Dynamic Timetables in Public Transport
Architecture Document

Alexandru Babeanu s3004872
Alexander Daffurn-Lewis s2906031
Ashton Spina s2906279

March 7, 2018

Contents

1 Architecture Description **1**
1.1 Servers . . . 1
1.2 Buses . . . 2
1.3 App . . . 2
1.4 Bus Stops . . . 2
1.5 Clients . . . 2

2 Technologies **3**

3 Database **3**
3.1 Design . . . 3
3.2 Diagram . . . 4

4 Interaction Diagrams **5**

5 Class Diagram **7**

1 Architecture Description

1.1 Servers

- There exists a table of server addresses which stores which route is which handled by which server and a copy of this table is stored in the local database of each server
- Basic data about each route is stored on each server. Collected data used for dynamic calculations is periodically shared between the servers in order to facilitate emergency route takeover.
- When a server needs to send the address of another server to a client, it pings/checks the status that server first to check for a failure before returning the address to the client. Assuming the servers are connected via socket, then the status of the server should be known and a check rendered redundant.
- If a server does not respond to a ping or a socket connection is noticed to no longer be connected, the pinging server will send out a message to the other server informing them of the death and a timestamp. If the server receives a death message for the same server from another server with a later timestamp it will ignore it. Once the server has received acknowledgements from all the other servers that they will not be taking on the load of the dead server it will take on the load. If that load overloads the server it will spawn a new server under the normal algorithm. If the server finds another dead server while asking for acknowledgement of the dead server it has

already found, it will proceed in a new, separate, death management response. All servers are connected via TCP/IP at a minimum so pinged servers must always respond if they are alive. If they are all connected via sockets then it should always be known if other servers are alive at any time.

- When a server reaches 100% utilization, it searches other servers for computational availability. If adding load to any other server would overutilize that server, then the current server spawns a new server to take the load. 1 server manages a minimum of 1 route. This ensures minimum waste as well as that a single route is not split over multiple servers in a way that might cause difficulties.
- If a server is under 40% load , it searches for servers that could take a route off it without reaching 100% utilization and offloads routes to other servers until it has none. If it can't offload all its routes it waits some time before checking again.
- When a server takes on a route, it should broadcast that it now handles that route information to other servers.
- Each server stores information in a database about the routes and updates the other servers with their local information periodically.

1.2 Buses

- Buses know what server handles their route and use POST requests to send Location, ID, Route Number, etc. to the relevant server for processing.
- Buses will get the up-to-date route information from the relevant server with a GET request.(eg. estimated time of arrival at each stop).
- When servers update their local timetables they send them through the sockets to the bus stops.
- Buses may send percentage route completed data to the servers in order to send more accurate and less data.

1.3 App

- App instances make GET requests to the servers for route data and sort the data locally.
- App instances are basic browsers which load a web page.
- The web page is a Web App and should be accessible from any browser.
- App instances will display data for a specific route or for a collection of routes for a bus stop.

1.4 Bus Stops

- Servers and Bus stops will be connected via sockets.
- If a socket is interrupted the client (the bus stop) makes a GET request to another server and the server will return the new server handling that route and a new socket connection is made.

1.5 Clients

- Table of servers mapped to handled routes stored on all clients that make a requests
- Any client (App, Bus, Bus stop timetable) should come with at least 2 (preferably more) server addresses hardcoded which should never be spun down in order to act as a failsafe. The first networking action of any client should be to GET the table of routes.

- When a client connects to a server it requests a route's datum (or multiple routes' data). The server will send back the information regarding the routes it handles along with the addresses of the servers handling the rest of the routes for the client to make requests to.

