



PDA Flexible Ticket System

3rd Semester Project

Team5, PROI3 IT-Project

Supervisor(s):

Svend Mortensen – DIST3

Ole Rydahl – NET3

Roger Munck Fairwood – COMP3

Student name

Student number

Signature

James Testmann

S071954

Lukas Janocko

S135272

Shicheng Dai

S133342

Sudhir Chaurasiya

S137239

Aimo Suikkanen

S082577

Abstract

The purpose of this project was to create a flexible ticket system based on using PDA.

The train/bus is able to recognize whether the customer is on the board. After the customer left the train/bus, the system is able to recognize that and charge customer from his balance that is stored on account.

The customer can check current price of journey. Using PDA, customer can also search for journey and show the fastest route to get there.

Our system is using the ring zone system based on which we can calculate the final price of the journey.

The customer has to create the account through the website where he can see the history of journeys as well as he can search for new journey. The website is also the interface for adding the money to his balance

Contents

1. Introduction	5
2. Problem Formulation	6
2.1. The Problem Boundary	6
3. Requirements Specification	7
3.1. Functional Requirements	7
3.2. Non-Functional Requirements	7
4. Problem Analysis and Solution	8
4.1. System Description	8
4.1.1 Main Server	8
4.1.2 Train Server	8
4.1.3 PDA	8
4.1.4 Website	8
4.2. System Distribution	9
4.3. Travel Plan	9
4.4. A* Algorithm	9
4.5. Database Analysis and Design	10
4.6. Use Case Diagram	11
4.7. State Chart and Activity Diagrams	16
4.7.1 Create Account	16
4.7.2 Customer Login	17
4.7.3 Search Account	18
4.7.4 Update Costumer	19
4.8. Proprietary Protocol	19
4.9. Web Application	20
4.9.1 Analyze	20
4.9.2 Why JSP/Servlet	20
4.9.3 How User Interacts with the Website	21
4.9.4 Home JSP	21
4.9.5 Sign Up Process	22
4.9.6 Login JSP	23
4.9.7 Logout JSP	24
5. Identifying Complications with the Implementation	25

6.	Problems in Detail	26
6.1.	Adding and Removing Clients to Bus/Train Server	26
6.1.1	Getting Server address.....	26
6.1.2	Connecting to the server.....	27
6.2.	Concurrent Clients	28
6.2.1	Creating Multi-Threading	29
6.2.2	Isolating Execution Threads	30
6.2.3	Implementing Shared Resources	31
6.2.4	Working towards Stateless Design.....	32
6.2.5	Speed of Client Processing	34
6.3.	Charging and Updating Concurrent Clients	35
6.3.1	Limiting Network Footprint.....	35
6.3.2	Keeping Data Reliable	35
6.4.	Finding Shortest Path.....	36
6.4.1	Designing heuristic.....	36
6.4.2	Creating the Traffic Network Graph.....	37
7.	Conclusion.....	39
7.1.	Product Conclusion	39
7.1.1	Client	39
7.1.2	Train Server	39
7.1.3	Main Server	39
7.1.4	Website	39
7.2.	Development, process, and Team Conclusion	39
7.2.1	Development Authorship.....	40
7.3.	Further Development.....	40

1. Introduction

The premise of the project is that the current travel card system developed for public transportation is flawed and must be replaced by a new and improved system, as well as facilitate a changeover to facial recognition.

The main flaw in the travel card system is the need for manual checking in and out of busses and trains with a fine for forgetting to check out.

This system punishes people for being forgetful and following their usual travel customs.

The new system must use a client's PDA or mobile device to board and debark a transport vehicle and automatically charge appropriately.

This project intends to develop a prototype for a flexible ticket system that can be used in buses and trains. The system must operate so that customers are not physically must check in and out on buses and trains. This must be done by the customer's PDA.

To communicate with the bus / train and automatically check in without the customer having to do anything. This communication will be through Bus / train WiFi connection.

Servers and train computers should be located on our virtual servers ("thelizard6") and use relevant DBMSs - MySQL. It is however acceptable to use a number of a team's portable computers as bus / train computers.

After completion of travel, the system itself could calculate the price of the journey made, and withdraw this amount from the customer Account.

During the journey the customer should always be able to see its provisional Zone structure.

It will also develop a travel plan, which customers can use to calculate the quickest route to their destination. This itinerary must be kept updated with delays and cancellations, so the customer will always get the fastest route based on the current traffic situation.

Customers should be able to log on to a website where they will be register before they can use the system. This website also allows customers to view account balance, fullness money in the account and correct Personal information.

2. Problem Formulation

Project tells the details about the new ticketing system for making the life easier to without doing physical interaction.

Public ticketing system where customers come to be recorded automatically by the buses or train. By doing this facilitates the Customers know that, they should not have to think about buying a ticket or scan in and out in each station.

For this to be done, there are a number of major issues to be answered. It includes:

- How to register customer PDAS?
- Where to users' information stored once they have been registered of a bus or a train?
- How do we handle many trains / buses close to each other?
- The customer is registered on the correct transport?
- Where and how is the payment handled?
- How is a route calculated?
- How should the total distributed so that the system is as effective as Possible?

2.1. The Problem Boundary

The system is being developed is a prototype. The goal is to establish a system.

The PDA part is not an app, but also simulated on a PC.

3. Requirements Specification

3.1. Functional Requirements

Criteria:

- Give the customer/client an easy and user experience.
- Design the system to require minimal user interactions.

Requirements	Description
R1	The customer must be able to create a user via the website.
R2	Client's PDA to registration by BTC at out and input by train/bus
R3	Customer must be able to log out /in its user
R4	The customer must be able to deposit money into their user account
R5	Client must be able to see his balance from his user account
R6	The customer must be able to correct its information on the website
R7	The client must be able see the price of its provisional travel at any time
R8	The Customer must present a ticket on his PDA
R9	The customer shall be automatically deducted from the price of the trip
R10	The customer should be able to find the fastest trip e.g. from A to B from travel plan

A customer needs to see and correct the information and requirements are designed.

So that the customer can find the information that is needed. This means that

System to meet all the requirements a customer has to make a journey.

3.2. Non-Functional Requirements

Requirements	Description
NFR1	BTC(BUS TRAIN SERVER) will be developed in Java
NFR2	Main Server to be developed in Java
NFR3	Proprietary protocol to be developed in java
NFR4	Algorithms for travel plan and price calculator to be developed in Java
NFR5	The database will be developed in MySQL
NFR6	The site should be developed with JSP and servlets
NFR7	Socket to be used for the server
NFR8	server must be on the DTU server

4. Problem Analysis and Solution

4.1. System Description

The system shall consist of six parts:

- Database
- Main server
- Bus / train computer (BTC)
- Website
- PDA
- Proprietary protocol (between BTC and PDA)

4.1.1 Main Server

- Main server to process all the requests coming from BTC.
- The server must be multithreaded so it can process multiple requests simultaneously.
- Main server are on the DTU servers.

4.1.2 Train Server

- Each bus or train will be installed with a BTC (Bus Train Computer).
- When the bus / train stops at a bus stop will BTC record all the customers who have gone on board the bus / train.
- For this communication, there is developed a proprietary protocol.
- BTC stores cached user information and sends to the main server.
- Main server verify all the customers and sends this information back to BTC.

4.1.3 PDA

PDA is the customer's smartphone / pocket computer which is used to register on buses and trains.

And show status to conductor.

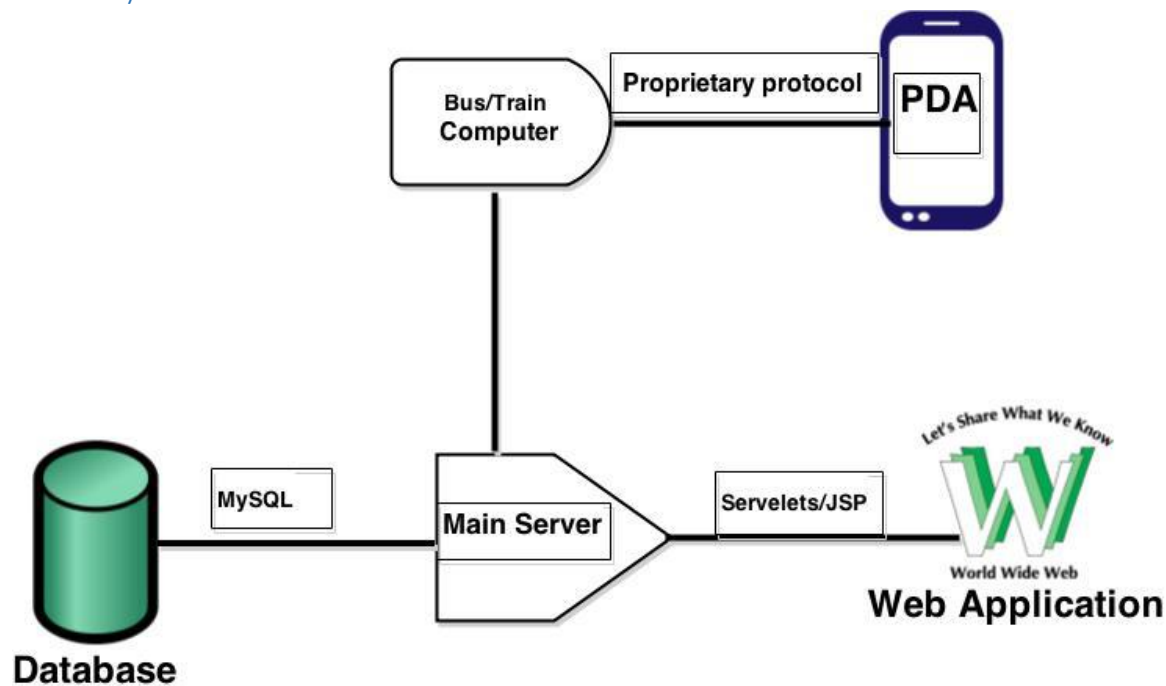
4.1.4 Website

The website is developed with JSP/Servlet (Java Server Page).

On the website, customers can create a user and once the customer has been registered successfully afterwards can be able to use the website through the process using Login system.

Change user info, see the travel history and time schedule.

4.2. System Distribution



4.3. Travel Plan

This project requires us to solve the problem by developing an application equivalent to Itinerary and this must be dynamic in relationship to the current traffic situation. The following is a summary of the design process.

