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

# Introduction

This is one of the most important components of the report. It should begin with a clear statement of what the project is about so that the nature and scope of the project can be understood by a lay reader. It should summarise everything you set out to achieve, provide a clear summary of the project's background, relevance and main contributions. The introduction should set the context for the project and should provide the reader with a summary of the key things to look out for in the remainder of the report. When detailing the contributions it is helpful to provide pointers to the section(s) of the report that provide the relevant technical details. The introduction itself should be largely non-technical. It is useful to state the main objectives of the project as part of the introduction. However, avoid the temptation to list low-level objectives one after another in the introduction and then later, in the evaluation section (see below), say reference to like "All the objectives of the project have been met...".

# 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 3.1  Page 5 |
| 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  Page 7 |
| 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 3.3  Page 12 |
| 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 3.4  Page 13 |
| Creating a website for user management and route planning |  |  |

# 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.

## Adding and Removing Clients to Bus/Train Server

The problem in detail

### 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 smartphones[[1]](#footnote-1) 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.

Even though a code offers a larger range or 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.

### 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 chancing 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 than 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).

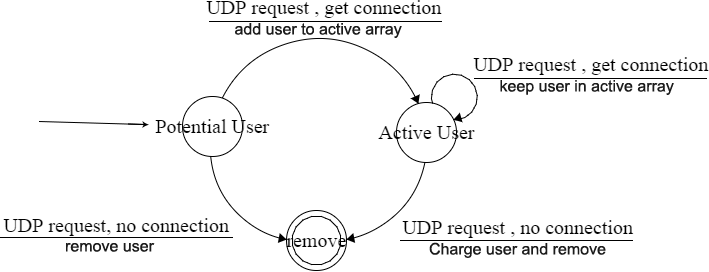


Figure 1

*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.

## Concurrent Clients

The problem in detail

### 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 on 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.

### 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:

#### 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.

#### 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.

### 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]](#footnote-2)

### 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.

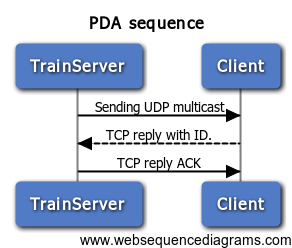
Solutions:

Figure 2

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 2*.

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.

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.

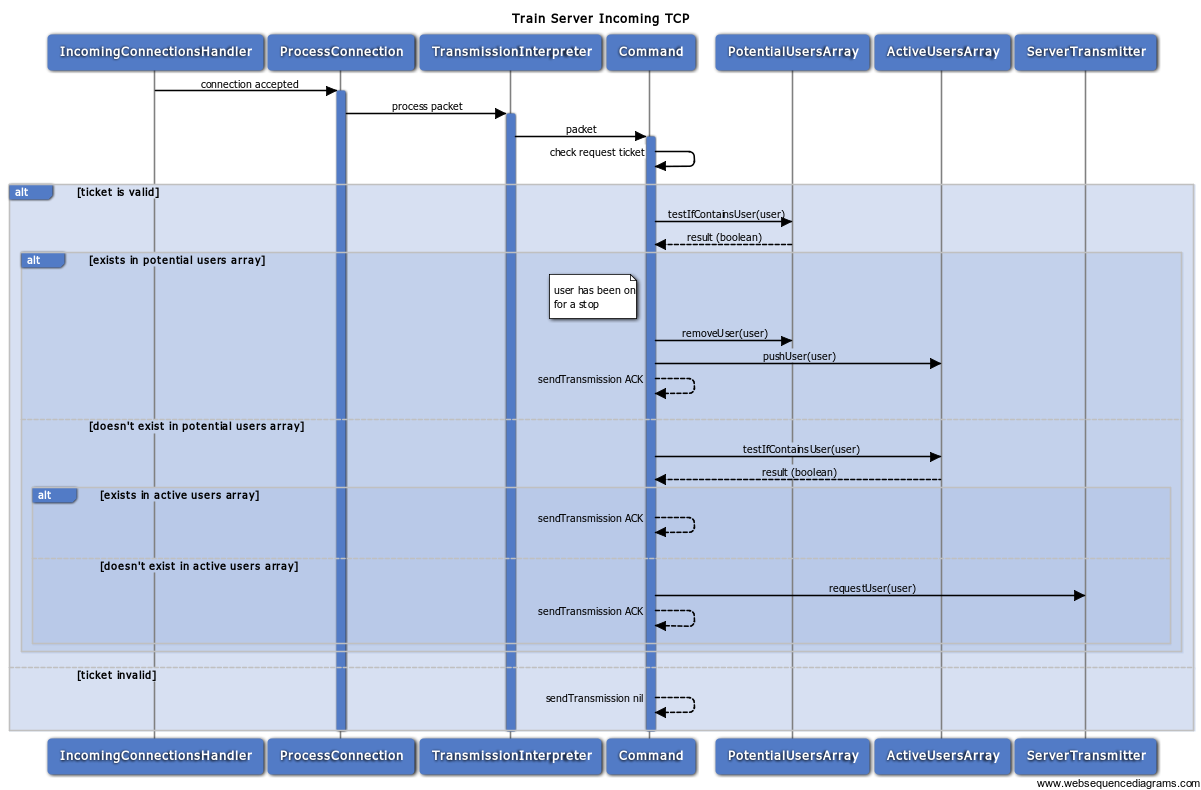
Figure 3 shows the sequence for receiving a client connection.

Figure 3

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.

### 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 train[[3]](#footnote-3) 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.

## Charging and Updating Concurrent Clients

The problem in detail

### 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 lacking threads.

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

Solutions:

Conclusion:

### Keeping Data Reliable

Premise:

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

Issues:

Updating a client after a charge operation can result in deleting the charge made.

Solutions:

Conclusion:

## Finding Shortest Path

The problem in detail

### Designing heuristic

Premise:

Issues:

Solutions:

Conclusion:

### Implementing A\*

Premise:

Issues:

Solutions:

Conclusion:

### Creating the Traffic Network Graph

Premise:

Issues:

Solutions:

Conclusion:

# Implementation

## Awaiting Client Connections

### Test

## A\* Search

### Test

## Faux GPS

### Test

## Database Handler

### Test

## Creating Directed Graphs

### Test

## Fibonacci Heap

### Test

# Conclusion

## Product Conclusion

## Development and Team Conclusion

### Development Authorship

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> [↑](#footnote-ref-1)
2. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [↑](#footnote-ref-2)
3. Siemens: 8-vogns S-tog litra SA, page 1: “Tekniske data” [↑](#footnote-ref-3)