2 Technologies

- Node.js with Express.js for server backend
- Web based application using Angular.js for timetables, which will be loaded by an android mobile application (also works in browser)
- Mock objects made in JavaScript to simulate buses and bus stops
- MySQL for the the database that holds route data
- Continuous integration environment set up with TravisCI
- Mocha and Chai used for testing

3 Database

3.1 Design

We use a database to store basic information about the system needed for servers to operate, and to offer persistence and failure protection to data generated or received by the server. This database consists of the following 6 tables:

- Stations: stores the stations and their related information, including a key used to authenticate with the servers
- Routes: stores the routes present in the public transit network
- Buses: stores the buses and their related information, including a key used to authenticate with the servers; one bus can have 1 or 0 routes assigned to it
- StationsOnRoute: used to represent the many-to-many relation between stations and routes; stores the location of each station on each of its routes as a floating point number (dist(routeStart, station)/ routeLength); stores the current delay computed by the server for each route segment
- ExpectedTimes: used to store a basic approximation of the time needed to travel from a station to the next, given a route, a time of day, and a time of the year
- RecordedTimes: stores all the data received from the buses, used for future approximations of the expected travel times

3.2 Diagram

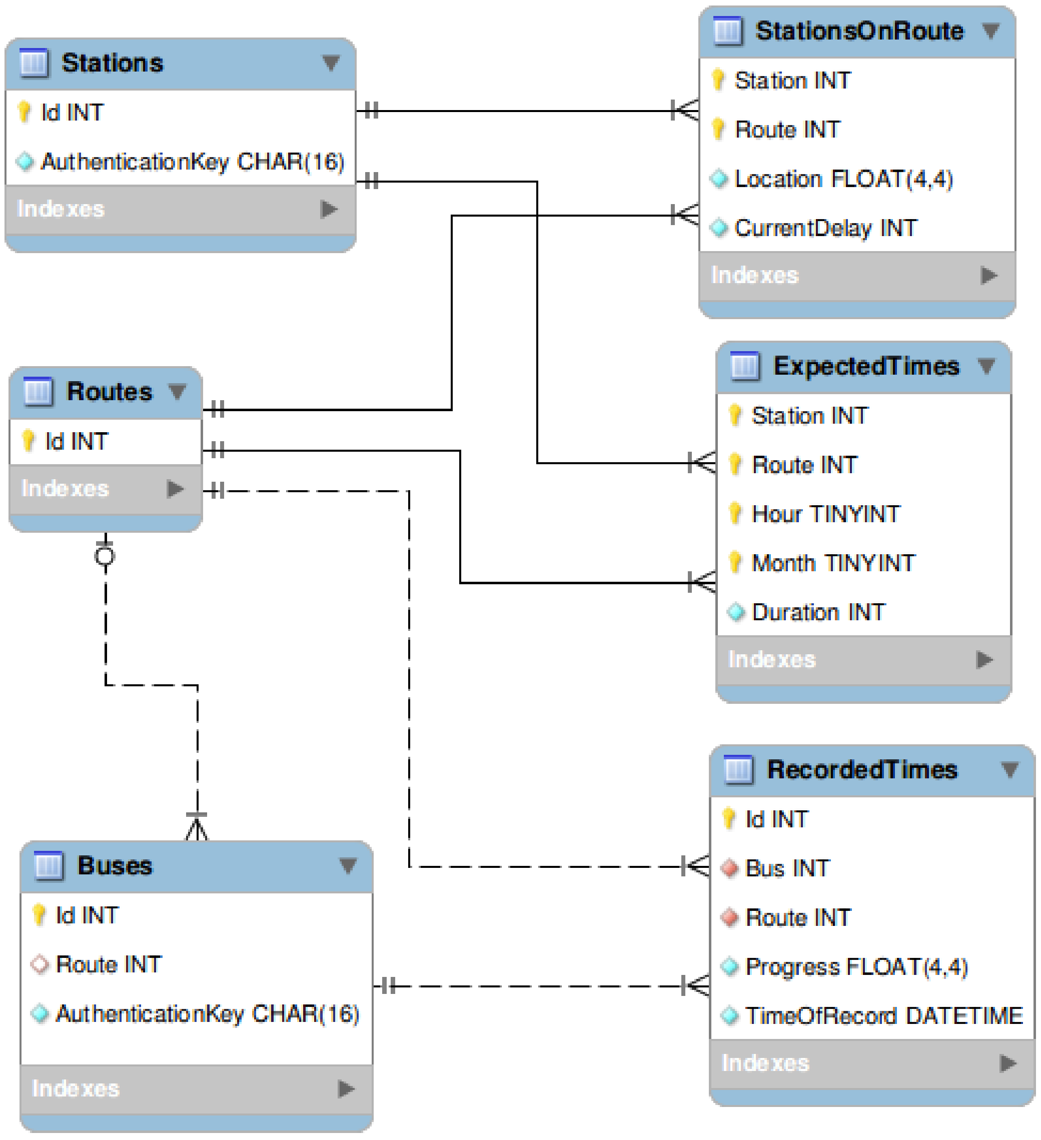
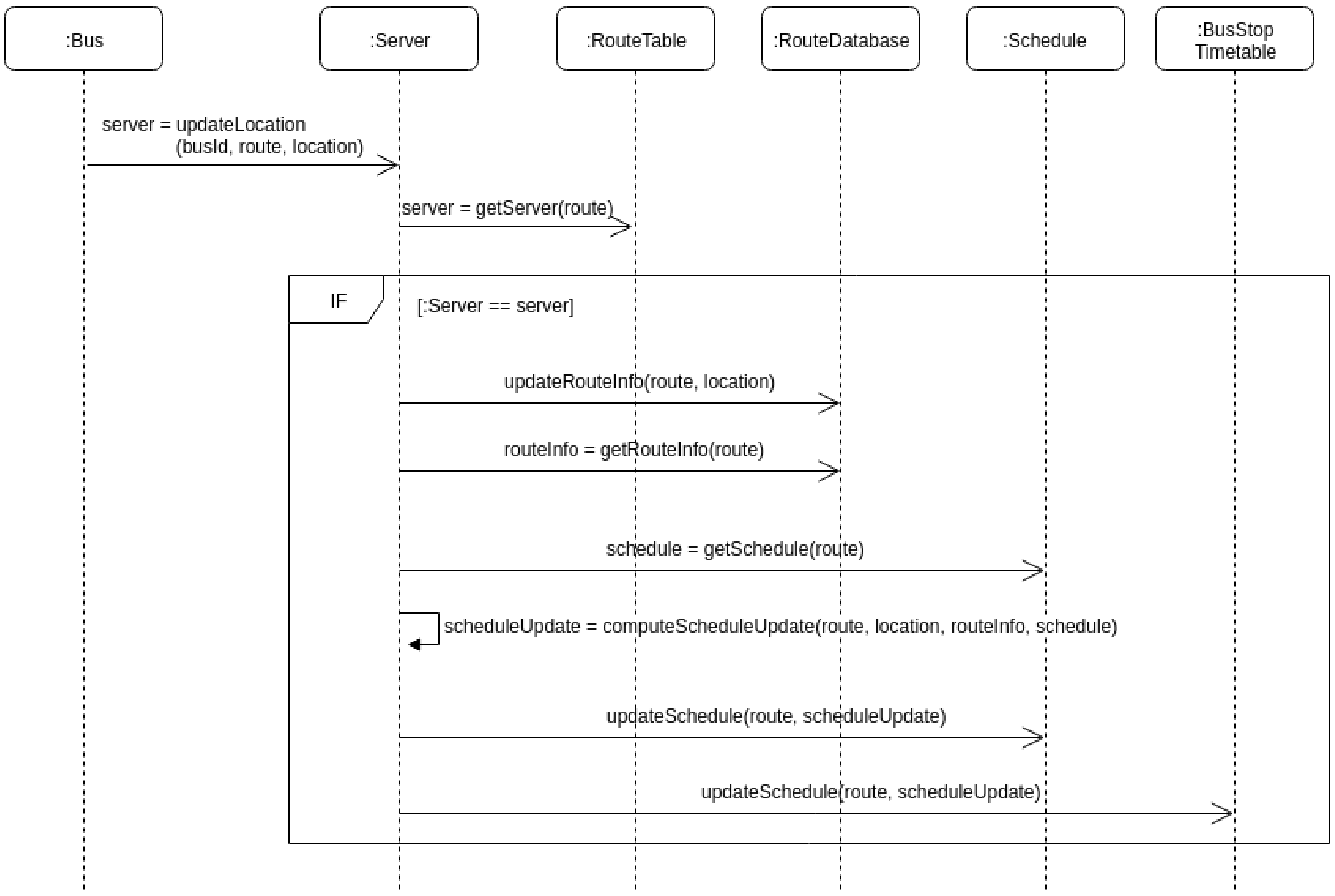