4.4. A* Algorithm

Before starting the coding, the first step to find the solution is to be a suitable algorithm to find the fastest/shortest route between two points. The following are the advantages and disadvantages:

Algorithm	Advantages	Disadvantages
Best first	Always find the best node in relationship to the previous one	Not checking its position in relationship to the start node
Dijkstra	Find the distance to each node only from the starting point	Work out from the starting node and have no direction.
A*	<ul style="list-style-type: none"> -It is always ready over the distance from the starting node to the current node. -Determine the next node out from a related value (heuristic) to the ending node. 	Maybe stay visited in "dead ends"

We decided to select A* algorithm when it is a combination of Dijkstra and Best first as our way to find the fastest/shortest path. It is also an algorithm often used to find the fastest and shortest path. Although it has some disadvantages, due to most of the advantages combined together with other disadvantages of algorithm. A* is enough to handle the best because it always has track on where is the relationship

between the starting node and the ending node (estimate). When there happens a change will be the only one lead you without "problem" and find a new fastest/shortest path again.

4.5. Database Analysis and Design

As our lot of discussion to DBMS finally we reached on the specific point. The problem statement, it appears that the database must be accessible from many different applications through the server, making the design of the database is very important for instance, to avoid duplicate entries and tables are the same. The database is set up on the server set available at the DTU "thelizard6". It is made with MySQL as it is a good free database service. In the Other hands to select thelizard6, the server is preloaded with two database handlers, MySQL and PostgreSQL. The MySQL is preferred as this database is being taught in depth during the project.

To reduce strain on one database, more databases are created for the purpose having one database with the customer information, one database with the travel history of every customer and one database containing station names.

The CustomerInformationDB contains a unique customer id, the first name, last name and email. The customer id is auto generated for each added customer. A precondition is that emails are personal and unique, therefore the email is also a unique identifier, and making sure a customer is not signed up with two accounts. This gives the option of two customers with identic names can sign up without interfering with each other's account.

A customer could sing up with two different emails, and disregard the purpose of a unique email as a customer identifier. This could be prevented by using social security numbers, but in this project there is no option for requesting a check on a social security number.

The stationDB contains a name for each station, but also a unique station number, making it possible for two stations to have the same name, but still be uniquely identified.

The TravelDB is going to be the largest of the databases, containing information about date, start time, start station, end time, end station, number of zones, cost of travel, customer id and travel number.

The date and travel number are mainly used for statistics.

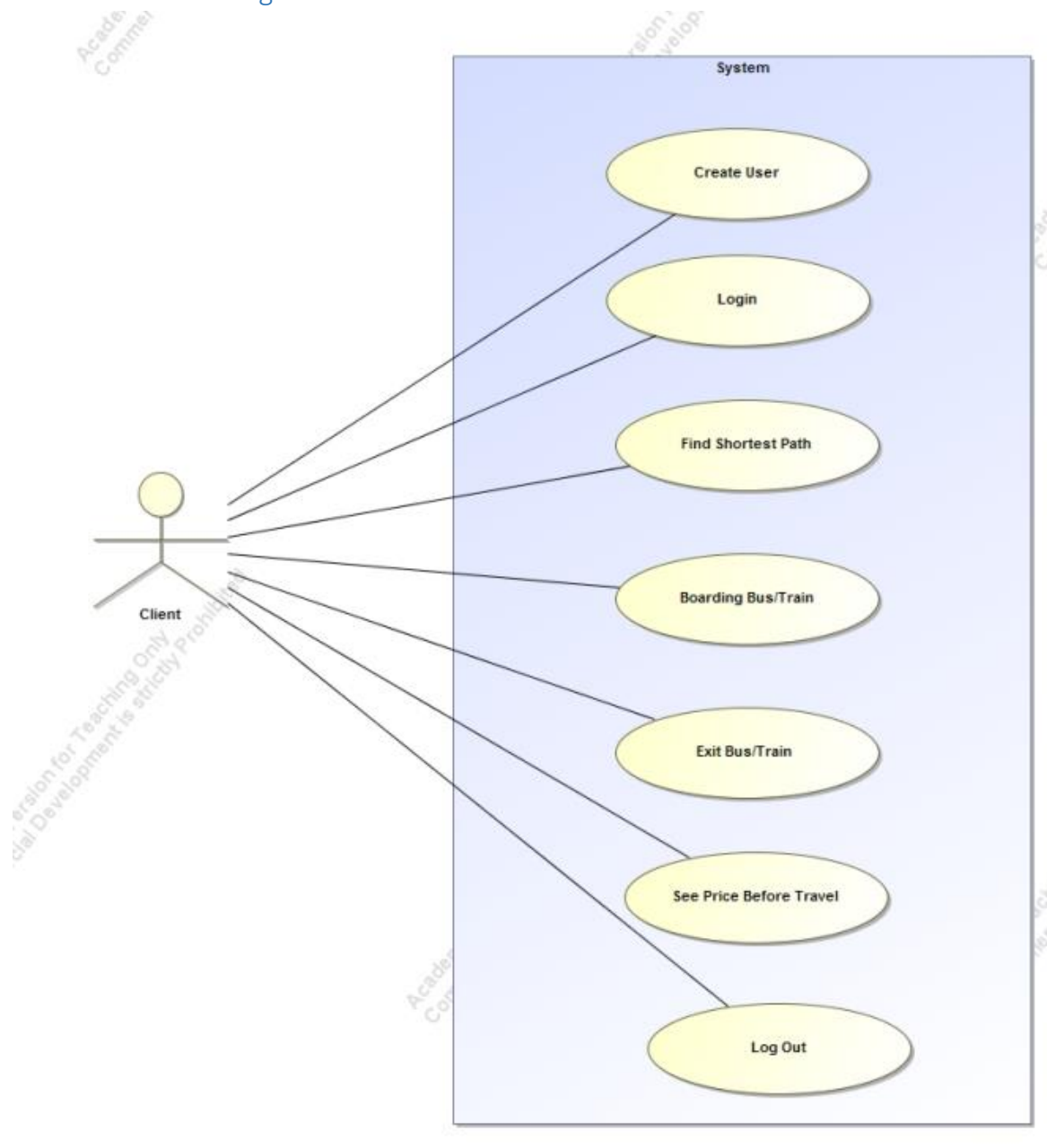
The start and end station is the station number, used to calculate the number of zones travelled.

The start and end times gives the travel duration, which is then compared to the number of zones travelled and the appropriate price can then be calculated. Zones is calculated as

$|\text{EndStation} - \text{StartStation}| = \text{Zones}$. The Price is calculated to be 20, - pr. Zone.

The customer id is used to tie a customer to a specific travel, making it easier to calculate price on demand.

4.6. Use Case Diagram



UC1: Log in
Brief description: Customer should be able to login.
Actors: Customer
Preconditions: The users can see that in the homiside of log in website.
Main flow: <ol style="list-style-type: none"> 1. This use case starts when the user wants to log into the system. 2. The correct firstName and password is entered. 3. The system checks if password is equal to the corresponding user. 4. If the password is correct, then the user has the permission to log in. 5. The user comes to the homeside.
Post conditions: Users log in after entering the right password and firstName.
Alternative flows: <ol style="list-style-type: none"> 1. The user enters the wrong code. 2. The system checks if password is equal to the corresponding user. 3. The user is told that the password is wrong and that the password must be entered again.

UC2: create User
Brief description: <i>If users have no account, users can register accounts.</i>
Actors: <i>Customers</i>
Preconditions: <i>The customer will be connected to the website</i>
Main flow: <ol style="list-style-type: none"> 1. This use case starts when a customer wants to use the travel card. 2. The customer chooses to create. 3. The customer enters their information. 4. The system stores all information in the database. 5. The system may provide instructions to the user is created
Post conditions: <i>The customer is created and the customer's data is stored in the database.</i>

UC3: Updating customer information
Brief description: The customer can be able to update customers' information.
Actors: customer
Preconditions: The customer is logged in on the website.
Main flow: <ol style="list-style-type: none"> 1. Customers can change their information. 2. The system will find the customer information from the database. 3. The system changes the information in the database. 4. Customer will be informed that the information is updated.
Post conditions: Customer has updated its information.
Alternative flows: <ol style="list-style-type: none"> 1. The customer presses cancel. 2. The customer will be notified that the update is interrupted.

UC4: Boarding the bus or train
Brief description: A customer should be automatically registered when boarding a bus or train.
Actors: <i>Customer</i>
Preconditions: The customer's PDA is enabled.
Main flow: <ol style="list-style-type: none"> 1. This use case starts when a customer embarking on a bus or train. 2. BTC detects that the client is on board. 3. The system retrieves the customer's initial travel and caches this in BTC server.
Post conditions: The customer is detected boarding on the bus or train.
Alternative flows: <p>If customer the PDA.</p> <p>Customer will not able to connect to the system.</p> <p>Customer will have to go to the further registration process.</p>

UC5: See preliminary price of travel
Brief description: Before customer starts travelling, customer can check the price of where the destination is
Actors: <i>Customer</i>
Preconditions: The customer is on a bus or train.
Main flow: 1. This use case when the customer chooses to see it's provisional. 2. Prices will be displayed on the website 3. Journey preliminary cost will be shown on the PDA
Post conditions: Customers are still on the bus or train.
Alternative flows

UC6: see balance
Brief description: A customer should be able to see his balance.
Actors: <i>Customer</i>
Preconditions: The customer is logged in to the site
Main flow: 1. This use case starts when the customer chooses to see his balance 2. The website takes the customer's balance from the database 3. The balance is displayed on the home website.
Post conditions: The customer is the home page
Alternative flows:

UC7: Find the shortest route
Brief description: A customer should be able to find the quickest route on the website
Actors: <i>Customer</i>
Preconditions: The customer is logged in to the site
Main flow: 1. This use case starts when the customer chooses to find the fastest route in the website 2. The customer selects the two stops he will soon find route 3. The website will find the fastest route and displays it on the website
Post conditions: The customer is on the website.
Alternative flows:

UC8: Viewing ticket on PDA
Brief description: When a customer is in travel, the customer must be able to show his ticket to the conductor
Actors: <i>Customer</i>
Preconditions: The customer is on a bus or train.
Main flow: 1. This use case starts when the customer chooses to see his ticket. 2. BTC server sends a valid ticket for PDA 3. The ticket can be displayed on the PDA
Post conditions: The customer is still on the bus or train.
Alternative flows:

