Root directory includes Makefile, source code and executables under ./bin/.

Before running the program, install the uuid package with: sudo apt-get install uuid-dev and the ldap package with sudo apt install libldap2-dev

Running server and client

Running ./bin/server is required before connecting with ./bin/client.

Arguments for either program are optional, and both are written to connect to another by default.

If you want to add arguments, the commands are:
./bin/server <port>

./bin/client <ip> <port>

Used technologies

- C++/C
- Uuid package: sudo apt-get install uuid-dev
- Ldap package: sudo apt install libldap2-dev
- Libraries: queue, thread, condition_variable, atomic, string, vector, iostream, sstream, fstream, filesystem, memory, map

Development strategy and needed adaptations:

For concurrency we used a thread-pool that starts off with a set number of threadWorkers, tracks the number of active threads and has a maximum number of threads. Once the number of tasks in the taskQueue exceed the number of threads available, it increases the number of threads. However, it does not increase it if the number of active threads exceeds the max number of threads. More information below (Synchronization methods)

The other changes to the code were relatively straightforward (blacklist, ldap). The login function was already implemented in a rudimentary form in the previous hand-in, so we could easily adapt it for the final tw-mailer.

From the intermediary hand-in (TW-mailer basic) we only needed to fix a bug. Since we didn't have access to the feedback from the workshop, we couldn't account for any issues found by the other students.

Synchronization methods:

The threads only have a few shared resources: the directories, the blacklist, the taskQueue, the serverRunning Boolean and the threadPool.

The threads use a threadWorker function. A condition_variable is used, which allows the threadWorker function to wait until it is notified by the main program. Once the threadWorker is notified, it takes the next task from the taskQueue. The taskQueue is protected with a mutex.

The Boolean serverRunning is atomic, so there won't be any issues with race conditions.

The email directory itself is protected with a mutex, whenever the program checks if a user's email directory exist. Each individual email directory is also protected with a unique mutex, which are stored in a map. When a specific user does READ, DEL, LIST or SEND to another user, that specific email directory is protected with its mutex. The blacklist also has a mutex.

The taskQueue and the threadPool are both protected by mutex, so that operations (retrieving from taskQueue, changing the number of threads) is protected.

These critical areas were tested with a pre-processor directive ENABLE_MUTEX_TESTING that we wrote, which when enabled, creates delays when using the mutex and prints to the console. That way we could see that attempted simultaneous access to a shared resource was protected.

handling of large messages:

In order to handle large messages, the server checks if the max buffer size was filled (BUF 1024 minus 1). If recv returns a size of 1023, we assume that the message was not fully received, and keep using recv in a while loop, until we get a size below 1024. Meanwhile we add the message fragments to a string, to build up the message.