Business Security Management

Matteo Gandossi
Badge No. 921173

Naveen Hosaagrahara Srinivas
Badge No. 916612

course of

ADVANCED COMPUTER PROGRAMMING

prof.
Alberto Ceselli

1 INTRODUCTION

Business Security management for today's businesses and corporations is a very complex task: a company's security officer is charged with managing the user's activities and the whole organization.

Our application is mainly divided into two parts: the user part (client) and the administrator part (server).

Here multiple clients can send requests to the server and server response to every client parallelly thanks to threading and sockets concepts [Figure 1]. Also, every client is constantly updated about every change done by the server such as new permissions, how many people are inside the room or increased security measures to a particular room.

Our Business Security Management application provides identity management where the client can sign and access to an authorized room in the organization and so on. The administrator manages every aspect relating to the management of users and rooms including the insertion, deletion, and management of individual permission

The application was developed using Java programming language and we used architectures such as Model–View–Controller (MVC), Factory and Singleton. There was also an AWK script that performs operations on the log file generated by the server, containing all the operation done by each client.

The main topics of our project are *Object orientation*, *Concurrency*, and *AWK programming*.

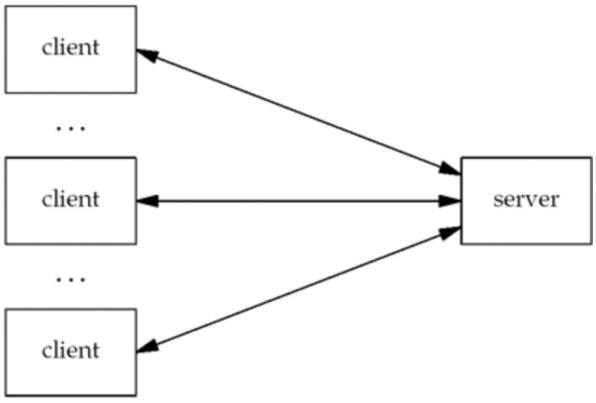

Figure 1: Multiple clients and a server

2 Object Oriented Programming (OOPs)

Object-Oriented Programming(OOP) refers to languages that use objects in programming. Object-oriented Programming aims to implement real-world entities and to bind together data and functions that operate on them and defines who can access these functions/data.

As mentioned before, we used Java as the main language and we will briefly describe some of OOP's mechanism that we used for this project.

2.1 Encapsulation

Encapsulation is the mechanism in which we can decide the *visibility* of both attributes and methods. Based on the visibility type, the data is restricted to different levels. In Java, there are four kinds of visibility:

- `public`, in which the resource is accessible to every part of the program;
- `private`, in which the resource is accessible only inside the class;
- `protected`, in which the resource is accessible only inside the package or in the subclasses;
- `friend`, it's the default type and the resource is accessible only inside the package and in the subclasses (the subclasses must be inside the package).

In this project we used encapsulation for every class, we defined all attributes as `private` and provided methods to access (getter) and/or modify (setter) them. We also defined some private methods or constructors to prevent unexpected behaviours.

2.2 Polymorphism

Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently.

We used method polymorphism, called *overloading*, by calling different behaviour methods with the same name in the same class, as long as they have different parameters in either number or type. In the project, we used overloading in class `Status`, in package `status` with the methods `permitted` that has only one parameter but with different *type* and different behaviour as well.

2.3 Inheritance

Inheritance is the mechanism in which one class, called *sub class*, is allowed to inherit the features (attributes and methods) of another class, called *super class*. This aspect is very useful for "reusability" of the code, in fact, when we wanted to create a new class and there was already a class that includes some of the code that we wanted, we can derive our new class from the existing class and adding only the new features.

In this project, we used inheritance for `RoomCount` class, which extends the `Room` class by adding only an attribute for counting people inside. We also used it for every class in both `threadClient` and `threadServer` packages in which all classes extend `Thread` class and override the `run` method.

3 Pattern

3.1 Model-View-Controller

Model-View-Controller (also called MVC) is not only a design pattern, but it's also an architecture that structure an application separating how data is stored (model), how data is shown (view) and how data is processed between storage and screen (controller).

As you can see in Figure 2, admin/user must submit his request to his controller (1); then the controller ask for the needed data to the model (2) who directly talks to the DBMS and asks for a specific query (3),then the DBMS responds (4) and the model send the answer to the controller (5). After receiving the response from the model, the controller chooses the proper view (6) to display the result and then it displays to the admin/user (7-8).