4.7. State Chart and Activity Diagrams

4.7.1 Create Account

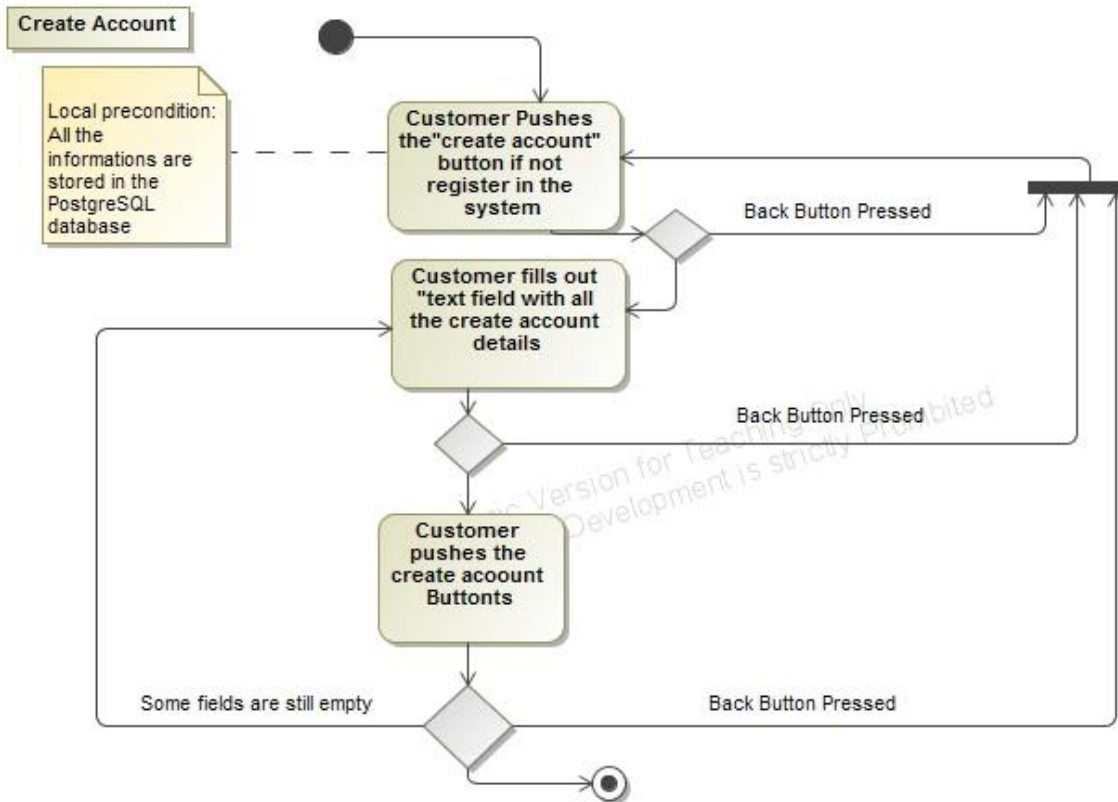


Figure 1

Figure 1 indicates that the Customer will create the account in the PostgreSQL database, according to the information provided by the customer. When all the fields has been filled, the Customer can press the "create" button, and the customer information will be stored in the PostgreSQL database.

4.7.2 Customer Login

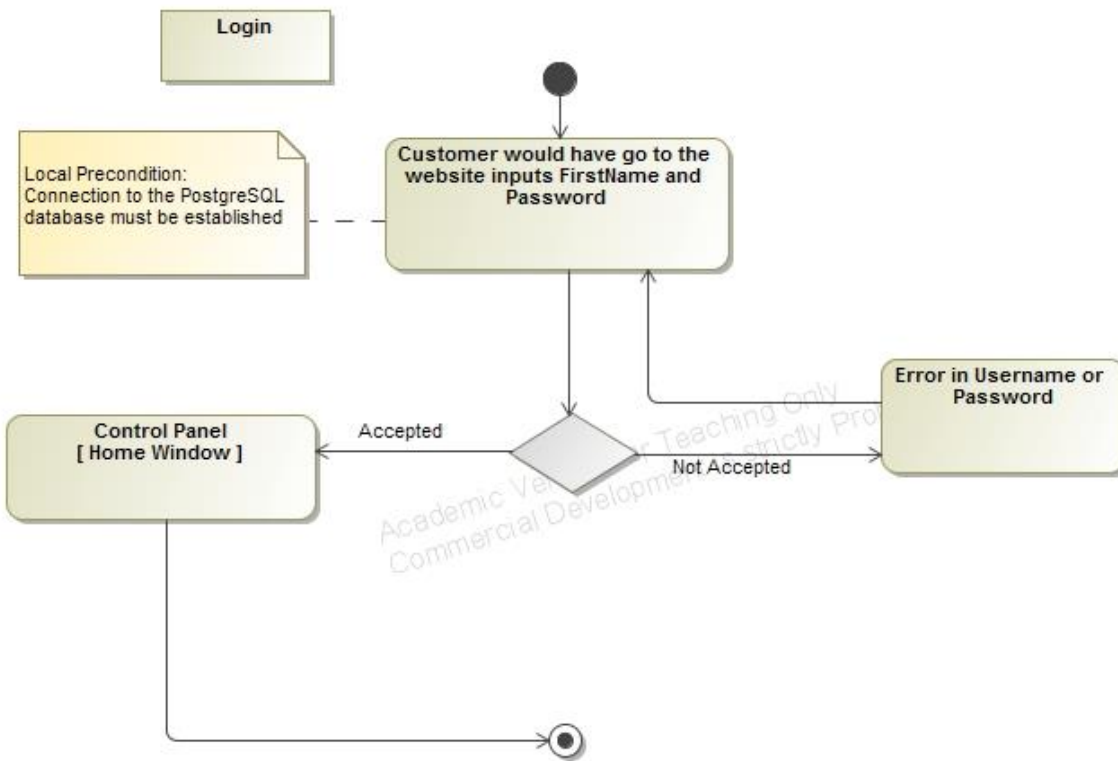


Figure 2

As figure 2 indicates, Customer can enter FirstName and password after he/she will press the "Login" button, which calls the PostgreSQL database. After a successful connection, the Customer will hereafter have access to the home window.

4.7.3 Search Account

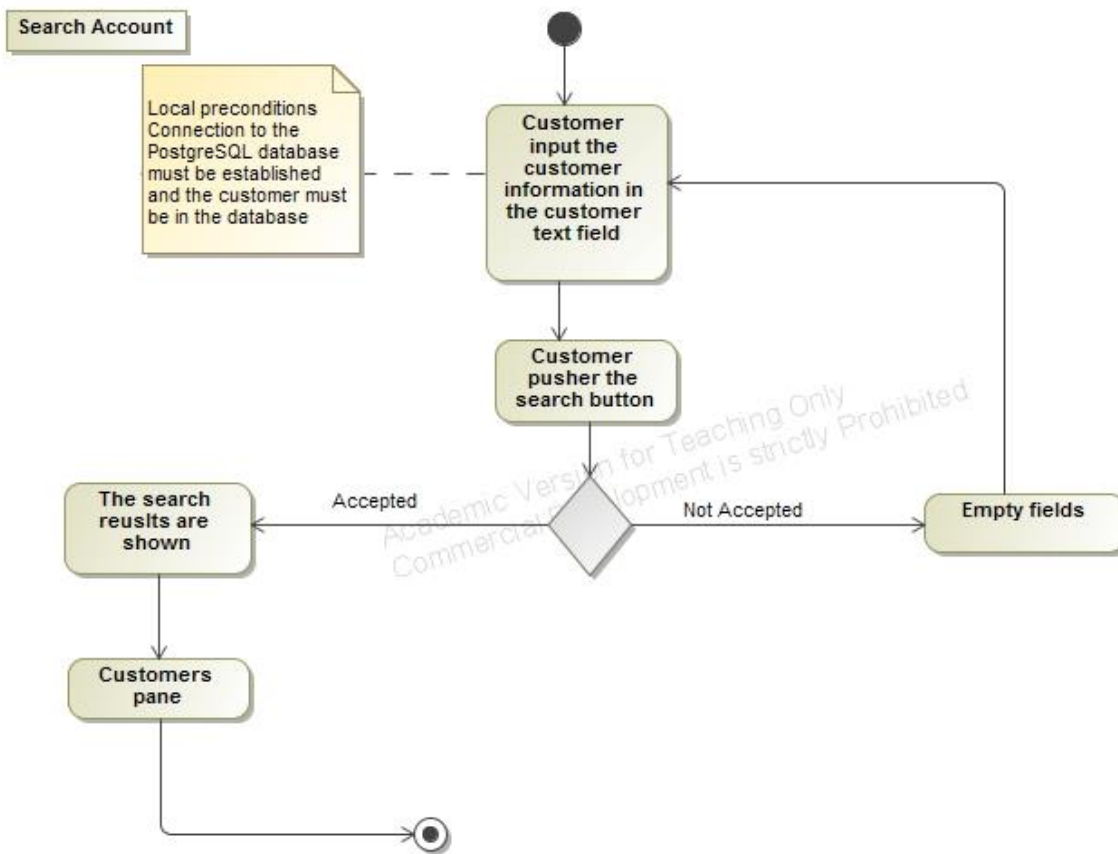


Figure 3

When a Customer wishes to search their account details. He/she will input the proper customer account details in the customer information text field, followed by a click on the "Search" button.

