) or the connection with the server is lost, then `dbclient` shall emit no output, and exit with status 3.

`PUT`ting key/value pairs

If `value` is specified, then `dbclient` shall attempt to set the corresponding key/value pair, using a `PUT` HTTP request as follows (see the Communication protocol section below for details):

```
PUT /public/<key> HTTP/1.1
Content-Length: <N>

<value>
```

where `<key>` and `<value>` are replaced with the required key and value strings respectively, and `<N>` is replaced by the number of bytes in `<value>`. As always, lines in a HTTP request should be terminated by a CRLF sequence. The body part of the request must be the unmodified value part of the key-value pair – no newlines are present unless they are part of the value.

If the server returns a 200 (OK) response, then `dbclient` shall exit with status 0. If the server returns any other response (e.g. 404 Not Found or 503 Service Unavailable) or the connection with the server is lost, then `dbclient` shall exit with status 4.

`dbclient` example usage

Setting and reading values from the database (assuming the `dbserver` is listening on port 49152):

```
$ ./dbclient 49152 mykey myvalue
$ echo $?
0
$ ./dbclient 49152 mykey
myvalue
$ echo $?
0
```

Using shell quoting for values containing spaces and/or newlines:

```
$ ./dbclient 49152 key "my long value
spread over two lines"
$ ./dbclient 49152 key
my long value
spread over two lines
```

Attempting to read a missing key:

```
$ ./dbclient 49152 badkey
$ echo $?
3
```

Failed attempt to write a key/value pair:

```
$ ./dbclient 49152 somekey somevalue
$ echo $?
4
$ ./dbclient 49152 somekey
$ echo $?
3
```

Specification – `dbserver`

`dbserver` is a networked database server, capable of storing and returning text-based key/value pairs. Client requests and server responses are communicated over HTTP.

The `GET` operation permits a client to query the database for the provided key. If present, the server returns the corresponding stored value.

The `PUT` operation permits a client to store a key/value pair. If a value is already stored for the provided key, then it is replaced by the new value.

The `DELETE` operation permits a client to delete a stored key/value pair.

`dbserver` must implement at least one database instance, known as `public`, which can be accessed by any connecting client without authentication.

For advanced functionality and additional marks, `dbserver` must manage a second database instance, called `private`, access to which is only permitted if the client provides the correct authentication string in the HTTP request headers. This functionality will be described in detail below.

Command Line Arguments

Your `dbserver` program is to accept command line arguments as follows:

`./dbserver authfile connections [portnum]`

In other words, your program should accept two mandatory arguments (`authfile` and `connections`), and one optional argument which is the port number to listen on.

The `authfile` argument is the name of a text file, the first line of which is to be used as an authentication string (see below for details).

The `connections` argument indicates the maximum number of simultaneous client connections to be permitted. If this is zero, then there is no limit to how many clients may connect (other than operating system limits which we will not test).

The `portnum` argument, if specified, indicates which localhost port `dbserver` is to listen on. If the port number is absent, then `dbserver` is to use an ephemeral port.

Important: Even if you do not implement the authentication or connection limiting functionality, your program must still correctly handle command lines which include those arguments (after which it can ignore any provided values – you will simply not receive any marks for those features).

Program Operation

The `dbserver` program is to operate as follows:

- If the program receives an invalid command line then it must print the message:

  ```
  Usage: dbserver authfile connections [portnum]
  ```

  to `stderr`, and exit with an exit status of 1.

  Invalid command lines include (but may not be limited to) any of the following:

  - no authfile or connections number specified
  - the connections argument is not a non-negative integer
  - the port number argument (if present) is not an integer value either zero, or in the range of 1024 to 65535 inclusive

- If `portnum` is missing or zero, then `dbserver` shall attempt to open an ephemeral localhost port for listening. Otherwise, it shall attempt to open the specific port number.
- If `authfile` cannot be opened for reading, or the first line is empty then `dbserver` shall emit the following message to `stderr` and terminate with exit status 2:

  ```
  dbserver: unable to read authentication string
  ```

- If `dbserver` is unable to listen on either the ephemeral or specific port, it shall emit the following message to `stderr` and terminate with exit status 3:

  ```
  dbserver: unable to open socket for listening
  ```

- Once the port is opened for listening, `dbserver` shall print the port number to `stderr`, followed by a single newline character, and then flush the output. **In the case of an ephemeral port, the actual port number obtained shall be printed, not zero**.
- Upon receiving an incoming client connection on the port, `dbserver` shall spawn a new thread to handle that client (see below for client thread handling).
- If specified (and implemented), `dbserver` must keep track of how many client connections have been made, and must not let that number exceed the `connections` parameter. See below on client handling threads for more details on how this limit is to be implemented.
- Note that all error messages must be terminated by a single newline character.
- The `dbserver` program should not terminate under normal circumstances, nor should it block or otherwise attempt to handle `SIGINT`.
- Note that your `dbserver` must be able to deal with any clients using the correct communication protocol, not just `dbclient`. In particular, note that `dbclient` will only send one request per connection but other clients may send multiple requests per connection.

