Passman (Password Manager)

Bursuc Serban-Mihai

Alexandru Ioan Cuza University Faculty of Computer Science, Iasi, Romania
serban.bursuc@info.uaic.ro

Abstract. Passman is an app designed to handle all user passwords in one place. From one account a user can access all their saved passwords on Passman's server. Passman has all functionality needed: adding, removing passwords in real time and sorting by category.

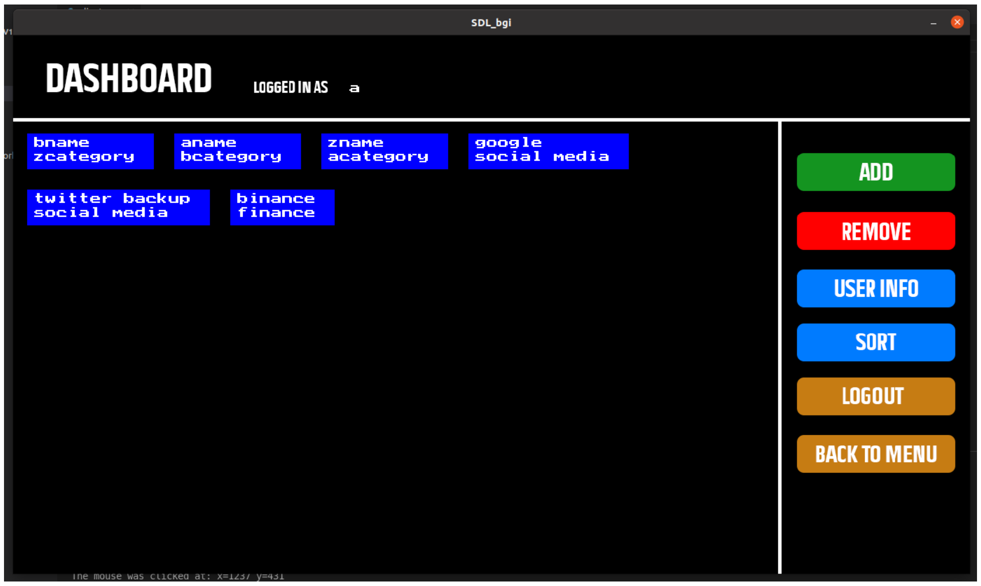
1 Introduction

The world is getting more and more digitalized. An average netizen nowadays is registered on dozens of websites. Everybody registers on their sites of interest as fast as possible without much thought. The problem however is that in order to remember all passwords, users usually sacrifice password strength, or start repeating their own passwords. This can lead to serious security issues where either a password is too weak and can easily be found or if one password of an user's account is found then the likelihood of finding the others is high.

Passman is an app designed to solve these issues. A user can store all passwords in one place and only have to remember one password in order to access all of them (the master password). This ensures that the user's passwords are in a more secure place since only the master password has to be strong, the rest can be more "user friendly".

Anyone can just open the app, register, login and begin adding passwords.

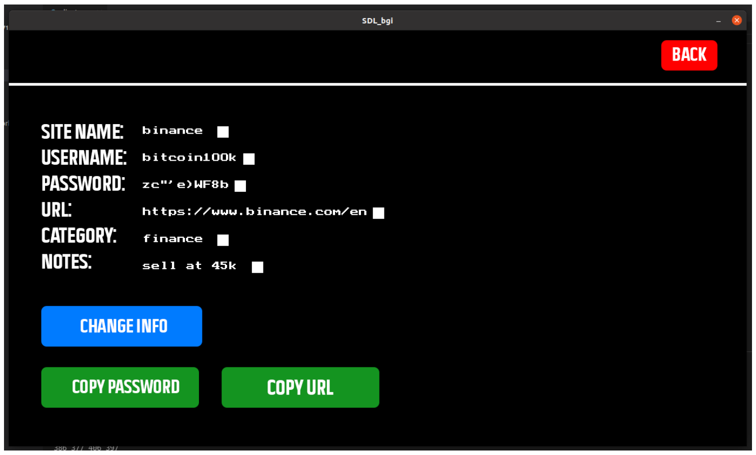
2 Features



```
bname          aname          zname          google
zcategory      bcategory      acategory      social media

twitter backup    binance
social media      finance
```