4.7.4 Update Customer

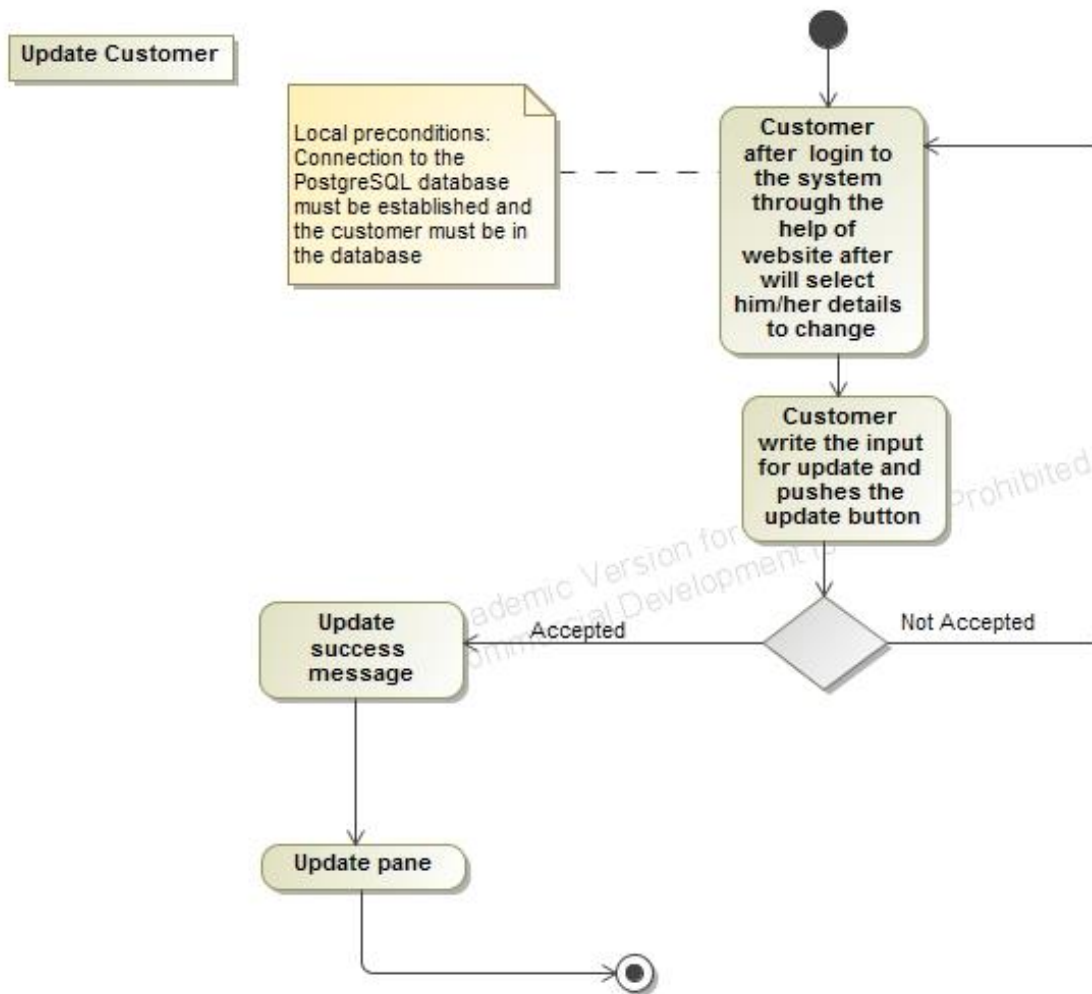


Figure 4

When a Customer wishes to update his/her information. He/she will select and change the text fields needed to change, followed by a click on the “update” button.

4.8. Proprietary Protocol

There are many factors needed to be designed to the proprietary protocol between PDA and BTC. The purpose of having this kind of protocol is to communicate between PDA and BTC and get a PDA registered as boarding on the train or bus. The first problem is how PDA makes a link for themselves on a BTC's network. Then this is decided to be used by the help of multicast. The advantage by multicast in this case is for many BTC doesn't need to know where many PDA multicast to. All PDA where wishes to receive signal can receive it, if they are only set to it. When multicast is selected is naturally to choose UDP as transport layer.

Before relationship between BTC and PDA explained, they will each especially stay short and described their roles.

4.9. Web Application

To develop the website, JSP/Servlet are chosen to be used, rather than CGI and Applets.

4.9.1 Analyze

JSP/Servlet benefits:

- -JSP is independent platform.
- -JSP is faster than other alternatives.
- -Less memory will be used.
- -Because servlet is located on a server "Inherits" the server security.
- -Servlets have built in exception act when is written in java.

4.9.2 Why JSP/Servlet

To the development of website, JSP/Servlets and java server page is selected. JSP/Servlets have a part of benefits in relationship to CGI and applets. Servlets is not dependent of which platform they are running on. The only requirement to the server is to set up to run in java. One of the main reasons to servlets is because they are faster than other alternatives. A servlet must only initialized one time and then is saved in memory, until server closed down or there happens a timeout. CGI program must be initialized each time when the program runs. This means, a servlet can handle a request by making a thread of each request. This means people can on use a single instance of servlet as then created a thread to each request. If CGI was used, each would require a request to each process, and this would require more resources and be slow compared with servlets. Servlets use themselves like applet of sandbox method which makes it safety and is better than CGI. Sandboxes makes to servlet not receive access to some areas where they could do damage. This means safety is much better compared with example CGI. A JSP/Servlet is always located on server side, and this means to the only server, they must be set up to go by java. There is no requirements to client on if jre must be installed. If applet is selected to develop the website. This would be a requirement to all clients should have jre installed. Another disadvantage by using applets would be that they takes further time to load when they have a GUI when need to be loaded unlike servlet.

4.9.3 How User Interacts with the Website

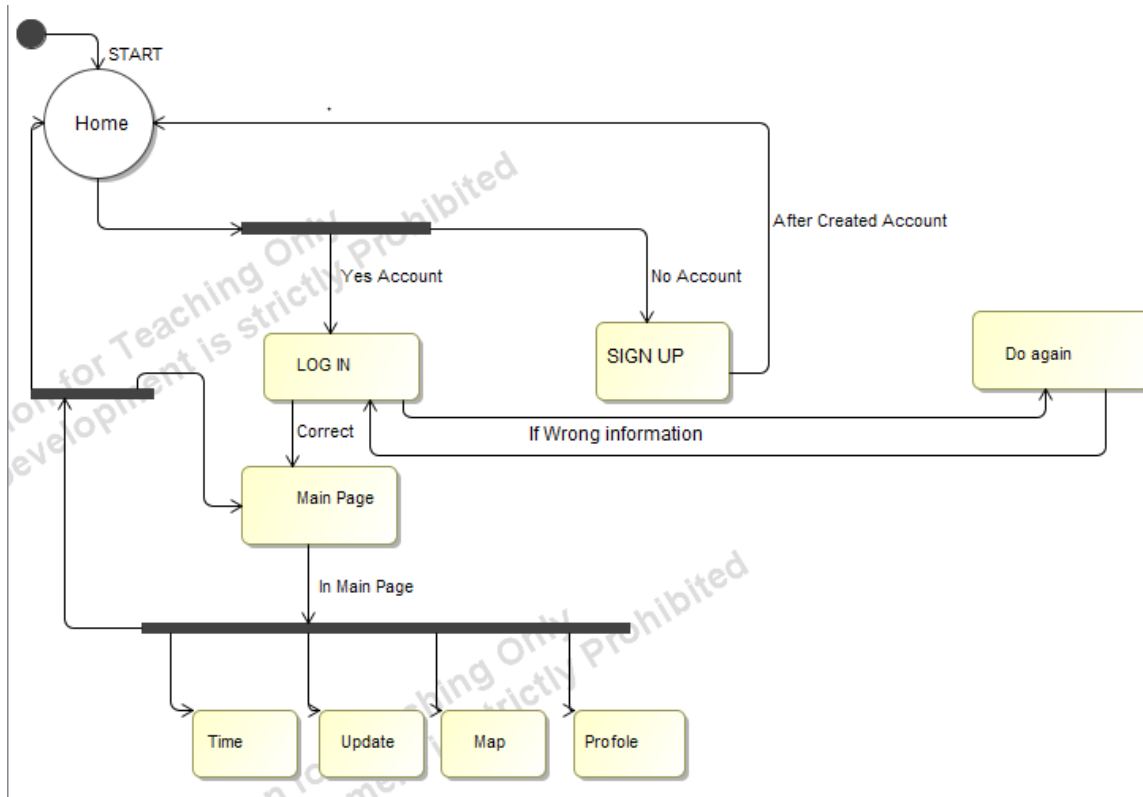


Figure 5 - State Machine Diagram

After entered the Home page. If user's have account. Then just by selecting Log in button can use the system. If user's have no account then first need to create an account. Once the account has been successfully created, go back Home window and start from the Login process.

4.9.4 Home JSP

Home JSP's purpose is to parse the user who has logged in to a session. The result in page as long as the session live (to the user logs out), it will be stayed until logged in the JSP page, the specific user fields, and other information will be displayed and manipulated.

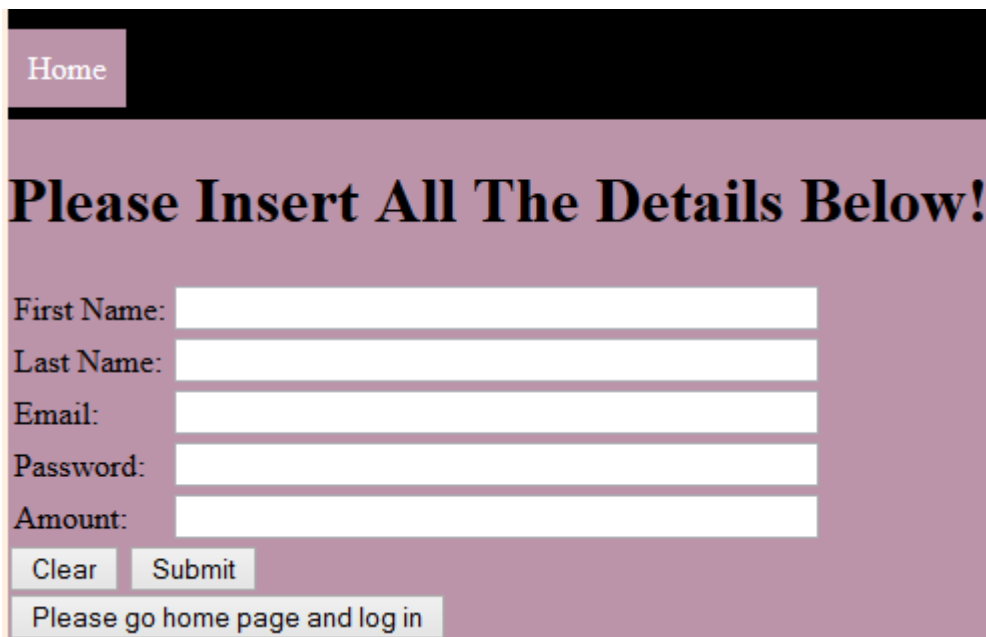


4.9.5 Sign Up Process

The purpose of having Sign up process to create the non-existing users. If user is not registered in the system then user has to create an account.