Client handling threads

A client handler thread is spawned for each incoming connection. This client thread must then wait for HTTP requests, one at a time, over the socket. The exact format of the requests is described in the Communication protocol section below.

To deal with multiple simultaneous client requests, your `dbserver` will need some sort of mutual exclusion structure around access to your key-value store.

Once the client disconnects or there is a communication error on the socket or a badly formed request is received then the client handler thread is to close the connection, clean up and terminate. (A properly formed request for an unavailable service shall be rejected with a Bad Request response – it will not result in termination of the client thread.)

`SIGHUP` handling (Advanced)

Upon receiving `SIGHUP`, `dbserver` is to emit (and flush) to `stderr` statistics reflecting the program's operation to-date, specifically

- Total number of clients connected (at this instant)
- The total number of clients that have connected and disconnected since program start
- The total number of authentication failures (since program start)
- The total number of <u>successful</u> `GET` requests processed (since program start)
- The total number of <u>successful</u> `PUT` requests received (since program start)
- The total number of <u>successful</u> `DELETE` requests received (since program start)

The required format is illustrated below.

Listing 1: `dbserver` SIGHUP `stderr` output sample

```
Connected clients:4
Completed clients:20
Auth failures:4
GET operations:4
PUT operations:15
DELETE operations:0
```

Note that to accurately account these statistics and avoid race conditions, you will need some sort of mutual exclusion structure protecting the variables holding these statistics.

Client connection limiting (Advanced)

If the `connections` feature is implemented and a non-zero command line argument is provided, then `dbserver` must not permit more than that number of simultaneous client connections to the database. `dbserver` shall maintain a connected client count, and if a client beyond that limit attempts to connect, `dbserver` shall send a 503 (Service Unavailable) HTTP response to that client and immediately terminate the connection.

Program output

Other than error messages, the listening port number, and `SIGHUP`-initiated statistics output, `dbserver` is not to emit any output to `stdout` or `stderr`.

Communication protocol

The communication protocol uses HTTP. The client (`dbclient`, or other program such as `netcat`) shall send HTTP requests to the server, as described below, and the server (`dbserver`) shall send HTTP responses as described below. The connection between client and server is kept alive between requests. Supported request types are `GET`, `PUT` and `DELETE` requests.

Additional HTTP header lines beyond those specified may be present in requests or responses and must be ignored by the respective server/client. Note that interaction between `dbclient` and `dbserver` is *synchronous* – a single client can only have a single request underway at any one time. This greatly simplifies the implementation of `dbclient` and the `dbserver` client-handling threads.

- Request: `GET /public/`*`key`*` HTTP/1.1`
  - Description: the client is requesting the value associated with the provided key, from the public database instance.
  - Request Headers: none expected, any headers present will be ignored by `dbserver`.
  - Request Body: none, any request body will be ignored
  - Response Status: either 200 (OK) or 404 (Not Found). 200 will be returned if the provided key exists in the database, otherwise 404 will be returned.
  - Response Headers: the `Content-Length` header with correct value is required (number of bytes in the response body), other headers are optional.
  - Response Body: the corresponding `value` from the public database (or empty if the key is not found).
- Request: `PUT /public/`*`key`*` HTTP/1.1`
  - Description: the client is requesting the server to write a key/value pair to the public database.
  - Request Headers: the `Content-Length` header is expected (the value will be the number of bytes in the request body). Other headers shall be ignored.
  - Request Body: the `value` to be set for the corresponding key/value pair.
  - Response Status: either 200 (OK) or 500 (Internal Server Error). 200 will be returned if the database update operation succeeds, otherwise 500 will be returned.

    - Response Headers: the `Content-Length: 0` header is expected (the zero value is the number of bytes in the response body), other headers are optional.
    - Response Body: None.

- Request: `DELETE /public/`*`key`* `HTTP/1.1`
    - Description: the client is requesting the server to delete a key/value pair from the public database.
    - Request Headers: none expected, any headers present will be ignored by `dbserver`.
    - Request Body: None.
    - Response Status: either 200 (OK) or 404 (Not Found). 200 will be returned if the database delete operation succeeds, otherwise 404 will be returned.
    - Response Headers: the `Content-Length: 0` header is expected, other headers are optional.
    - Response Body: None.

