Dynamic Timetables in Public Transport
Architecture Document

Alexandru Babeanu s3004872
Alexander Daffurn-Lewis s2906031
Ashton Spina s2906279

April 2, 2018

Contents

1 Architecture Description

1.1 Overview

The architecture in its most basic description is a system of P2P servers distributing load among themselves and sending and receiving data to and from clients. The servers are connected to each other via sockets, while the clients connect with GET and POST HTTP requests. We chose this design in order to maximize uptime and failsafes. The desired functionality is as follows:

1.2 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 on a set interval in order to facilitate emergency route takeover in case of server failure.
- Because all servers are connected via socket to every other socket, a server will know immediately if a socket has been closed for any reason and will initiate load redistribution. As such, if a request comes from a client to a server, that server will not return an address of a dead server.
- If a server died then the sockets through which it is connected report closed to the other servers. The receiving servers of this close message delete that socket from their socket map. The surviving server which handles the lowest numerical route in the mapRouteServer takes on the dead server's work.
- When a server reaches 100% utilization it informs that it is overloaded and provides a command to connect another server to the cluster and relieve the load. This utilization is currently calculated by counting routes handled by a server and seeing if the number is higher than a constant integer. This could be replaced by any load calculating function, for example one that checks core usage. This is a simplification of an easily expandable system of load redistribution. Because functions already exist to communicate with other servers in intervals and to redistribute load, it would be possible to have servers communicate and take on load to ensure maximum load of each server and close the now unused server. It would also be possible to automatically spawn new servers when maximum load is reached. These features were avoided for this project as they would be partially dependent on the hosting service used and how it spawned new servers and also would not be convenient for demonstration purposes as newly added servers would be impossible to add if they weren't strictly needed. As such, it has been a design decision not to actively redistribute load and add/remove servers from the cluster automatically for the purpose of this proof of concept project.
- When a server redistributes load in the mapRouteServer it broadcasts an updated map to all other servers. Servers that find themselves tasked with a certain route should automatically be able to take on their newly assigned routes because at least minimal information to complete this task had been transmitted previously.
- Each server stores information in a database and in volatile memory about the routes and updates the other servers with their local information periodically.
- When a server is spawned it simply loads information from the local Database. If a server is spawned with the "SPAWN" environment variable set to the ip:port of another server, it will not load from the Database, but rather update itself with the information the spawned from server is using to ensure freshness of data.

1.2.1 HTTP Protocol

The server has the ability to handle the following HTTP requests:

- GET '/' The server will respond with the homepage of the webapp
- GET '/appData' The server will respond with general information about all routes and stations. The response will have the following fields in the body field:
  - **mapRouteServer** which will contain for each route (in the form r0 for route 0) the address of the server handling that route

  - **stationRoutes** which is an array of objects with fields:
    * **id** which is the id of the station,
    * **name** which is the name of the station
    * **routes** which is an array of the routes that the station is part of (in the form 0 is route 0).

    There is an entry in this array for each station.

- POST '/busData' This is used for sending bus information to the server. The request should have the a body containing the following fields:
  - **id** The ID of the bus
  - **key** the authentication key which corresponds to the bus ID
  - **route** ID of the route that the bus is on
  - **progress** The percentage of the route that the bus has completed

  If a valid request is made the server's response will have the following fields in its body:
  - **route** ID of the route the bus is on
  - **serverOfRoute** The address of the server handling that route

  If the server receives a bad request it will reply with 400.

- POST '/routeData' This is used for requesting information about a specific route and station. The request should have a body containing the following fields:
  - **route** ID of the route
  - **station** ID of the station, the server will provide data for the whole route if null or undefined

  If a valid request is made the server's response will have the following fields in its body:
  - **serverOfRoute** The address of the server handling that route. If this is different to the address that the request was sent to the client will update this information in its mapRouteServer and send the request to the new address.
  - **schedule** which is an array of objects which represent each bus of the route. The array has the following fields:
    * **route** ID of route
    * **station** ID of station
    * **arrivalTime** Time in seconds until arrival at the station specified.