Once the account has been successfully created, the user will be able to log in the system and can be able to use the service. User shall have to follow the given rule to creating the account.

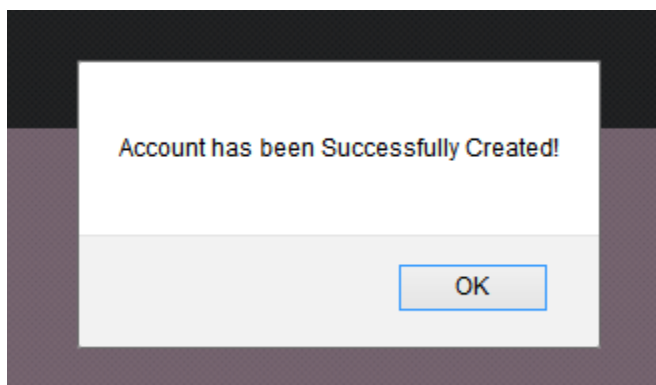
First, the user need to click the sign up button, after clicking the button the user has to fill all the required information.



A screenshot of a web application's sign-up form. The form is set against a light purple background. At the top left, there is a 'Home' button. The main heading of the form is 'Please Insert All The Details Below!'. Below this heading, there are five input fields labeled 'First Name:', 'Last Name:', 'Email:', 'Password:', and 'Amount:'. At the bottom of the form, there are two buttons: 'Clear' and 'Submit'. Below these buttons is a link that says 'Please go home page and log in'.

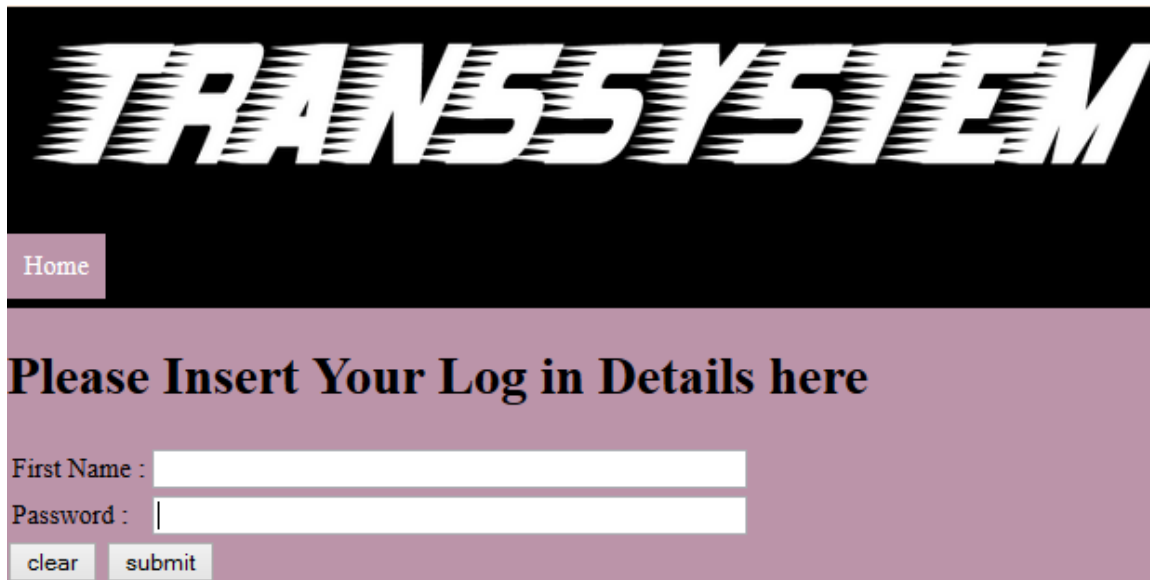
Once data of the user is correct then pop-up the message that shows the information has been inserted into database.

Like Account has been successfully created.



And afterwards, by clicking Please go home page and log in and process even further.

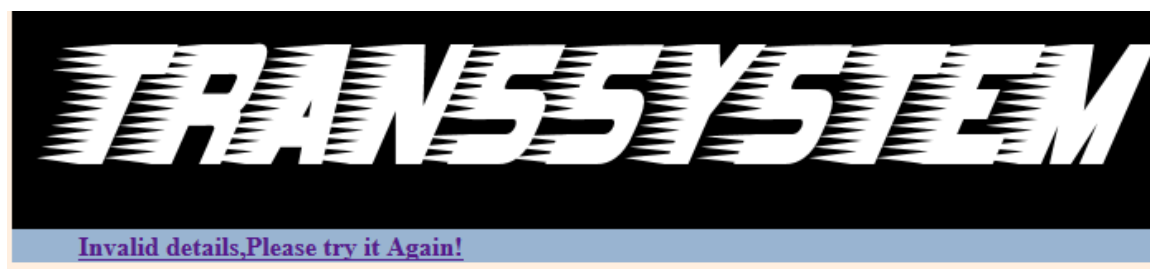
4.9.6 Login JSP



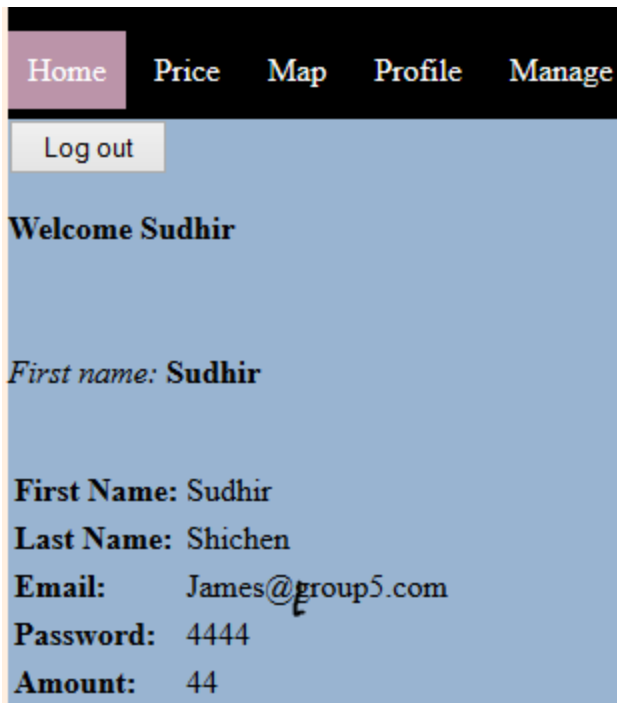
Login JSP requests a customer's firstName and self-selected password as parameters.

The values need to be entered only if customer becomes valid through Sign up process that check according to the database in order to make sure that customer enters the correct data correspondent to the database. The values will be parsed to the JSP, which will then check the strings that have been input to the predefined "string" (regular expressions). If the entered data are consistent with how the type of input should be, the customer information will be found in the database by using Customer firstName and password. Then login method called from the server. It compares the user's data information with registered data information that stored in the database.

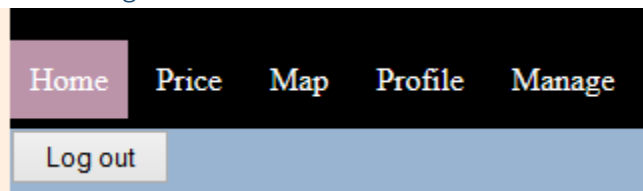
If it will not match in the database then a message will be pop-upped for you to retry, like the picture shown below,



If both of the information are matched, then the user is logged in, and will be redirected to JSP home page. As the picture shown below,



4.9.7 Logout JSP



The purpose of Logout JSP is to put the ongoing session to be disabled,

Which means that when a user chooses to log out, he cannot go back to their profile without to log in again, thus start a new session. The reason for this is to protect the user that others will not have the access to use the use's personal data without any permission which is for security protection.

5. Identifying Complications with the Implementation

In this chapter we try to identify some of the overall problems to be solved in order for the Requirements Specification to be upheld. The problems are explained at a conceptual level and any further detailing of the individual problems will follow in later chapters.

The problems we'll encounter in development are as follows:

Problem	Description	Reference
Adding Clients and removing clients to the train/bus computer	A way for the clients to logon to the transport system of their choice using their PDA, and later log out using their PDA.	Chapter 6.1 Page 26
Handling multiple clients logging on/out at once	Concurrently connecting clients to the train/bus server must be handled efficiently with little communication footprint from and to the main server.	Chapter 6.2 Page 28
Charging multiple clients when logged off	Multiple concurrent transactions with the main server to charge and update user entries from different train/bus servers and the website must be handled without risk of data loss/corruption by concurrency.	Chapter 6.3 Page 35
Finding the fastest route from point A to point B	A solution to finding the fastest path between two stations/stops using the available public transportation lines or walking.	Chapter 6.4 Page 36

6. Problems in Detail

The following chapter seeks to explore the identified complications in detail by raising more technical issues with each. Then suggest and criticize the possible solutions, and finally conclude on the chosen implementation.

6.1. Adding and Removing Clients to Bus/Train Server

The problem in detail

6.1.1 Getting Server address

PREMISE:

The user has to be able to connect with a specific transport vehicle.

ISSUES:

1. The address for each train/bus has to be unique on the network.
2. The network has to be readily available and not require extra cost on the users' part.

SOLUTIONS:

The first issue of uniqueness on the network leaves three options:

- Either using IPv6 on the Internet which includes the MAC address of the individual bus or train computer to ensure global uniqueness.
- Or use a local area network on which a range of IPv4 or IPv6 addresses can be used to connect with a range of vehicles all within range of the user.
- Or to use a local area network on which a single specific IPv4 or IPv6 can be used to connect with any vehicle. Each vehicle has a "code" that the client uses to single out a specific vehicle.

The second issue of availability and cost both speak in favor of implementing a local area network for communications.

In 2014 Wi-Fi will continue to become the primary network for smartphones¹ and even though the implementation of free Wi-Fi Internet is on the rise, there is still no guarantee for Internet connection or data provided.

On the contrary a LAN Wi-Fi connection to the train or busses would not require data cost for the user or a wireless data connection from the users service provider.