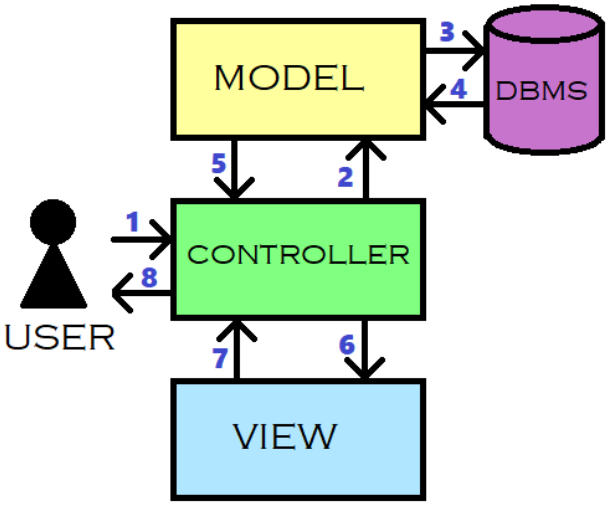

Figure 2: Model View Controller schema

3.1.1 Controller

The controller is the core of the application: it intercepts all the incoming requests from the user and performs coordination between model and view to satisfy the request.

In our project, it also kept track of the status of the whole building (only in server part), keeping the information about every client connected and in which room are they at that moment. It is also responsible for keeping all the client updated as he always provides coordination between all the threads running and handles every client request.

3.1.2 Model

The model is responsible to get data from storage: it processes the request of the controller and asks for the data to the source (the database). At this point, the data is processed in order to let the controller manipulate it and, after completing this operation, it will be sent back to the controller.

3.1.3 Database

In this project, we used as storage a local database and our application send queries like select, update, insert and delete to the DBMS and manage the storage.

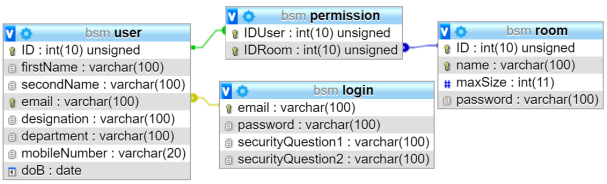

Figure 3: Database structure

It's important to remember that **only** the server controller is allowed to talk to

the database, and clients must always pass through the server in order to do any kind of operation.

Furthermore, the database is used only as a storage of information: it doesn't contain any information about the current situation of the building such as who is inside or outside the system, because this part is completely handled by the server.

3.1.4 View

As the name suggests, the view is called by the controller and it's responsible for rendering the data received from the model.

In our application, the view model has provided a good interface to both clients and server(admin) even though it's console only.

Client interface includes login, password recovery, an interactive menu that allows clients to choose to enter specific rooms and show information about accessibility.

Server interface provides an interactive menu that allows admin to manage the insertion or removing of users, rooms or permissions, change room passwords and see the status of the whole building.

3.2 Factory

Factory pattern is one of the most used design pattern in Java. It's used when we want to hide the logic of the creation of an object by providing some methods that create them. We create a private constructor, in order to prevent "illegal" creation and we create different kind of static method with different parameters that return the class object.

In this project, we used this pattern for both client and server messages: in classes `ServerMessage` and `ClientMessage` we have a lot of attributes and, for every kind of message, there are only a few needed.

For example, in class `ServerMessage`, we have two kinds of messages: accept and reject. In accept messages we need only the `UserStatus` object and the `String` is irrelevant but for reject messages the `String` is necessary because it contains the reason of the failure and the `UserStatus` object is useless because no changes occurred. So, we created the methods `createAcceptMessage` and `createRejectMessage` that requires only the useful parameters and both return the desired `ServerMessage` object.

3.3 Singleton

In object-oriented programming, a *Singleton* is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

In order to achieve this behaviour, we declare a private and static object of the singleton class. Then we make a private constructor and a static method that if the object of this singleton class is not instantiated yet, it creates it and from now on it will return always the same object.

In this project, we used this pattern for the `Log` class that writes all the clients' op-

erations on a text file. This was useful because we define only a single `FileWriter` object in order to prevent problems with mutual exclusion on the file resource, when the log features are needed in different parts of the code.

4 Concurrency

Concurrency is the ability to run several programs or parts of a program in parallel.

In our project, it's very useful for both server and client side: in server side, it' useful because it allows to satisfy requests from multiple clients and, in client side, it's useful because it provides synchronization at runtime.