    If the address field is different to the address the request was sent to this will be undefined.

1.2.2 Socket Protocol

The sockets exist as a means for servers in the cluster to communicate with eachother. This web of sockets means that every server in the cluster should be connected to every other server in the cluster, allowing them to instantly communicate and know the status of eachother. A socket can do two actions once a connection is established: send a message or close. All sockets are json-sockets and only communicate in json objects.

Sending a message operates in the following way. A socket is selected from a map of sockets stored on the server. This map contains for each key of (host address : socket port) as a string, the actual socket object through which to communicate and the corresponding (host address : http port) for the server that socket corresponds to. If a message is received on a socket the protocol is always to first check if the server has that socket in its own socket map. If it does not it should add it to the map. Otherwise it continues to handle the message.

A message consists of a json object with multiple fields. The protocol is always that an object will contain first a command field with a string corresponding to the intended handling of the message and thus which fields will eventually be contained within. All messages contain information as follows:

- command
- host
- httpPort
- socketPort

In addition, for the following commands additional information will be listed as follows:

- requestData This request has no additional fields and only sends information about itself. The receiving server will send the initialize command back.
- requestWork This request also receives no extra fields. What it does is send the updateServerMap command to the server which send the requestWork command. It first splits the load of routes handled by this server between itself and the requesting server and broadcasts the new map to all servers in the cluster.
- initialize
  - appData
  - routeStations
  - socketAddressList

  This command tells the receiving server to update all its data, including the map it contains in its server data with is a map from routes to stations. It also updates any other appData and provides a list of socket addresses for the server to attempt socket connections to. This allows the new server
- hello This command simply informs another server of its existence in order to allow that server to add the communicating server to its socket map.
- updateServerMap
  - mapRouteServer

  This command tells the receiving server to update its map of servers with the data provided. This allows a server to update the map of servers to routes so that they know which servers handle which routes and respond accordingly to route information requests.
- updateRouteStations
  - routeStations

  This command tells the receiving server to update the map it contains in its server data with is a map from routes to stations. The data received is only information that the sending server might have changed and not the entire map.

1.2.3 Message Queuing

Since Node.js uses asynchronous function calling, message queuing is the default behaviour of our application. When messages are received, either HTTP requests or socket messages, they are handled one by one in the order in which they were received, without blocking the event loop with time consuming operations like database queries or message sending. Therefore, the default way in which the frameworks we used (Express.js for rest, json-socket for sockets) are handling communication incorporates message queuing.

1.3 Clients

- Each client has a table of mapped routes to servers that handle them
- Any client (App, Bus, Bus stop timetable) should ideally come with at least 2 (preferably more) server addresses hard-coded 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 that does not handle the route it is interested in, the server should reply with the address of the server that handles that route. The client will then update its table and make a request to the correct server.

1.4 Buses

- A bus informs the server responsible for its route of its current location on its route (a floating point value) every minute (or any arbitrary amount of time).
- For demonstration purposes, a mock object was made to simulate a bus in "MockObjects/mock_bus.js". Instructions:
  1. Open a terminal and run *node* in it
  2. Load the mock bus constructor
  3. Pass an id, the id of the route handled by the bus, the authentication key corresponding to the bus id, the address of the server to send HTTP requests to, the speed at which the bus travels through the route (a floating point number in [0.0, 1.0))
  4. All the above values need to match the ones in the server's database
  5. Call *start*() to start the bus
  6. For example:

```
>busFactory = require('./MockObjects/mock_bus');
>bus = busFactory(0, 0, '0000000000000000',
                   'http://192.168.0.113:36620/', 0.05);
>bus.start();
```

  Optionally you can also pass a value to *bus.start*() which will correspond to the rate at which the bus posts updates on its movements in milliseconds. If left blank the default value is 60000. Once you have finished with a bus you can call *bus.stop*() to stop it. If you want to just see the bus updating once without starting the routine you can call *bus.sendData*()