Taking the path of the local network solution, there is still a choice between a range of IP addresses each unique to a specific bus or train, or a single address with a code unique to a specific bus or train.

¹ <http://www.wired.com/2014/01/collision-course-wi-fi-first-role-changing-mobile-communications/> and <http://www.pcmag.com/article2/0,2817,2425853,00.asp>

Even though a code offers a larger range of possibilities, the number of possible IP addresses is beyond the number of possible vehicles. Therefore both solutions offer equal merit.

However, for testing purposes during development, the use of a code instead of a range of IP addresses is more feasible. Since we can test many random codes on a single computer, yet not use many random IP addresses and expect a connection.

CONCLUSION:

The bus and train server will all use the same specific IPv4 address and port number. But each implements a specific code to connect to only one unique bus/train.

6.1.2 Connecting to the server

PREMISE:

The user has to know when to connect, and perform the connection.

ISSUES:

1. The bus or train computer has to choose when to accept new connections to not get false connections between stations.
2. The user subsequently has to know when the bus or train is listening.

SOLUTIONS:

The first issue of choosing when to accept connections can be solved in three different ways:

- Closing the server socket while between stations would stop any incoming connections.
- Changing the internal server code between stops, making possible to connect but impossible to register a connection.
- Have an internal server state that, when set to “left station” denies user registration.

The first solution of closing the server socket has the issue of throwing exceptions both in the server when the blocking await function is forced to terminate, and in each client when they attempt connection.

The second solution of changing the connection code between stops would effectively deny user connections. As long as the between stops code is random, it won't be a security issue either.

Using states to solve the issue is equally effective in denying users and also without security issues. Furthermore, it has a less difficult implementation than setting a random code (since the code mustn't be unchanged, nor can it be the same as any other train or bus nearby) and offers more flexibility as using states allows for alternate connection handling between stations rather than no connections.

The second issue of knowing when to connect can be solved by either constantly attempt connection for a brief period or for the train or bus to message the user that it is listening.

The approach of bombarding the server with TCP requests is not favorable and should be avoided. The idea of having the train/bus message to all clients that they can connect is preferable. The way to do this is to multicast a UDP datagram to all listening clients and have them connect back with a TCP request.

The combination of using UDP to connect the clients along with using the flexible states implementation allows for fully automatic registration and deregistration with the train/bus server.

This is possible because the server knows whether a user is near the train/bus by sending out a UDP signal. This way it also knows whether a user has traveled with the vehicle by requesting the user to connect at two subsequent stops, if a user connects at both, then the user is actively on board.

If a user that has been active is suddenly not responding, the server can assume the client has left the train/bus and has to be charged.

The server code can furthermore be sent out with the UDP datagram, which removes the need for the client to know anything about the server TCP IP or code. Only the UDP port and IP is needed.

CONCLUSION:

The server implements a UDP transmitter to contact all potential clients near stops which sends them the needed code, and TCP IP address. The client can then connect automatically to the requesting server and go back to listening.

A user is active if it connects to the same server at two subsequent stops.

A user is charged if it used to be active and stopped replying to UDP requests.

Registrations cannot happen unwanted due to a randomized code and internal server state (whether left or arrived at station).

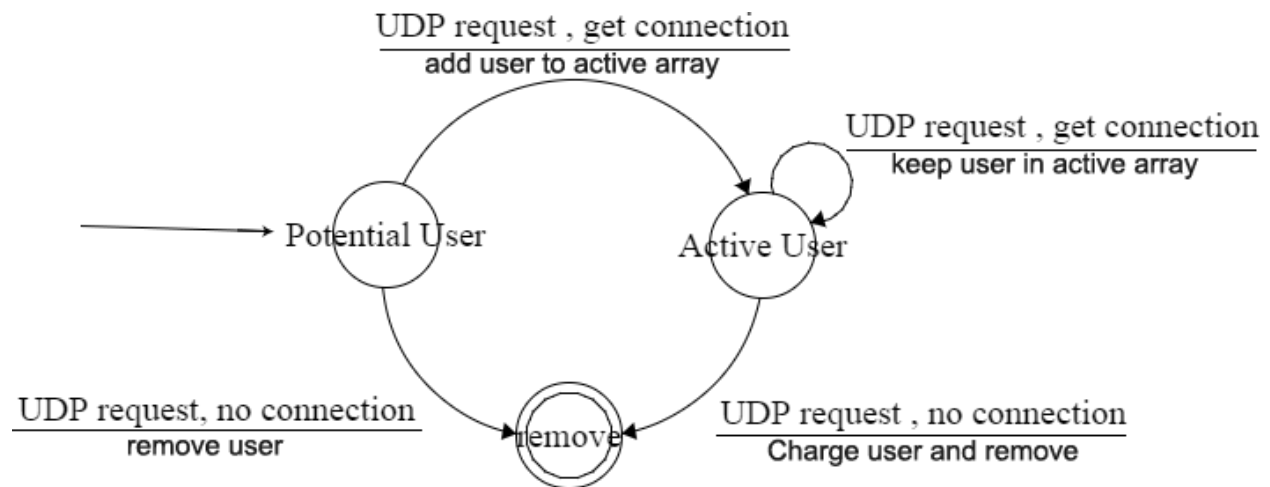


Figure 6

The figure shows a finite state diagram of the suggested solution for automated user registration and deregistration. A user registers and is added to a map of potential users, if the user replies on the next UDP transmission (at the following stop) it is moved to an active user array where it is kept until it stops responding.

6.2. Concurrent Clients

The problem in detail

6.2.1 Creating Multi-Threading

PREMISE:

The train/bus server and the main server has to start separate threads to handle client connections so new connections won't wait for a single thread.

ISSUES:

1. For every client connection, the server must start a new thread to handle its execution and reply.
2. The servers have to keep a limit on the amount of threads it can create; less than JVM's hard limit.
3. While locked at a thread limit, the server has to queue up execution calls in a schedule without interfering with the main server loops.

SOLUTIONS:

To start a new thread one has to consider whether the thread has to return a value upon finishing or not. The Java Runnable interface doesn't allow for returning an object while the Callable interface does return a Future object.

Since we're working towards a stateless communication design there should be no need to save connections in a cache, and therefore no need to return any object from the thread execution. Any getting and setting of shared resources within the server can happen using static objects instead.

For the second issue, two approaches can be used to keeping a limit on the amount of threads.

- A *fixed* number of threads waiting to accept Runnable tasks from the task queue. When a thread is ready and the queue has elements, the thread runs the task. When all threads are busy, the calls queue up waiting for available "workers".
- A *dynamic* number of threads incremented up to a certain limit whenever new Runnable tasks are scheduled without available worker threads. These threads, when created and finished, are cached and waiting for new tasks. The cached thread can be cleared after a while of steady availability.

If dynamic threads are prone to clearing after a while of availability they can be put into a stack, where after the bottom worker thread is the oldest.

To keep the main execution thread from locking up when submitting a Runnable task to a full thread pool, a Runnable objects FIFO queue with the tasks can be implemented. This always allows the newest task to be executed first by calling the poll method.

CONCLUSION:

Runnable tasks with no return objects are used to handle the incoming clients. These Runnable classes are wrapped in a request class to hold basic information like returning socket and message body.

For the purposes of creating, executing, and managing threads, the Java Executor framework offers the needed functionality. This is implemented with the default `Executor.newFixedThreadPool` with its default `ThreadFactory`.

6.2.2 Isolating Execution Threads

PREMISE:

Because threads execute independently of each other in a nondeterministic manner, it is necessary to isolate them as much as possible.

ISSUES:

1. Having multiple threads act on the same object instance creates unpredictable execution orders and corrupts non-atomic operations.
2. All client threads has to be able to request data using the same socket connection to the main server.

SOLUTIONS:

6.2.2.1 *Producer Consumer Model*

One solution for isolating threads is to push tasks between the threads and leave them free to decide when to pick up a given task, in a Producer/Consumer relationship. In this case one or several threads requests tasks to be completed while a single task handles execution of the requests.

The single consumer thread is deterministic and predictable and is therefore not liable to corrupting calls or data in the execution stack. Only the point of contact between the single consumer and the many producers needs to be thread safe. This point of contact is some form of queue to which all client threads can write and the consumer thread can read, this shared resource is discussed in the next chapter.

6.2.2.2 *Synchronized Method Calls*

The synchronized keyword introduces a lock to the scope it's attached to. All code guarded by the same lock are guaranteed to have only one thread executing through them at any given time. This makes them able to ensure that one thread requests data at any given time and the calls to the main server will not be corrupted nor will the reply packet.

The issue with using locks is that they create delays and add overhead on the executed code, because every thread must wait in a queue for the method to unlock. This expensive nature of the synchronized keyword means that it should be used sparsely and only cover small methods so executing thread can unlock it quickly.

CONCLUSION:

The implementation is a modified producer/consumer model that uses a cyclical execution with a timed delay rather than a flag for signaling data/space availability in the bound resource shared between them. This is done to collect a group of requests into one transmission rather than transmit as soon as any data is available and lessen the load on the network and main server.

More than that, the implementation doesn't require locks on the transmission method that would otherwise stop client threads from executing, and ultimately stop new clients from connecting to the bus/train server due to the thread limit.

To send requests to the server, a static server transmission class is added from which all threads can request client data or graph data from the main server.

The execution of the transmission class is as follows:

1. The transmitter thread sleeps for a set amount of time before execution.
2. The single transmitter thread tests if there are any tasks to be executed.
3. The thread pulls the available data to a local array. This allows for new data to be added to the “live array” by producer threads.
4. This array of request data is packed into a single transmission packet along with the appropriate command.
5. The packet is sent in one transmission to the main server. This keeps the communication chunky rather than chatty, and the thread count on the main server low.
6. If an error occurs, or an error (nil) message is returned from the main server, the local array is put into the original queue.

FUTURE DEVELOPMENT:

The transmission class could implement a concurrent map or priority queue as the point of contact between the producers and consumers. This would enable the potentially failed requests to get higher priority during next execution cycle.

6.2.3 Implementing Shared Resources

PREMISE:

The train/bus server has to keep a shared array of active and one of potential users to implement the automated registration process.

ISSUES:

1. Read and write corruption when two or more threads access the same shared resource.
2. Speed of operation with thread isolation.