- Any other (well-formed) requests should result in `dbserver` sending a 400 (Bad Request) HTTP response. Badly formed requests (i.e. the data received can not be parsed as a HTTP request) will result in the server disconnecting the client (as described earlier). Note that any attempt to use a space in the key will be interpreted as a bad request because the HTTP request line will have too many spaces.

Advanced communication protocol – authentication

If implemented, the HTTP request `Authorization:` header may be used (note American spelling with a 'z'), along with an authentication string, to access the private database instance. This access is identical to the standard protocol above, with the following extensions

- All URL addresses become `/private/`*`key`*
- The request header `"Authorization: <authstring>"` must be provided, where `<authstring>` is replaced by the correct authentication string
- If the authentication header is not provided, or is incorrect, `dbserver` is to respond with the HTTP response "401 (Unauthorized)"

Note that library functions are provided to you to do most of the work of parsing/constructing HTTP requests/responses. See below.

The `stringstore` database library and API

An API (Application Programming Interface) for the database implementation, known as 'StringStore', is provided to you in the form of a header file, found on moss in `/local/courses/csse2310/include/stringstore.h`.

An implementation of `stringstore` is also available to you on moss, in the form of a shared library file (`/local/courses/csse2310/lib/libstringstore.so`).

However, **to receive full marks you must implement your own version of StringStore** according to the API specified by `stringstore.h`. You must submit your `stringstore.c` and your `Makefile` must build this to your own `libstringstore.so`.

This will allow you to start writing `dbserver` without first implementing the underlying database.

Your `dbserver` must use this API and link against `libstringstore.so`. Note that we will use our `libstringstore.so` when testing your `dbserver` so that you are not penalised in the marking of `dbserver` for any bugs in your StringStore implementation. You are free to use your own `libstringstore.so` when testing yourself – see below.

`stringstore.h` defines the following functions:

Listing 2: `stringstore.h` contents

```
/// Opaque type for StringStore - you'll need to define 'struct StringStore'
// in your stringstore.c file
typedef struct StringStore StringStore;

// Create a new StringStore instance, and return a pointer to it
StringStore *stringstore_init(void);
```

```
// Delete all memory associated with the given StringStore, and return NULL
StringStore *stringstore_free(StringStore *store);

// Add the given 'key'/'value' pair to the StringStore 'store'.
// The 'key' and 'value' strings are copied with strdup() before being
// added to the database. Returns 1 on success, 0 on failure (e.g. if
// strdup() fails).
int stringstore_add(StringStore *store, const char *key, const char *value);

// Attempt to retrieve the value associated with a particular 'key' in the
// StringStore 'store'.
// If the key exists in the database, return a const pointer to corresponding
// value string.
// If the key does not exist, return NULL
const char *stringstore_retrieve(StringStore *store, const char *key);

// Attempt to delete the key/value pair associated with a particular 'key' in
// the StringStore 'store'.
// If the key exists and deletion succeeds, return 1.
// Otherwise, return 0
int stringstore_delete(StringStore *store, const char *key);
```

Note that it is not expected that your StringStore library be thread safe – the provided `libstringstore.so` is not.

Creating shared libraries – `libstringstore.so`

This section is only relevant if you are creating your own implementation of `libstringstore.o`.

Special compiler and linker flags are required to build shared libraries. Here is a Makefile fragment you can use to turn `stringstore.c` into a shared library, `libstringstore.so`:

Listing 3: Makefile fragment to compile and build a shared library

```
CC=gcc
LIBCFLAGS=-fPIC -Wall -pedantic -std=gnu99

# Turn stringstore.c into stringstore.o
stringstore.o: stringstore.c stringstore.h
        $(CC) $(LIBCFLAGS) -c $<

# Turn stringstore.o into shared library libstringstore.so
libstringstore.so: stringstore.o
        $(CC) -shared -o $@ stringstore.o
```

Running `dbserver` with your `libstringstore.so` library

The shell environment variable `LD_LIBRARY_PATH` specifies the path (set of directories) searched for shared libraries. On moss, by default this includes `/local/courses/csse2310/lib`, which is where the provided libraries `libcsse2310a4.so` and our implementation of `libstringstore.so` live.

If you are building your own version and wish to use it when you run `dbserver`, then you'll need to make sure that **your** version of `libstringstore.so` is used. To do this you can use the following:

```
LD_LIBRARY_PATH=.:${LD_LIBRARY_PATH} ./dbserver
```

This commandline sets the `LD_LIBRARY_PATH` **just for this specific run of `dbserver`**, causing it to dynamically link against your version of `libstringstore.so` in the current directory instead of the one we have provided.