1.5 WebApp

The WebApp itself is the front end of the project. It consists of a basic html page that is loaded into the browser and decorated with Bootstrap. Most of the power of the WebApp is in the Angular javascript that is used to populate the table of buses. A user selects a route or a station or a combination and the Javascript makes POST requests to the servers in order to populate the lists. If it requests a route that is not handled by the server the request is made to, it is returned a map of routes to servers so it can make requests to the appropriate servers. It will make a series of requests if many routes need to be requested. It will update itself periodically to keep the data current. The functionality can be defined as follows:

- App instances load the basic page with a GET request.
- App instances make POST requests to the servers for route data and sort the data locally..
- App instances will display data for a specific route or for a collection of routes for a bus stop or for a specific route and bus stop.

- App instances will update themselves periodically to keep information current.

The WebView of the Android App acts as a browser and does and has the same functionality, but its unfortunately not practical to setup because of the necessity of a hardcoded domain.

1.6 Bus Stops

- Bus stops are assigned the routes that use the stop. They then use HTTP requests to the server to get the schedule for each route at the stop. They will constantly update themselves every minute (or you can specify a update time).
- For demonstration purposes we created a mock object to simulate a bus stop in "MockObjects/mock_stop.js". Instructions:
  1. Open a terminal and run *node* in it
  2. Load the mock stop constructor
  3. Pass the ID of the stop, a list of routes handled by the stop and the address of a server to send HTTP requests to
  4. The ID of the stop and each route in the list must exist in the database
  5. Call *initialize*() to update the mapRouteServer
  6. Call *start*() to begin the stop
  7. Call *display* to print the schedule
  8. For Example:

```
>stopFactory = require('./MockObjects/mock_stop.js');
>stop = stopFactory(0, [0,1,2,3,4,5,6,7,8,9],
                    "http://192.168.0.108:36480/");
>stop.initialize();
>stop.start();
>stop.display();
Route: 9 Arrival Time 16:55
Route: 1 Arrival Time 16:55
Route: 1 Arrival Time 16:55
Route: 2 Arrival Time 16:55
Route: 3 Arrival Time 16:55
Route: 8 Arrival Time 16:55
Route: 7 Arrival Time 16:55
Route: 4 Arrival Time 16:55
Route: 0 Arrival Time 17:00
Route: 0 Arrival Time 17:00
Route: 6 Arrival Time 17:03
Route: 5 Arrival Time 17:03
```

Again you can pass a value to *stop.start*() to specify the rate it will request updates from the server (in milliseconds). The stop can be stopped with *stop.stop*() and if you want to just update the schedule without starting the routine you can call *stop.update*()

2 Technologies

- Node.js with Express.js for server backend
- Web based application using Angular.js for timetables

- Android / Java (Note : a WebApp was made for this project, but because of the dynamic nature of the host ip because the project is not being hosted anywhere, the necessarily hardcoded ip of the WebApp is infeasible and really unusable until the project is hosted somewhere)
- 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
- 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

4 Tests

We chose to use test driven development to ensure our project was working correctly throughout development. Our test suite currently tests all possible HTTP requests that can be made to the server. As this functionality is the core of our project, it is essential that they are always working correctly. We used Mocha with Chai to perform these tests, and implemented them within TravisCI so that whenever something was pushed to the github repository these tests would be run. The setup script that Travis runs also creates and populates the database as the tests depend on the correct data being in the database.

The tests themselves can be found in './test' with a file for each of the routes the servers can handle. For testing the '/appData' route they make a GET request to that address and verify that the reply is in the correct format that the contents of the reply are correct. For the other two routes '/busData' and 'routeData' they make a data object containing the correct fields and values and POST it to the address. They then verify that the reply is correct. We also POSTed some incorrect data to the servers and verified that the server did not accept it.