4.1 Threads

The core of Java concurrency is threads. A thread is a lightweight process which has its own call stack but can access also the same data of other threads in the same process. Within a Java application, it's possible to have many threads to achieve parallel processing.

In order to achieve this behaviour, we use the default Java class called `Thread`. We create different classes that extend the `Thread` class and all of them overwrite the method `run` which is automatically called by the OS whenever the thread is started by calling the method `start`.

In Figure 4, you can see the thread schema used in this project: the yellow part indicates the threads used by the client side while the blue part indicates the threads used by the server side.

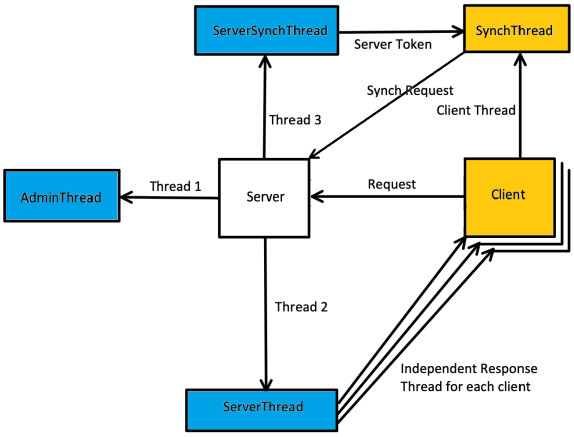

Figure 4: Thread and calls schema of the application

4.2 Server

The server side is composed by three threads: `AdminThread`, `ServerThread` and `ServerSynchThread`.

4.2.1 AdminThread

It's the main thread of the server application: it provides the interface that allows all the admin operation such as adding or removing users and room and so on.

When the program is closed by the admin, it automatically stops the other two threads. This operation was, at first, very challenging because the other threads were "trapped" in the accept waiting loop and the only way to interrupt this operation was to close the relative sockets.

4.2.2 ServerThread

This thread is responsible for taking care of the client requests such as login, enter or exit a room, logout and recovery password. It has a `ServerSocket` which is always waiting for a request. Whenever a

new request is accepted, this thread creates a new `Thread` called `ClientHandler`. This new thread is responsible for satisfying the client's request and after the response is sent, it's destroyed.

Basically, if we have n client requests, the `ServerThread` will create n different `ClientHandler` even if part of this requests are done by the same client. We decided to adopt this approach because we think that the human movements inside a working building are not very frequent so it's useless to keep active a thread for each person inside the building when, probably, the number of requests in a day are very few.

4.2.3 ServerSynchThread

This thread is responsible for taking care of the logged client synchronization requests. Like the `ServerThread`, it has a `ServerSocket` which is always waiting for a request. Whenever a new request is accepted, this thread creates a new `Thread` called `ClientSynchHandler`. This new thread is responsible for satisfying the client's synchronization request and it will respond with a server token, called `SynchStatus`, which contains all the information about the client and the list of all his accessible rooms with the number (but not the name or other information) of people inside them.

We decided to adopt the same approach of `ServerThread` because we think that the human movements inside a working building are not very frequent and also the client's permissions should not be changed very often.

4.3 Client

The client sides is composed by two threads: `UserThread`, `SynchThread`.

4.3.1 UserThread

It's the main thread of the client application: it provides the interface that allows the user to login, logout, enter or exit a room and also recovery his password if he forgot it.

This thread completely handles the `SynchTread`: it starts the `SynchThread` when the client successfully logged in and stops it when the client successfully logout from the system.

4.3.2 SynchThread

This thread is responsible to send synchronization request to the Synchronization Server: it sends the user's ID and it will receive the related server token.

After receiving the token, this thread will be put on a forced wait of fixed time (using Java `Thread` method `wait`) and then repeat the same operation.

Because it sends the user's ID, the `SynchThread` is active *only* if the client is logged into the system and it will be stopped otherwise.

5 AWK

The AWK language is a data-driven scripting language consisting of a set of actions that can process streams of textual data to extract or transform text. The language extensively uses the string data type, associative arrays, and regular expressions.

In our project, we decided to use this powerful tool to get some statistics on the user's behaviour.

5.1 Log file

As mentioned before, every time the users send a request to the server, even if it is rejected, the request is saved in a log file, called by default `log.txt`.

In Table 1, we described the fields of the log file: `param` and `description` may be not necessary depending on the operation and outcome. To avoid problems with AWK, whenever a parameter is not present or needed, we decided to use the character "-" as an empty string.