Note that we will use our `libstringstore.so` when testing your `dbserver`. Your `libstringstore.so` will be tested independently.

Provided Libraries

libstringstore

See above.

libcsse2310a4

Several library functions have been provided to you to aid parsing/construction of HTTP requests/responses. These are:

```
char** split_by_char(char* str, char split, unsigned int maxFields);

int get_HTTP_request(FILE *f, char **method, char **address,
        HttpHeader ***headers, char **body);

char* construct_HTTP_response(int status, char* statusExplanation,
        HttpHeader** headers, char* body);

int get_HTTP_response(FILE *f, int* httpStatus, char** statusExplain,
        HttpHeader*** headers, char** body);

void free_header(HttpHeader* header);

void free_array_of_headers(HttpHeader** headers);
```

These functions and the `HttpHeader` type are declared in `/local/courses/csse2310/include/csse2310a4.h` on moss and their behaviour is described in man pages on moss – see `split_by_char(3)`, `get_HTTP_request(3)`, and `free_header(3)`.

To use these library functions, you will need to add `#include <csse2310a4.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing these functions. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a4`

(You only need to specify the `-I` and `-L` flags once if you are using multiple libraries from those locations, e.g. both `libstringstore` and `libcsse2310a4`.)

libcsse2310a3

You are also welcome to use the `"libcsse2310a3"` library from Assignment 3 if you wish.

Style

Your program must follow version 2.2 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site. Note that, in accordance with the style guide, function comments are not expected in your `stringstore.c` for functions that are declared and commented in `stringstore.h`.

Hints

1. Review the lectures related to network clients, HTTP, threads and synchronisation and multi-threaded network servers. This assignment builds on all of these concepts.
2. Write a small program to explore the basic HTTP request/response patterns, and associated API functions in `libcsse2310a4.so`.
3. You can test `dbclient` and `dbserver` independently using `netcat` as demonstrated in the lectures. You can also use the provided demo programs `demo-dbclient` and `demo-dbserver`.
4. The `read_line()` function from `libcsse2310a3` may be useful for reading the authentication key.
5. The multithreaded network server example from the lectures can form the basis of `dbserver`.

6. Use the provided library functions (see above).
7. Consider a dedicated signal handling thread for `SIGHUP`. `pthread_sigmask()` can be used to mask signal delivery to threads, and `sigwait()` can be used in a thread to block until a signal is received. You will need to do some research and experimentation to get this working.

Forbidden Functions

You must not use any of the following C functions/statements. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `longjmp()` and equivalent functions
- `system()`
- `mkfifo()` or `mkfifoat()`
- POSIX regex functions
- `fork()`, `pipe()`, `popen()`, `execl()`, `execvp()` and other `exec` family members.

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or test files. **You are not expected to submit `stringstore.h`. If you do so, it will be marked for style.**

Your programs (named `dbclient` and `dbserver`) and your stringstore library (named `libstringstore.so` if you implement it) must build on `moss.labs.eait.uq.edu.au` with:

```
make
```

If you only implement one or two of the programs/library then it is acceptable for `make` to just build those programs/library – and we will only test those programs/library.

Your programs must be compiled with gcc with at least the following switches (plus applicable `-I` options etc. – see *Provided Libraries* above):

```
-pedantic -Wall -std=gnu99
```

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (e.g. an executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries (besides those we have provided for you to use).

Your assignment submission must be committed to your subversion repository under

```
https://source.eait.uq.edu.au/svn/csse2310-sem1-sXXXXXXX/trunk/a4
```

where sXXXXXXX is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/a4` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes – the `reptesta4.sh` script will do this for you.