The final element of the tests was implementing code coverage checks with the Istanbul library. We also set it up so that this result will be sent to coveralls.io so the results can be access online along with Travis. At the moment the coverage sits at around 53% which is a little lower than desired, however this is mostly due to our tests not covering the creation of multiple servers with sockets.

Ideally if we were to continue development on this project will would add more tests to cover this functionality.

5 Database Diagram

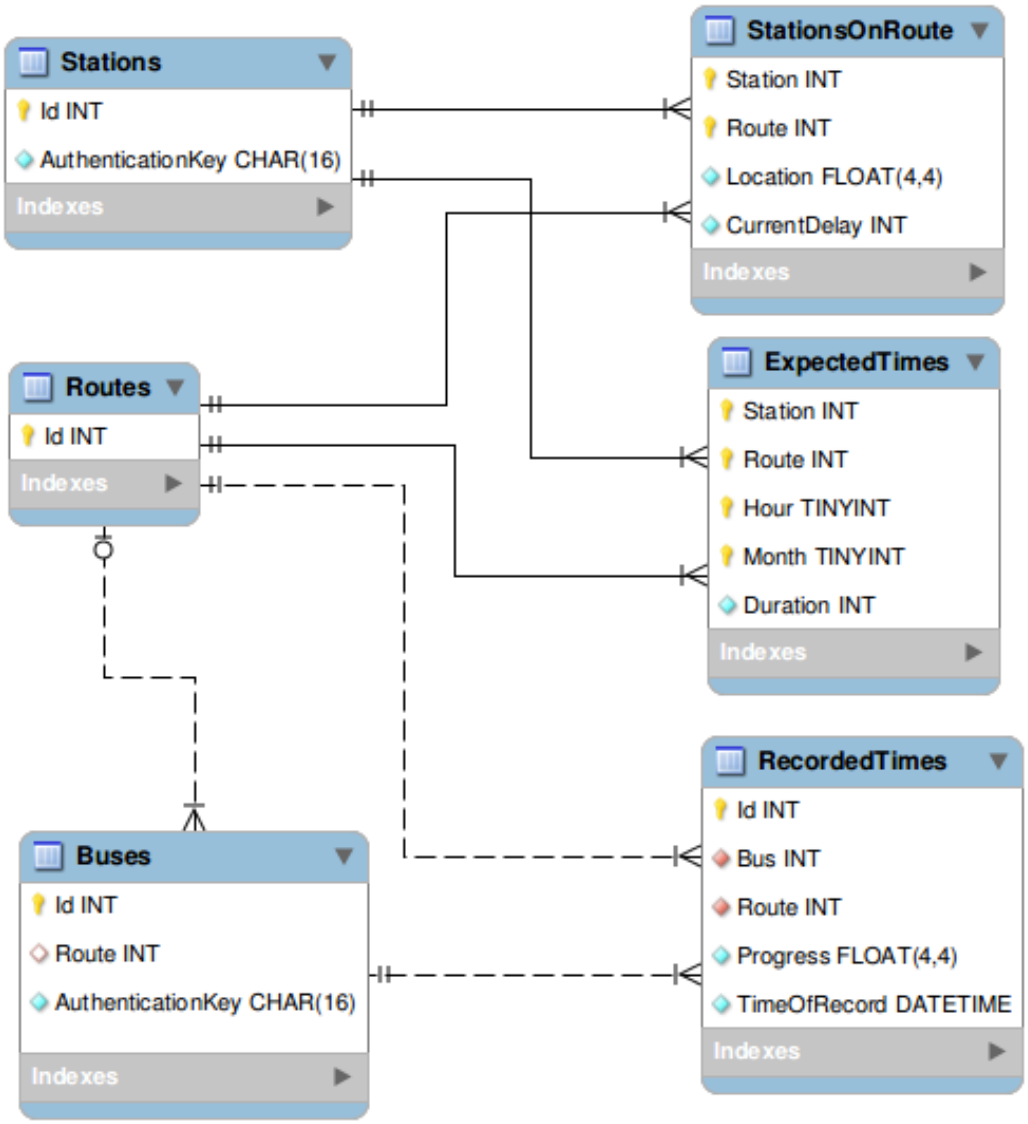

Figure 1: Database diagram

6 Interaction Diagrams