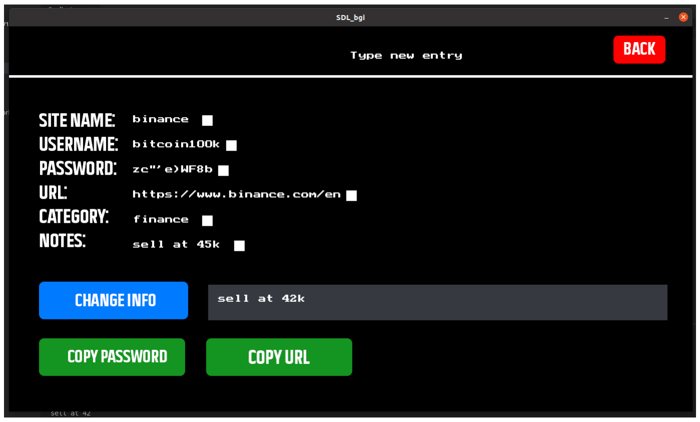
Once an user opens the app, there are 2 options, either to login or to register an account. After registration, the user can log in into the newly made account and experience a dashboard. This dashboard will display all passwords that are stored in the user's account. Here the user can add, remove, or sort the passwords by the category they were added. Clicking on a site name will display a page of information regarding that site, all introduced while adding a password. The information stored includes: the site name, the password, the URL of the site, the category (a tag given by an user, could be anything), and lastly some notes the user could enter (not mandatory).



SITE NAME:	binance
USERNAME:	bitcoin100k
PASSWORD:	zc"' e)WF8b
URL:	https://www.binance.com/en
CATEGORY:	finance
NOTES:	sell at 45k

Note: no info shown above is real. Except the notes. I think that's a selling point for BTC.

There are 3 options in a password screen: change info, copy password and copy URL. Both copy options do what they say and copy said info onto the clipboard. After clicking change info a user is then prompted to click on a square next to the field they want to change. A gray box will appear where the user can see the keyboard input for the new field. Pressing enter changes the info and sends the user back to the dashboard screen.

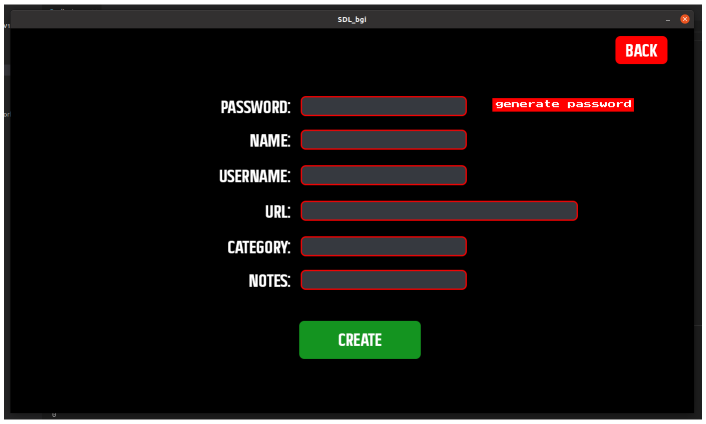


SITE NAME:	binance
USERNAME:	bitcoin100k
PASSWORD:	zc"' e)WF8b
URL:	https://www.binance.com/en
CATEGORY:	finance
NOTES:	sell at 45k

sell at 42k

Here is a quick rundown of all features:

- **ADD**: This will prompt the user to the add screen, where all necessary info is added for a password. The generate password button generates a 10 character password with random characters. All fields can be completed by using clipboard text (CTRL+V).



- **REMOVE**: Upon clicking this button the user is asked to click on a password. That said password will then be removed.

- **USER INFO**: A tab containing user time since login, the number of passwords created and the last password added for the client.
- **SORT**: Sorts all passwords based on either add order (default), alphabetically using the site's name and also alphabetically using the site's category. Clicking on the button cycles between these sorting methods.
- **(while looking at a password) CHANGE INFO**: Upon clicking the button, the user needs to click a square next to the field that needs to be changed. Immediately after doing so, they will be stuck in a text input box until ENTER is pressed. After this the field will be updated.
- **LOGUOUT**: Logs out the client.
- **BACK TO MENU**: Goes back to the main menu. If the user is still logged in, clicking on Login will go straight to the user's dashboard. Quitting the program automatically logs out the user.

3 Technologies used

The application is made in C++ using a variety of technologies.

The server is a **pre-threaded TCP server**. The graphical interface used is a Linux port of BGI based on SDL called SDL BGI.