Figure 1: Database diagram

4 Interaction Diagrams



:Bus
:Server
:RouteTable
:RouteDatabase
:Schedule
:BusStop Timetable
server = updateLocation (busId, route, location)
server = getServer(route)
IF
[:Server == server]
updateRouteInfo(route, location)
routeInfo = getRouteInfo(route)
schedule = getSchedule(route)
scheduleUpdate = computeScheduleUpdate(route, location, routeInfo, schedule)
updateSchedule(route, scheduleUpdate)
updateSchedule(route, scheduleUpdate)

Figure 2: A bus sending its location data

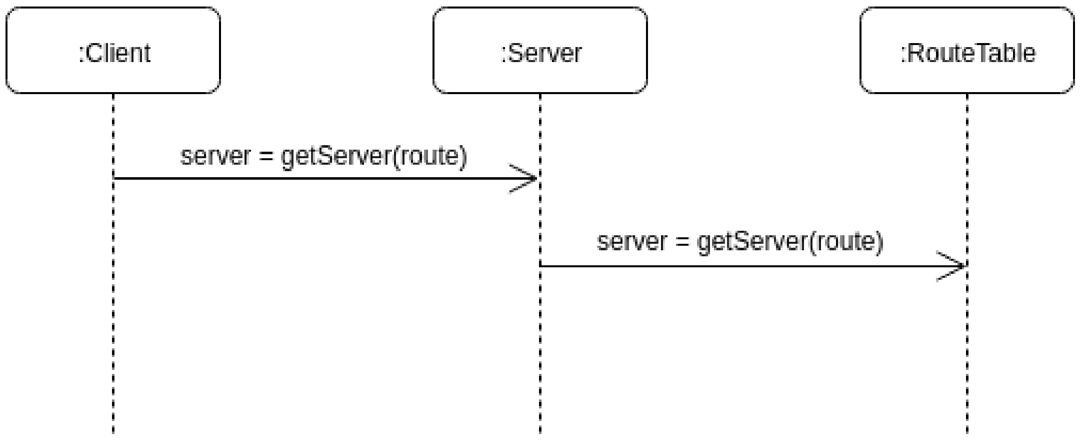


Figure 3: A client requesting a unloaded server to handle its route

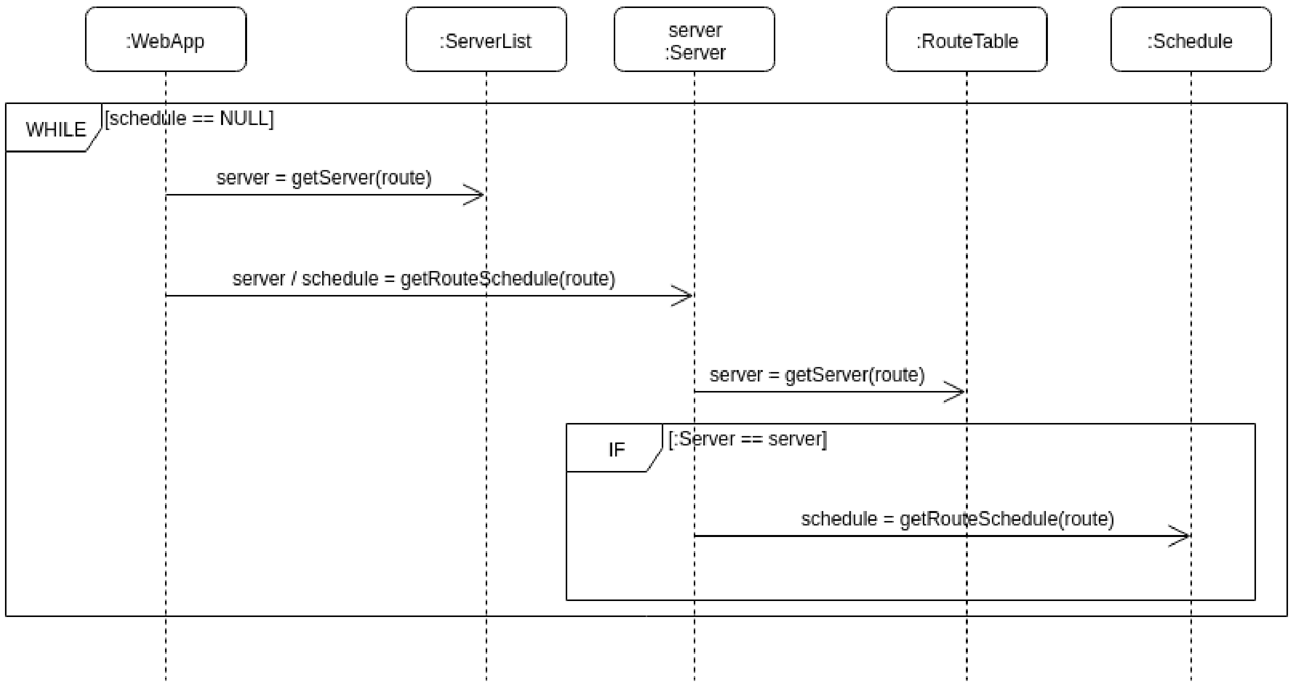
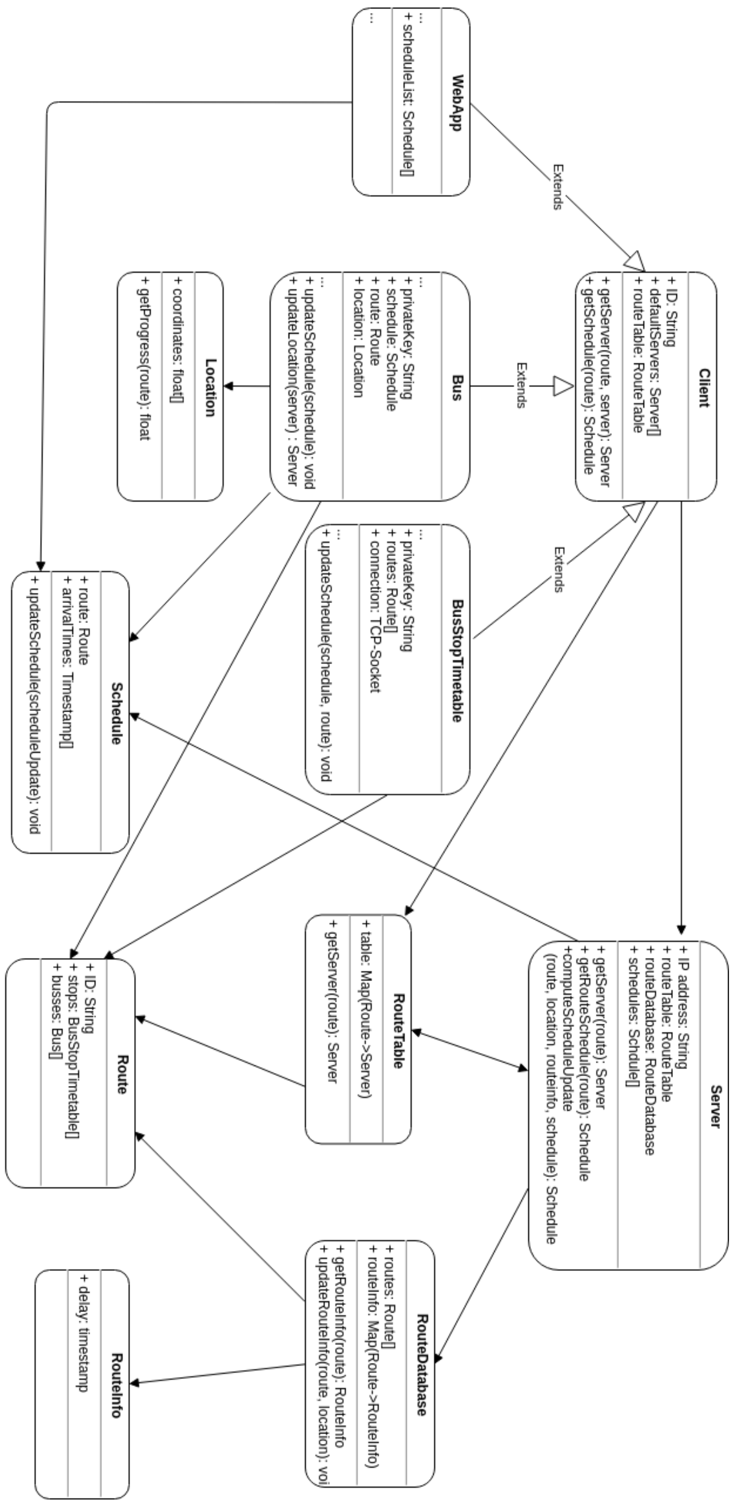