Name	Description
`timestamp`	The timestamp of the request.
`who`	The user's email that sent the request.
`action`	The action requested. It can be login, recovery, enter, exit or logout.
`param`	The parameter of the action (if needed).
`description`	Describe the reason why the request failed (not present if the request is accepted).

Table 1: Log file fields

5.2 Script

We decided to write a script that shows the overall behaviour of the users. To run this script, called `stats.awk`, the following line must be run:

```
awk -f stats.awk log.txt
```

The features provided by this script are:

- counting how many login attempts are done by the users and how many of them have a positive outcome;
- counting how many recovery password attempts are done. This is useful to detect if the system is currently targetted by some bots;
- show the most visited room;
- show the rate between accepted and rejected requests and, for the rejected, shows all the reasons why it failed and how many time this error has occurred.

In Figure 5 we can see an example of the script's outcome.

```
Log Statistic

Login
Attempted 4 successfull login over 6 in total.
Attempted recovery: 3

Room
The most visited room is PeramLab

Accepted 20 operation(s) over 29
        Accept rate: 69.0%
Rejected 9 operation(s) over 29
        Reject rate: 31.0%
All failed attempts
        full_room, 1 time(s)
        wrong_password, 2 time(s)
        wrong_security_questions, 2 time(s)
        wrong_room_password, 4 time(s)
```

Figure 5: AWK script example

6 Implementation

The project was completely developed using Java; we use a few built-in Java classes and we also write several new classes.

In this section, we are going to briefly describe every package created in the project and mention the most important classes.

- **basic**: it contains the most important classes of the system: `Room`, `User` and `Permission` and a subclass of `Room`.
- **client**: it contains the client's controller, called `UserController`, and the main class that starts the client and create the synchronization thread, called `Client`.
- **exceptions**: it contains all the classes, that extends `Exception`, used when an object of the classes in basic package couldn't be found inside the corresponding list. The classes are `UserNotFoundException`, `RoomNotFoundException` and `PermissionNotFoundException`.
- **log**: it contains only the class that generates the log file.
- **message**: it contains the message classes that are sent from the client to the server, called `ClientMessage`, and from the server to the client, called `ServerMessage`.
- **model**: it contains the `Database` class that implements all the database calls and queries and the `Model` class, which provides a layer of abstraction to the database class and perform the data translation.
- **server**: it contains the admin's controller, called `Admin`, and the main class that starts the client and create both the request and synchronization server, called `Server`.
- **status**: it contains all the classes that contains information about the status of a user (`UserStatus`), room (`RoomStatus`), the whole building (`Status`) or the update token sent to synchronize a client (`SynchStatus`).
- **threadClient**: it contains all the classes that extends Java's `Thread` class used by the client
- **threadServer**: it contains all the classes that extends Java's `Thread` class used by the server
- **view**: it contains all the classes used for rendering the information for both client and server

7 The application

As mentioned before, the application is completely done in Java's console.

In this section, we are going to show some of the features offered by our application for both server and client side.

7.1 Server

After the server starts running, it will create the three thread mentioned in section 4.2 and the main menu will be shown to the admin.

```
Select the operation...
1) See Status of the building
2) Add new User/Room/Permission
3) Remove User/Room/Permission
4) Show User/Room/Permissions
5) Add/Update/Remove Room's password

0) Quit
```

Figure 6: Server's main menu

7.1.1 Show building

This option let the admin saw the position of each logged client: they can be inside a room or at the entrance, waiting to enter one.

It also showed the list of not logged users.

```
Status of all server.
In room 'it01' (0/5):

In room 'it02' (2/6):
matteo gandossi
naveen hs
In room 'hall' (1/20):
colin fernandies
In room 'restroom1' (1/10):
gandhi mohan

Logged and at the entrance (0):

Not logged (0):
```

Figure 7: Server's show building status

7.1.2 Add new entity

If the admin chose to add a new entity, the application will ask him to choose what kind of entity he wants: user, room or permission.

```
Select what entity do you want to add...
1) User
2) Room
3) Permission
```

Figure 8: Server's add menu choice

After this, the admin needs to enter all the required parameters. In this example, we decided to show how to add a new room.

```
Add new room.
Please enter the room's name: restroom1
Please enter the room's capacity: 10
Do you want to set a password on this room?
1) Yes
0) No
Choice: 0
Operation completed successfully.
```

Figure 9: Server's add new room

7.1.3 Remove entity

This option let the admin remove any entity of the system. After deciding what entity should be removed, the application will ask to choose one of it.

```
Select what entity do you want to remove...
1) User
2) Room
3) Permission
```

Figure 10: Server's remove menu choice

This operation could cause consistency problem while the server is running: for example, if the admin removes a room while a client is still inside it, the client will be in an inconsistent state and it could cause unexpected crashes both server and client side.