3.1 The server

For this project a TCP server fits the purpose of the application. There are many reasons why: first of all the application does not require a constant influx of data which UDP is suited for. No constant communication is required between the client and the server. The client requests for example a login, the client checks the pending login, and either accepts it or not. There are no continuous login pendings from a single user. The connection is done over an IP and the usage of a port.

TCP also ensures that the connection is maintained at all times and that every data packet is guaranteed to not be lost. This is key because unlike scenarios where UDP is used, say in video streaming where a data can be lost and the consequences wouldn't be noticeable, any data failure will be immediately noticeable in Passman. Ranging from failed logins to not being able to see user passwords. Speed is not a concern since the data flow in this project is fairly simple; only a few bytes of information are sent at the time, mainly text data.

Being a server with multiple possible clients a concurrent server solution was needed. I chose to make a pre-threaded server (server using a thread pool) because of reliable it is. We are assured that at any given moment there are enough active threads to handle any new incoming connection. No time is spent on creating new threads, we know for sure that there are active threads looking for work. Of course this approach could cause scheduling problems. A big possible flaw could be that all threads are being blocked by the *accept()* call, causing a thundering herd problem. In practice this particular problem is solved by using *mutexes*.

Pre-threaded servers also have good performance.

The server being pre-threaded however means that once all threads accept a connection if another user requests a connection there will be no threads to answer. For this reason the server doubles the amount of threads that it started with. For example if the server started with 5 threads and 5 clients have just connected, the fifth thread will create 5 more threads and so on. Of course if all users leave then all 10 threads will remain active, but that is intentional. The philosophy behind the idea that always having double the amount of threads available as the maximum ever recorded clients in session assures the server will be more than ready to handle an influx of clients.

A database is also needed of course. For the database **text files** should be enough for this project. For example one file storing user information such as username,password and an UID, and another text file storing all user passwords. There are two text files storing information: clients.txt and passwords.txt. Clients.txt stores the username and password of a client and passwords.txt stores all password data added to a password. The text files use a CSV format but instead of a comma the separator is composed of two underscores. For fields where space can exist the space is marked by three #.

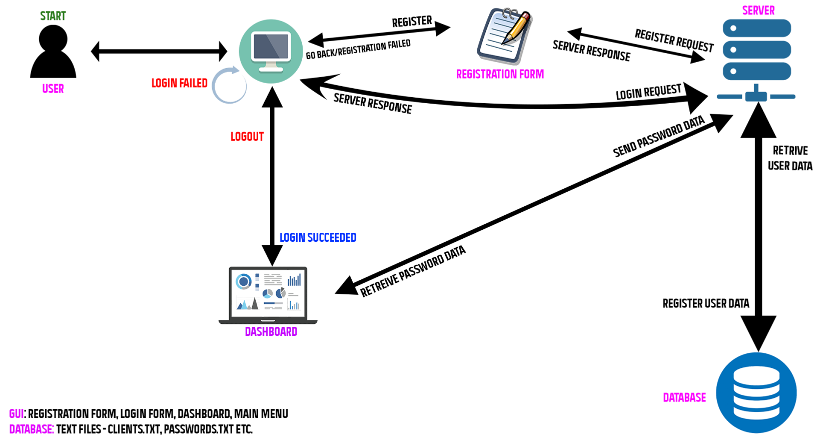
There is obviously a huge problem with this approach: for example using just one _ (due to how strtok works) will change the number of arguments on one line, which breaks the password in the database. A password "abcd_e" will only be seen as "abcd_" and e will be the next field, which in this application happens to be the site's name. Any # put on notes for example will just be seen as a space. For this reason, the program disables these 2 characters from being put by the user from the keyboard. Every other normal character should be available.