:WebApp
:ServerList
server
:Server
:RouteTable
:Schedule
WHILE
[schedule == NULL]
server = getServer(route)
server / schedule = getRouteSchedule(route)
server = getServer(route)
IF
[:Server == server]
schedule = getRouteSchedule(route)

Figure 4: The WebApp requesting the schedule of a specific route

5 Class Diagram



WebApp
...
+ scheduleList: Schedule[]
...
Extends
Client
+ ID: String
+ defaultServers: Server[]
+ routeTable: RouteTable
+ getServer(route, server): Server
+ getSchedule(route): Schedule
Extends
Bus
...
+ privateKey: String
+ schedule: Schedule
+ route: Route
+ location: Location
...
+ updateSchedule(schedule): void
+ updateLocation(server) : Server
Location
+ coordinates: float[]
+ getProgress(route): float
Extends
BusStopTimetable
...
+ privateKey: String
+ routes: Route[]
+ connection: TCP-Socket
...
+ updateSchedule(schedule, route): void
Schedule
+ route: Route
+ arrivalTimes: Timestamp[]
+ updateSchedule(scheduleUpdate): void
Server
+ IP address: String
+ routeTable: RouteTable
+ routeDatabase: RouteDatabase
+ schedules: Schdule[]
+ getServer(route): Server
+ getRouteSchedule(route): Schedule
+computeScheduleUpdate
(route, location, routeinfo, schedule): Schedule
RouteTable
+ table: Map(Route->Server)
+ getServer(route): Server
Route
+ ID: String
+ stops: BusStopTimetable[]
+ busses: Bus[]
RouteDatabase
+ routes: Route[]
+ routeInfo: Map(Route->RouteInfo)
+ getRouteInfo(route): RouteInfo
+ updateRouteInfo(route, location): voi
RouteInfo
+ delay: timestamp