SOLUTIONS:

It should always be the goal to write as few shared resources as possible and to limit the amount of concurrent code. In other words threads should be as independent as possible. This is because shared resources and dependent code are impractical to test and often slow to run, locking up dependencies.

Usually to implement a shared resource the producer, consumer, or an intermediary must perform a lock on the piece of shared code about to be executed. This ensures that only one thread can execute the operations at a time and additional threads must queue and wait for the resource to be unlocked.

This method is called “blocking” because it withholds the resource and stops other threads from accessing it. It’s usually implemented by writing the **synchronized** keyword on any method or field to be guarded by a lock.

In recent years several non-blocking shared resources have appeared which are taking advantage of the modern processor design to ensure read/write stability across multiple threads without locking up that resource. This keeps the execution fast and reliable with simultaneous concurrent reads and writes.

This non-locking approach is often possible because of the operation in modern CPU's typically called *Compare and Swap* (CAS) which is an optimistic approach to sharing resources. It will assume that two threads will not modify the same value often enough for it to be a problem. Instead it detects if the update happens successfully and retries if necessary.

The library `java.util.concurrent` gives a collection of such non-blocking queues to implement.

CONCLUSION:

This project takes advantage of the `ConcurrentMap` and `ConcurrentLinkedQueue` found in `java.util.concurrent` which is a non-blocking thread-safe hashmap. Retrieval operations reflect the most recent completed update.

The queue employs an efficient "wait-free" algorithm based on one described in "*Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*" by Maged M. Michael and Michael L. Scott.²

6.2.4 Working towards Stateless Design

PREMISE:

The changeover to facial recognition and potential horizontal server expansion needs to be as painless as possible.

ISSUES:

1. Since the upcoming method of registering passengers based on facial scanning will remove the need for a PDA device, the product must not depend too much on this device.
2. The number of clients can change and additional processing capacity might be needed.
3. Keeping states for each client requires additional shared resources in an already complex concurrent server.
4. Error handling with states introduces a large complexity in clean up and reverting to previous states.

SOLUTIONS:

To minimize the products dependency on the PDA, the information provided to the PDA is minimized as well down to merely knowing the ID of the user.

It is not feasible to imagine a system without any states as the phone needs to know if it's either logged in or logged out and show the corresponding state to the user. However, functionally the device is single state – a basic ID responder as shown in *Figure 7*.

The trainserver is stateless and will treat any incoming packet in the same way. It then switches execution method based on the command provided in the specific packet.

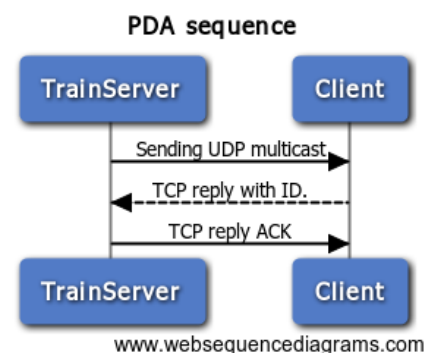
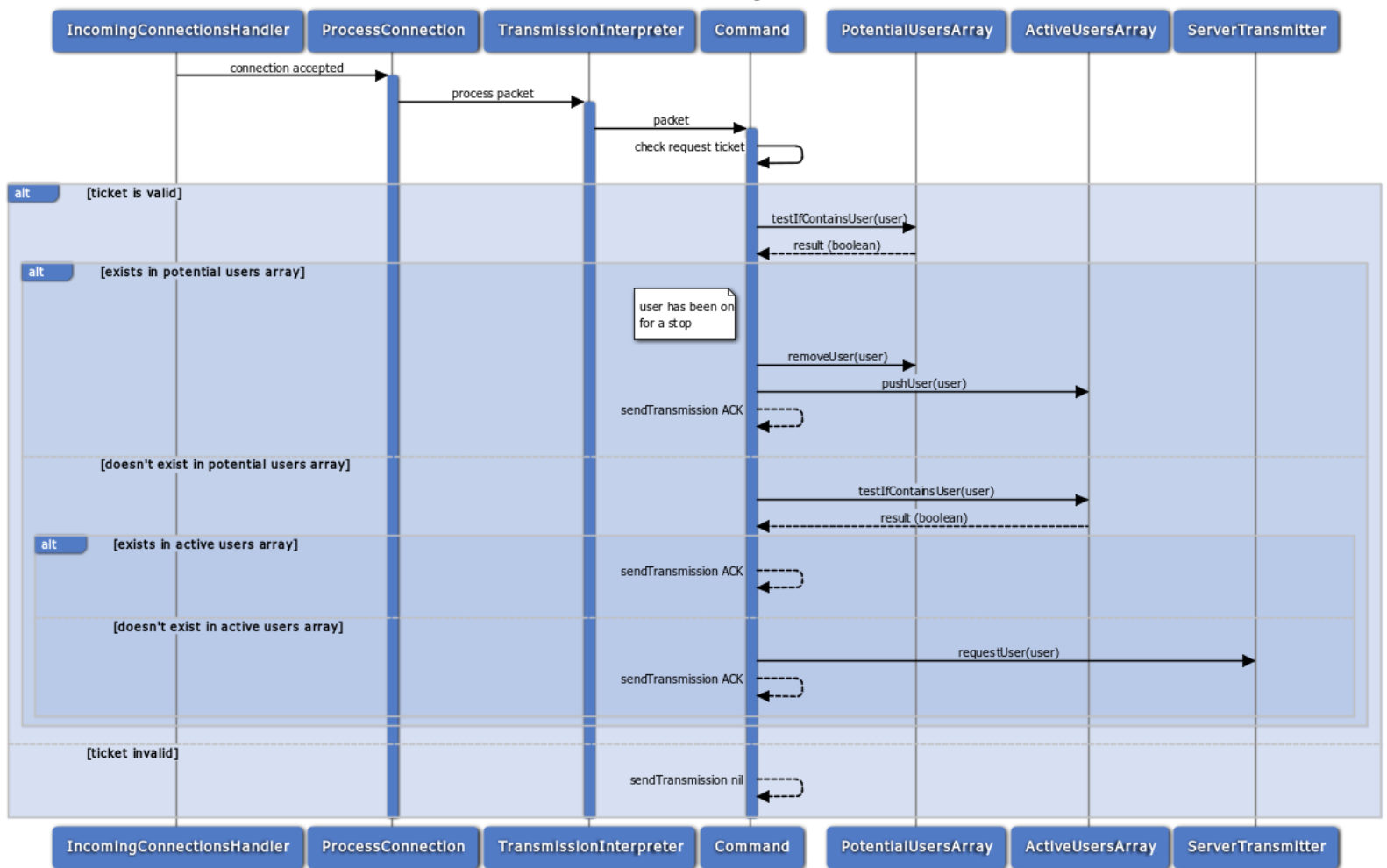


Figure 7

² <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html>

This means any packet could be sent to any identical server at any random time and be treated the same way. This design is compliable, expandable, and reliable.

Figure 8 shows the sequence for receiving a client connection.
Train Server Incoming TCP



www.websequencediagrams.com

Figure 8

CONCLUSION:

Designing the protocol to use commands instead of states for each TCP connection allows the system to be fully scalable. It also lessens the complexity of error handling and incidentally fits well with the concept of a minimalized PDA.

6.2.5 Speed of Client Processing

PREMISE:

The system has to handle a large number of clients at each station. Not only those onboard the train but also those in the close vicinity of the train.

ISSUES:

1. The number of simultaneous client threads processed by the train computer is less than the expected number of passengers able to board (312 seated + 360 standing passengers in an 8-car train³ not counting the potential passengers on the platforms). Therefore a full cycle of all threads must be processed a multiple of times in the seconds it takes the train to stop and wait.
2. The speed issue requires the server to finish and close the client connection before getting a confirmation reply from the main server and database.

SOLUTIONS:

One solution to the many clients problem can be to accept the clients with x number of threads, add them to a queue of clients, and then process the queue with y number of threads. This would be an implementation of the producer/consumer model.

This solution would keep a minimal wait time between clients getting accepted but lessens the amount of threads available for processing each request.

This might make the process take longer, but the extra processing time would occur after the train has left the station and will therefore not affect the efficiency of the solution significantly. Furthermore, this solution enables the client connection to remain open even in a stateless design until the reply from the main server has been received.

Another, more optimistic solution, is to keep the processing time for each request so low that the time taken is negligible. If a request takes 8ms the system can roughly handle a theoretical 5000 requests in a probable 10 second wait at the platform with a 4 core processor.

A combination of the two can be imagined where a producer/consumer model is implemented and the producers (threads accepting clients) will start consuming requests when there are no incoming requests. This could be implemented with a concurrent priority queue giving higher priorities to incoming clients.

CONCLUSION:

The train server will be keeping the processing time short by closing the client connection before receiving a main server reply. This way, enough clients can be processed before the train leaves the station.

Note: Making the server stateful can add the ability to contact a potentially failed client and ask it to resend or to give an error to the user depending on the reply from the main server and database.

³ Siemens: 8-vogns S-tog litra SA, page 1: "Tekniske data"

6.3. Charging and Updating Concurrent Clients

The problem in detail

6.3.1 Limiting Network Footprint

PREMISE:

The train server needs to confirm and charge each user with the main server as well as collect the most recent traffic network graph.

ISSUES:

Several trains and busses accessing the main server each with several client requests will flood the main server with requests and will severely limit accessibility on the server due to lack of threads.

Many TCP requests adds an unnecessary amount of traffic on the internet and potentially congesting the routers close to the main server.

SOLUTIONS:

The way to lower the network footprint and helping the Main server from queuing up too many threads is by sending larger chunks of data in one transmission.

This can be done by queuing up requests locally and send the entire queue as one packet. This queue can then be read at the destination and a reply queue can be sent back. The queue can either be sent with a timer or after a request count has been reached.

Using a request count as a trigger for pushing the request will also need a timeout in case too few requests are needed. This will avoid a potential livelock with this method.

Using a timed event to send the requests requires a single thread to cyclically sleep, collect, and send if there is any data.

CONCLUSION:

This product uses a timed cyclical event to collect and send requests if any data was collected. This takes up a single thread on the server, but frees up the client threads' responsibility to send anything on and wait for the reply.

This single thread is also able to roll back all failed transmissions in one operation in case of returned error packet or thread exception.

6.3.2 Keeping Data Reliable

PREMISE:

Charging and updating the same client multiple times has to perform those operations without fault.