3.2 GUI

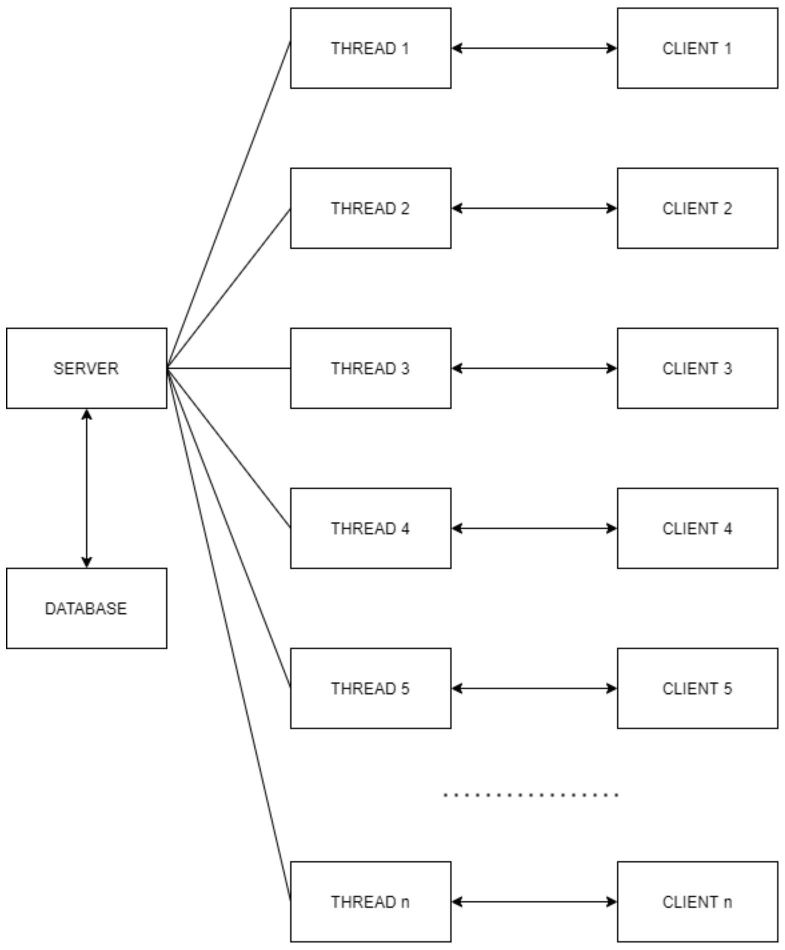
There are many graphic libraries available for Linux notably SDL, OpenGL or raylib, and this project uses a sort of simplified version of SDL. SDL BGI ports Borland Graphics Interface which was originally only available for Windows. BGI offers simple drawing tools and is very fast and straightforward. Being quite an old library however means that there are several limitations, which we will discuss later.

4 Application architecture

Below is a simple diagram of the entire application.



And this is an abstract representation of the client-server connection.



The features of this program architecture come with the features of TCP itself: the server is a full duplex server, it can receive and send data, errors are checked and known making it easier to troubleshoot, the server is "connection oriented", which is very important for this application since we want to establish a connection to the server and then start requesting data (once a side note this explains why *almost* all websites use TCP). Another important feature of TCP is that data reaches the intended destination in the same order it was sent, which guarantees no situations where say the client requests to see a password for a site, then does the same for the other site, but the server shows the data between the two sites swapped between each other.

5 Implementation details

5.1 Key points

Let's take a look at some key points in the server code.

The code below could be considered the heart of the pre-threaded server concept. We generate *nthreads* threads, a constant which we define before starting the server. After that we enter an infinite for loop. Here the *pause()* function suspends the program's executions until a signal for handling a function is received, which means the program pauses until a connection is established with a client.

```
int i;
for (i = 0; i < nthreads; i++)
  threadCreate(i);

for (;;)
{
  printf("[server]Asteptam la portul %d...\n", PORT);
  pause();
}
```

Below is the treat function, which tells each thread what to do.

```
  void *treat(void *arg)
{
  int client;

  struct sockaddr_in from;
  bzero(&from, sizeof(from));
  cout << (int)arg << endl;
  fflush(stdout);

  for (;;)
    {
        socklen_t length;
        pthread_mutex_lock(&mlock);
        printf("Thread %d trezit\n", (int)arg);
        if ((client =
        accept(sd, (struct sockaddr *)&from, &length)) < 0)
        {
            perror("[thread]Eroare la accept().\n");
        }
        clients++;
        if (clients == nthreads)
        {
            for (int i = nthreads; i < 2 * nthreads; i++)
            {
                threadCreate(i);
            }
            nthreads = 2 * nthreads;
        }
        pthread_mutex_unlock(&mlock);
        threadsPool[(int)arg].thCount++;

        raspunde(client, (int)arg); //procesarea cererii
        /* am terminat cu acest client, inchidem conexiunea */
        clients--;
        close(client);
    }
}
```

Each thread runs an infinite for loop. A mutex lock is called because like discussed earlier we don't want multiple threads to get blocked by the blocking call *accept()*. The mutex is unlocked after because no other line of code needs a mutex protection. The *answer* function will handle the logic of the application. The thread with the index *nthreads-1* will create double the current amount of threads.