To submit your assignment, you must run the command

```
2310createzip a4
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)[1]. The zip file will be named

[1]You may need to use scp or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.

`sXXXXXXX_csse2310_a4_`*`timestamp`*`.zip`

where sXXXXXXX is replaced by your moss/UQ login ID and *timestamp* is replaced by a timestamp indicating the time that the zip file was created.

The `2310createzip` tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command '`make`', and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

We will mark your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline[2] will incur a late penalty – see the CSSE2310/7231 course profile for details.

Marks

Marks will be awarded for functionality and style and documentation. Marks may be reduced if you are asked to attend an interview about your assignment and you are unable to adequately respond to questions – see the **Student conduct** section above.

Functionality (60 marks)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. Reasonable time limits will be applied to all tests. If your program takes longer than this limit, then it will be terminated and you will earn no marks for the functionality associated with that test. If your `dbserver` does not link against `libstringstore.so` then you can earn no marks for tests that interact with the key-value pair database (i.e. all tests in categories 12 to 15, and most tests in categories 16 to 19).

Exact text matching of files and output (`stdout` and `stderr`) is used for functionality marking. Strict adherence to the output format in this specification is critical to earn functionality marks.

The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories. There are 10 marks for `libstringstore.o`, 10 marks for `dbclient` and 40 marks for `dbserver`.

1. `libstringstore` – correctly create and free memory associated with a StringStore object (4 marks)
2. `libstringstore` – correctly add and retrieve key/value pairs (4 marks)
3. `libstringstore` – correctly delete key/value pairs (2 marks)

4. `dbclient` correctly handles invalid command lines (including invalid keys) (2 marks)
5. `dbclient` connects to server and also handles inability to connect to server (2 marks)
6. `dbclient` correctly requests key values from the server (2 marks)
7. `dbclient` correctly sets key/value pairs on the server (2 marks)
8. `dbclient` correctly handles communication failure (2 marks)

9. `dbserver` correctly handles invalid command lines (3 marks)

[2]or your extended deadline if you are granted an extension.

10. `dbserver` correctly listens for connections and reports the port (3 marks)

11. `dbserver` correctly handles invalid and badly formed HTTP requests (3 marks)

12. `dbserver` correctly handles public key/value pair set/get (6 marks)

13. `dbserver` correctly handles public key/value pair delete (3 marks)

14. `dbserver` correctly handles authenticated/private key/value pair set/get (4 marks)

15. `dbserver` correctly handles authenticated/private key/value pair delete (3 marks)

16. `dbserver` correctly handles multiple simultaneous client connections using threads (including protecting data structures with mutexes) (4 marks)

17. `dbserver` correctly handles disconnecting clients and communication failure (3 marks)

18. `dbserver` correctly implements client connection limiting (4 marks)

19. `dbserver` correctly implements SIGHUP statistics reporting (4 marks)

Some functionality may be assessed in multiple categories, e.g. it is not possible test the deletion of keys without being able to add keys first.

Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.2 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission. This does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All `.c` and `.h` files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not[3].

Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your `.c` and `.h` files as follows. If any of your submitted `.c` and/or `.h` files are unable to be compiled by themselves then your automated style mark will be zero (0). (Automated style marking can only be undertaken on code that compiles. The provided `style.sh` script checks this for you.)

If your code does compile then your automated style mark will be determined as follows: Let

- W be the total number of distinct compilation warnings recorded when your `.c` files are individually built (using the correct compiler arguments)
- A be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually[4].

Your automated style mark S will be

$$S = 5 - (W + A)$$

[3] Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

[4] Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file.

If $W + A \geq 5$ then S will be zero (0) – no negative marks will be awarded. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on moss. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`.

Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for "comments", "naming" and "other". The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1.0	All names used are appropriate

Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

SVN commit history assessment (5 marks)

Markers will review your SVN commit history for your assignment up to your submission time. This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality) and/or why the change has been made and will be usually be more detailed for significant changes.)

The standards expected are outlined in the following rubric:

Mark (out of 5)	**Description**
0	Minimal commit history – single commit OR all commit messages are meaningless.
1	Some progressive development evident (more than one commit) OR at least one commit message is meaningful.
2	Some progressive development evident (more than one commit) AND at least one commit message is meaningful.
3	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of all functionality AND meaningful messages for most commits.
5	Multiple commits that show progressive development of all functionality AND meaningful messages for ALL commits.

Design Documentation (10 marks) – for CSSE7231 students only

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program. This must be submitted via the Turnitin submission link on Blackboard.

Please refer to the grading criteria available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked. Note that your submission time for the whole assignment will be considered to be the later of your submission times for your zip file and your PDF design document. Any late penalty will be based on this submission time and apply to your whole assignment mark.
This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them. It must meet the following formatting requirements:

- Maximum two A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram(s) must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification – it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and will be asked to explain why.

Total Mark

Let

- F be the functionality mark for your assignment (out of 60).
- S be the automated style mark for your assignment (out of 5).
- H be the human style mark for your assignment (out of 5).
- C be the SVN commit history mark (out of 5).
- D be the documentation mark for your assignment (out of 10 for CSSE7231 students) – or 0 for CSSE2310 students.

Your total mark for the assignment will be:

$$M = F + \min\{F, S + H\} + \min\{F, C\} + \min\{F, D\}$$

out of 75 (for CSSE2310 students) or 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to `csse2310@uq.edu.au`.