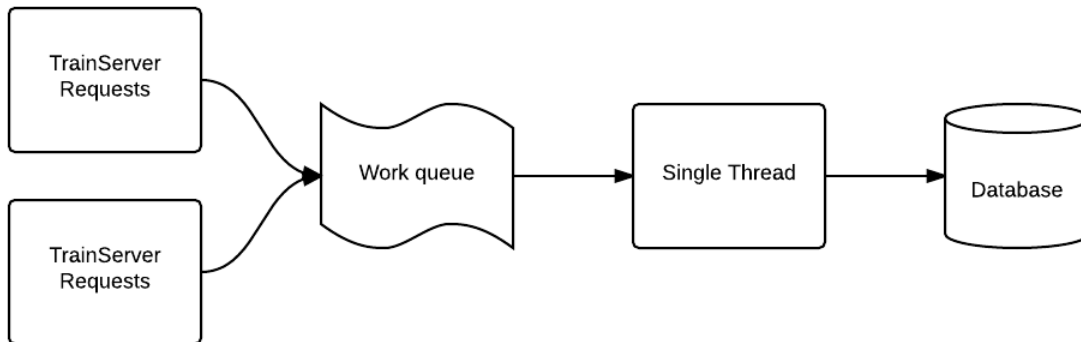
ISSUES:

1. Updating a client after a charge operation can result in deleting the charge made by overriding with the update amount.
2. Writing to the same client (User) object concurrently can result in faulty writes, which could alter the client's balance in unpredictable ways.

3. The client objects must not have any copies. A single user ID has one and only one client object.

SOLUTION:

To make sure all faulty reads and writes are eliminated, a producer/consumer model is implemented. Every request to update or charge the user is the “producers” and a single consumer thread will cyclically collect and process the requests after a timeout has elapsed – in the same way the requests are queued and sent from the trainserver.



To make sure that no update operation can override the charge operations performed on a client, the single worker thread will first perform all updates and then perform all charges. This way a charge cannot be overridden, and a charge is always an additive operation – and so will not directly override the updates done.

To make sure only one copy of each client exists a map is implemented with the user IDs as the keys to find the user objects and a check to see if the key exists before adding/getting the user respectively.

6.4. Finding Shortest Path

The problem in detail

6.4.1 Designing heuristic

PREMISE:

To speed up the pathfinding, a heuristic can be implemented to guess at a best search path to the goal.

ISSUES:

1. A heuristic must be admissible. Meaning that it must not overestimate the cost of travelling from the searched nodes to the destination. Or in other words, the search heuristic must be optimistic.
2. The heuristic must use one and the same unit to calculate the cost of every node and every edge. This unit can be speed, distance, etc. but cannot be variable.

SOLUTIONS:

There are three obvious candidates for heuristics in this case:

- Distance
- Speed
- Time

Using distance as the heuristic does not take the different transportation's velocity into consideration and a bus going from non-stop from A to B will score the same as a train non-stop from A to B. Moreover, using distance as the heuristic will often not consider backtracking to a train station for long distance travel even if this trip might be faster. Using distance will therefore not be fully admissible even if implementing a direct Euclidian distance as the node to node score.

Using speed will take all these cases into consideration when calculating the optimal path, and a train station will be sought out faster than a bus stop.

Using speed is admissible if the direct distance between stops is used to calculate the edge score, and the maximum speed of any transportation vehicle is used to calculate the node to target score.

Lastly using time as the heuristic is as admissible as speed, for the same reasons. But it allows for direct implementation of timetables in the scores.

CONCLUSION:

The heuristic for each node is based on time derived from the maximum speed of any transportation vehicle multiplied by an approximation of the Euclidian distance (approximated due to the earth's curvature)

The heuristic for each edge is based on time derived from the speed of the individual vehicle multiplied by the approximated Euclidian distance between the two edge nodes.

6.4.2 Creating the Traffic Network Graph

PREMISE:

There is a need for a graph to search through, maintained on a database as a list of stops and routes.

ISSUES:

1. Calculating distances between stops.
2. Creating a relation between stops and routes.
3. Making the graph in a single data structure.
4. Weighing the graph.

SOLUTION:

Calculating the distance between stops required the positions for each stop. These positions are found on google maps as a longitude and latitude coordinate. The distance formula between two sphere (earth) positions require processor costly operations such as square root several times for each distance.

A close approximation called “Equirectangular projection” can be used to significantly speed up the calculation times.

To create the relations between available stops and the specific routes at each stop, as well as the edges generated by these routes in both directions is made by inner joining a relation table with a stops/stations table, and that result inner joined with a routes table for each stop in the stops/stations table.

This results in a table that contains all information about the bus/train routes available at the specific stop as well as destination stops for each edge.

This information can then be used to write a collection of edges for each node.

When all nodes with edges are added to a map with their ID as key, and can then be used to create a weighted graph once a start and target node is selected.

Weighing the graph is a process of going through each node in the map, giving it a destination to target score, then looking at each edge in the node and scoring those with the distance between the two edge nodes and the transport method for that edge. All this information is found within the Edge class fields.

7. Conclusion

7.1. Product Conclusion

The product in its current form as of publication has the following functionality:

7.1.1 Client

- Clients can turn on their PDA and run the listener application as the only interaction necessary to connect, register, and travel with public transport.

7.1.2 Train Server

- Is able request to multiple unknown users to connect and send their ID number.
- Can handle concurrent incoming client connections and register, maintain, or charge them as needed depending on whether they're onboard or departed from the vehicle.
- Will collect and request data from the main server in a "chunky" rather than "chatty" way.
- Can calculate the user's charge amount based on ring zones travelled.
- Can calculate the shortest path from node A to node B using A* pathfinding algorithm and speed heuristic for the different transport methods.

7.1.3 Main Server

- Can create/update/delete/get users from an external MySQL database.
- Can collect traffic stop and traffic route information from a MySQL database and create a directed graph from the information.
- Can charge users in in the database with a reliable producer/consumer method.

7.1.4 Website

- Can create and update user accounts.
- Can show customer status on recent travels.

7.2. Development, process, and Team Conclusion

Throughout the project the team has been suffering from a struggling communication and a skewed workload. This seems to indicate the lack of planning that got only got worse later on as group work faltered.

Initially there was a proposal to use SCRUM as a way of implementing a dynamic and fast workflow where everyone had a say in which tasks we wanted and a product owner to manage the timing of the stories produced. With a weekly SCRUM standup meeting every Tuesday and supervisor meetings every Friday.

However, after the first meeting with little work prepared and barely any feedback amongst team members, the idea fell apart to quiet isolation rather than teamwork. This is when the suggestion to do extreme programming in pairs was born as a way to force the de-isolation (two groups of two and one single developer). Yet the group still had difficulty maintaining communication between pairs.

The team decided early on that it was important to share all produced work with the group and using a versioning tool like GitHub was ideal for the task. Yet months into development some were still having problems making it work or even pulling submitted files from the server.

This highlights the fact that we should, within the group, focus on developing a more thorough “Best Practices” document to explain the necessary functionality for our tools to work.

7.2.1 Development Authorship

Here follows a list of all report sections and one of all product sections and the author(s) or each.

7.2.1.1 Report

Chapter	Author	Comments
Introduction	Lukas	
Problem Formulation	Shicheng/Sudhir/James	
Requirements Specification	Shicheng/Sudhir	
Problem Analysis and Solution	Shicheng/Sudhir	
Identifying Complications with the Implementation	James	
Problems in Detail	James	
Conclusion	James	
Appendix – Deprecated Code	Shicheng/Sudhir/James	

7.2.1.2 Product

Feature	Author	Comments
Client-Server Communication	James	
Managing Incoming Connections with a Thread Pool	James	
New Passenger Search with UDP multicast	James	
Processing Client Requests Using Commands	James	
Trip Finding Algorithm using A* and Fibonacci Heap	James	
Cost Algorithm using Ring-Zone System to Charge Users	James	
Batching Data for Minimal Network Footprint	James	
Database Handling and Queries	James	
Unit Tests	James	

*See Appendix for deprecated code, Authors, and Author comments.

7.3. Further Development

There are currently a number of features missing from the product. Here is a compiled list of unsolved problems.

- The website needs to implement a method for getting the fastest trip from point A to B. The method for finding the journey is made, the website just needs to get that data.
- The website needs to implement the Main Server’s database handling to let the database handle transaction isolation.
- The trip planner needs to use transport timetables as a part of its heuristic and weight scores.

Appendix I - Deprecated Code

Database Handling

File: *APPENDIX\Deprecated Code Snippets\DatabaseConnection - SS.rar*

Authors: *Shicheng and Sudhir*

Authors note:

For database connection part, at the beginning, we divided the work for two parts, we implemented the database connection in our own way. But for some reason, one of the group's members implement the database connection in his way, but the principles between the two non-big-difference databases are the same. Meanwhile our implementation in our database connection was also working.

Dijkstra PathFinding

File: *APPENDIX\Deprecated Code Snippets\ DijkstraTraversal.java*

Authors: *James*

Authors note:

Before settling on using A* as the pathfinding algorithm, we used Dijkstra. It was assumed that a simpler pathfinding algorithm could be faster than a heuristic because the network graph to search through is very static and can be sorted before searching. The overhead on A* was not worth the heuristic.

Yet it was later found a requirement to implement a heuristic and the assumption about speed was never explored further.

ThreadQueue

File: *APPENDIX\Deprecated Code Snippets\ThreadQueue.java*

Authors: *James*

Authors note:

An initial attempt to design a smart thread queue. It was supposed to batch up jobs in a queue and create worker threads to handle them in a similar way to the now implemented ExecutorService.

It was abandoned for the standard Java implementation. Standard implementations are often fast and reliable, and was simple to use.

Concurrent Collection Tests

File: *APPENDIX\Deprecated Code Snippets\ConcurrentCollection Tests.rar*

Authors: *James*

Authors note:

One of many test projects created during development to create and analyze code in isolation. This specific test is for the concurrent collection found in Java.util.concurrent. It was created as a process speed test.

RMI

File: *APPENDIX\Deprecated Code Snippets\RMI.rar*

Authors: *Aimo and Lukas*

Authors note:

Was developed as an alternative to the implemented communication between the Train server and Main server, as well as between the web server and the Main server.