5.2 Use case

The client begins by connecting its socket to the server's family socket.

```
  struct sockaddr_in server;
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = inet_addr(argv[1]);
  server.sin_port = htons(port);

  if (connect(sd,
  (struct sockaddr *)&server, sizeof(struct sockaddr)) == -1)
  {
    perror("[client]Connect() error.\n");
    return -1;
```

```
}
```

After connection, the client enters the application in the form of an infinite while loop, where the program waits for mouse clicks.

```
void mainScreen()
{
    delay(50);
    readimagefile("images/mainScreen.bmp", 0, 0, 1366, 768);
    int x, y;
    while (true)
    {
        while (!ismouseclick(WM_LBUTTONDOWN))
        {
        }
        getmouseclick(WM_LBUTTONDOWN, &x, &y);
        delay(100);
        if (x >= 558 && x <= 811 && y >= 351 && y <= 416)
            loginScreen();

        if (x >= 557 && x <= 813 && y >= 454 && y <= 516)
            registerScreen_1();

        if (x >= 557 && x <= 810 && y >= 547 && y <= 612)
            exit(0);
    }
}
```

On the server side, we bind the server's family socket with the socket declared in the server code, and then we listed to this socket for connections.

```
if (bind(sd, (struct sockaddr *)&server, sizeof(struct sockaddr)) == -1)
  {
    perror("[server]Bind() error.\n");
    return -1;
  }

  if (listen(sd, 10) == -1)
  {
    perror("[server]Listen() error.\n");
    return -1;
  }
```

Notice how the *listen* function has the second parameter 10, that is the length of the queue with pending connections. This size can be increased suiting to the server's demand.

After this the server runs the code listed in key points and handles all requests requested from the client. A good way to visualize the application process is by using the diagram at section 4.

It is worth noting that the server works with a basic console client. This is noticeable in the server's code. For example in order to modify something the syntax is modify url site_username (if there are more than 2 entries with same url) what_to_update (url,username,name...) new_entry. The actual client composes these types of messages and sends them to the server. The only thing that the server doesn't do is sorting, that is done client side because we would lose the order in which the passwords were added in the database. In hindsight there shoudld have been a "date added" camp to passwords as well.

The client application is basically like a console one except text input is done after clicking certain things, and the server response is shown on the application instead of a console. It also just looks nicer.

5.3 What could've been better?

The graphics library used here, effectively being BGI is one of the *oldest* graphics libraries there is. It hasn't been updated since 2004, although at least it is open source. With this in mind it's easy to understand the limitations of this library. In particular with this project, I wished to implement a scrollable section of passwords in the dashboard menu. JavaFX for example easily offers such a feature, despite it also being a very primitive graphics library. BGI is usable with some tweaks here and there, thanks to it being an SDL port, it means that some SDL commands work too. SDL is still being used in games. Still, BGI remains one of the easiest ways to generate graphics in C/C++, alongside with the benefits aforementioned.

Another decision I regret doing is using text files as databases. Before starting the project I told myself that it would be fast to make and still good for handling data, but in reality that was not the case. I had to copy paste a lot of code because it is not easy to read a file line by line, I never would've imagined the problem with storing in CSV format (I misunderstood how strtok works, I could've maybe had a better solution had I used some std functions). Although I have just learned the basics of SQL this very semester I was too afraid to dive into SQLite. I am not wrong, it would've been just as hard learning SQLite but the payout would've been much better at least.

And the idea that would've made the project much better was to use C++ completely. The source files are in C++ but 98% of the code is C. I was too lazy to learn how to use std::string processing because I was so used to doing C string processing. The project would've looked much cleaner with C++ due to how a single line of code usually does multiple lines from C. Classes would've been available which would've been helpful. I learn from my mistakes and whenever I have the opportunity to do something in C++ I will be using ONLY std functions. That and everything I have mentioned above.

6 Bibliography

https://www.tutorialspoint.com/data_communication_computer_network/transmission_control_protocol.htm https://www.freecodecamp.org/news/

tcp-vs-udp/ https://unixism.net/2019/04/linux-applications-performance-part-iii-preforked-s
https://en.wikipedia.org/wiki/Thundering_herd_problem http://libxbgi.
sourceforge.net https://home.cs.colorado.edu/~main/cs1300/doc/bgi/bgi.
html https://www.libsdl.org https://stackoverflow.com/questions/5970383/
difference-between-tcp-and-udp https://profs.info.uaic.ro/~computernetworks/
files/NetEx/S12/ServerPreThread/servTcpPreTh.c