In order to avoid this misbehaviour, we decided that:

- a **user** can be deleted only if he's not logged into the system;
- a **room** can only be deleted if it's empty;
- a **permission** can only be deleted if it's not used at the moment. This means that the user is not inside the room of the chosen permission.

In this example, we decided to show room deletion.

7.1.4 Show entity information

This option let the admin see one entity of the system. After deciding what entity

```
Select Room.
1) it01(0/5)
2) it02(2/6)
3) hall(1/20)
4) restroom1(1/10)
5) restroom2(0/10)
Id: 5
Operation completed successfully.
```

Figure 11: Server's remove chosen room

should be shown, the application will show all the relative information.

```
Showing user with ID 2.
Name: naveen hs
DoB: 1991-01-12
E-mail:  naveenhs@gmail.com
Designation: researcher
Department: it
The user can enter the following rooms:
it01
it02
hall
restroom1
END.
```

Figure 12: Server's show user's information

7.1.5 Manage room password

This option let the admin manage a room's password: it can be added (if not present), modified or removed (if present).

```
Select Room.
1) it01(0/5)
2) it02(1/6)
3) hall(0/20)
4) restroom1(0/10)
Id: 2
This room has a password.
Do you want to remove or update the password?
1) Remove
2) Update
```

Figure 13: Server's manage room password

7.2 Client

After the application starts, the not logged menu will be shown to the client.

```
Welcome to the BSM system.
Choose your operation.
1) Login
2) Recovery Password

0) Quit
```

Figure 14: Client's main menu

7.2.1 Login

This option let the user log into the system by providing his email and password.

If the operation is successful, the application will show the logged menu.

```
Insert your email: naveenhs@gmail.com
Insert your password: 1234
Operation accepted and completed.
Hello naveen hs!
You are inside the building.
Choose your operation.
1) Enter a room
2) Exit room
3) Show my information

0) Logout
```

Figure 15: Client's login

7.2.2 Recovery password

This option let the user recovery the login password if he forgets it. In order to change it, he needs to provide his email, his first pet name and birthplace.

```
Insert your email: gandhimohan@gmail.com
Insert your first pet's name: dog
Insert your birth place: place
Insert your new password: 1234
Operation accepted and completed.
```

Figure 16: Client's recovery password

7.2.3 Enter a room

This option is available only to logged users and not inside any other room. The application will show all the available room (based on his permissions) and ask to chose one of them. If the room requires a password, it will be asked, otherwise, the operation will go on.

```
Select the Room you want to enter.
1) it01(0/5)
2) it02(1/6)
3) hall(1/20)
4) restroom1(1/10)
Id: 2
Insert room's password: 1234
Operation accepted and completed.
Hello naveen hs!
You are in room "it02".
```

Figure 17: Client's enter a room

7.2.4 Show information

This option is available only to logged users. It will show all the user information, including all of his permissions.

```
Showing your information...
Name: naveen hs
DoB: 1991-01-12
E-mail:  naveenhs@gmail.com
Designation: researcher
Department: it
You can enter the following rooms:
it01
it02
hall
restroom1
END.
```

Figure 18: Client's show personal information

8 Conclusions

Developing this project in Java makes us learn a new programming language, we understand better how the Object orientation, Concurrency, and AWK programming work.

The project was also very challenging especially for the creation of the interface with the DBMS and for the synchronization of the sockets between the different threads.

8.1 Future implementations

In the future, we could implement a better user interface and we can make it useful for big companies. Fortunately, we use the MVC pattern which provides separation between view (user interface) and controller (the actual system) so the migration should be much easier.

We could implement the registration form for the user and their subscription should be accepted by the administrator. Users should have the possibility to ask permission to enter a room that is not accessible by their permissions list and the administrator can accept or deny the requests.

For access permission, instead of using the couple userID - roomID, we could implement groups and assign every user to different groups so the permission list could become smaller.

The tradition login part can be replaced with a badge number or biometric information of the individual.

With a few additional features, we can make it a perfect Human Resources Management System (HRMS).