6.1 Client Diagrams

6.1.1 Bus Behaviour

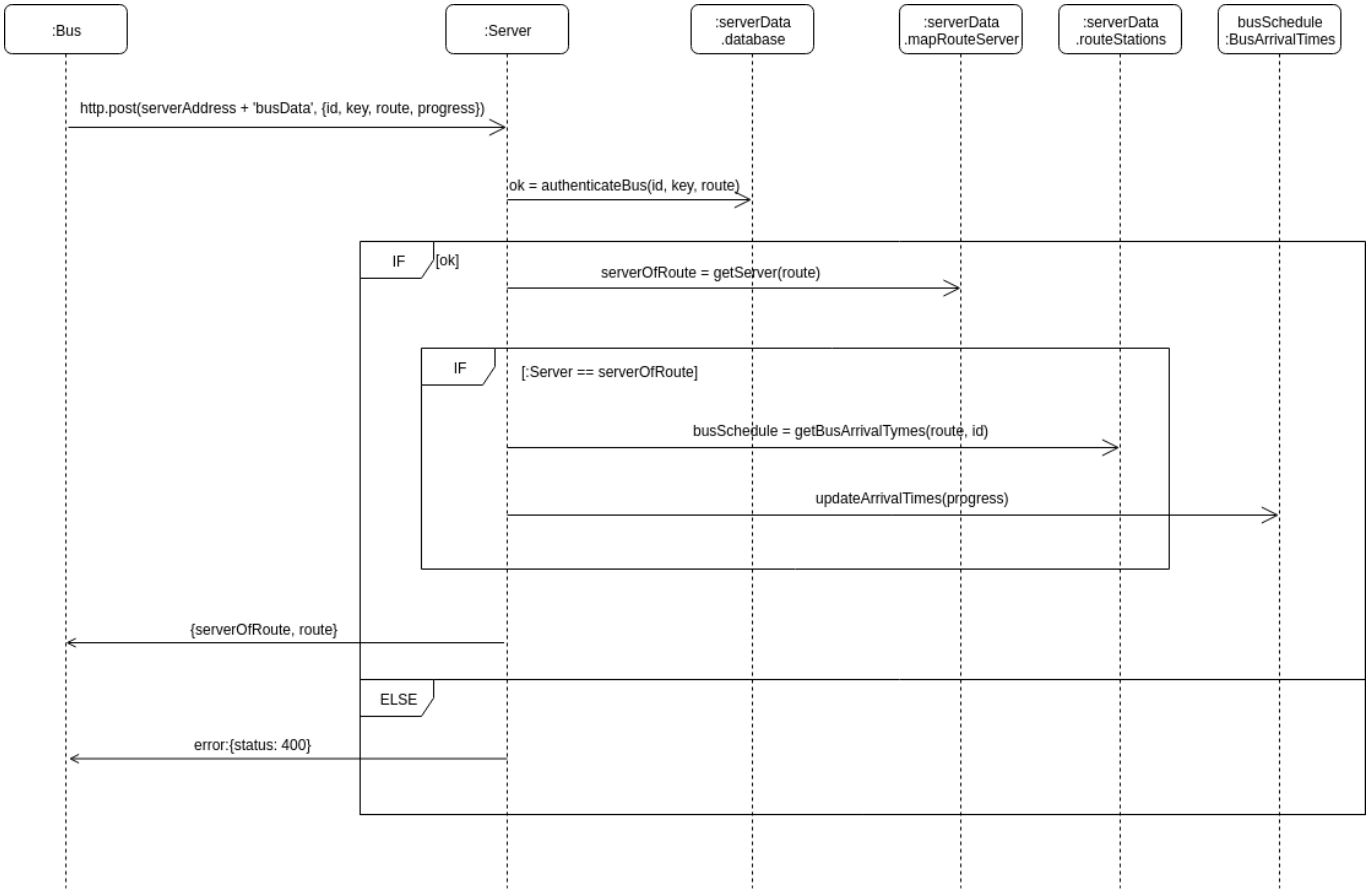

Figure 2: A bus sending its location data

6.1.2 WebApp Behaviour

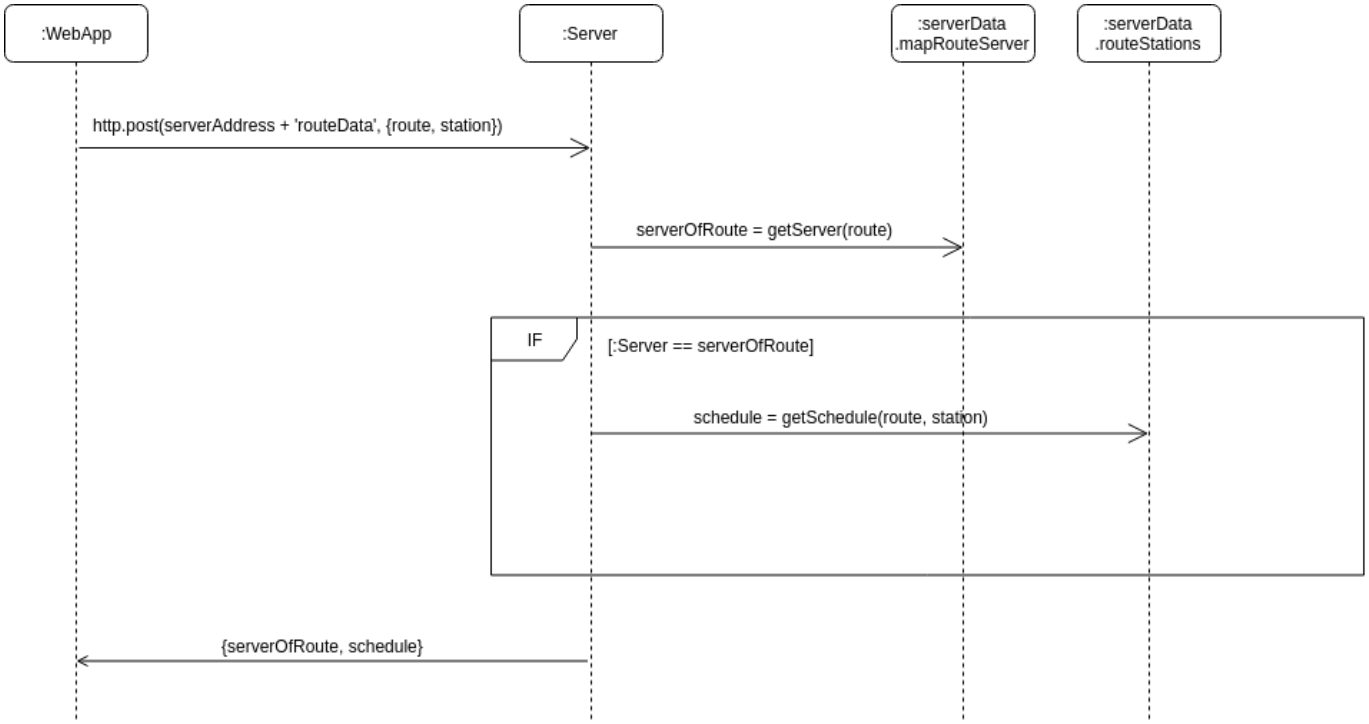

Figure 3: The WebApp requesting the schedule of a specific route

6.2 Server Diagrams

6.2.1 Server Close

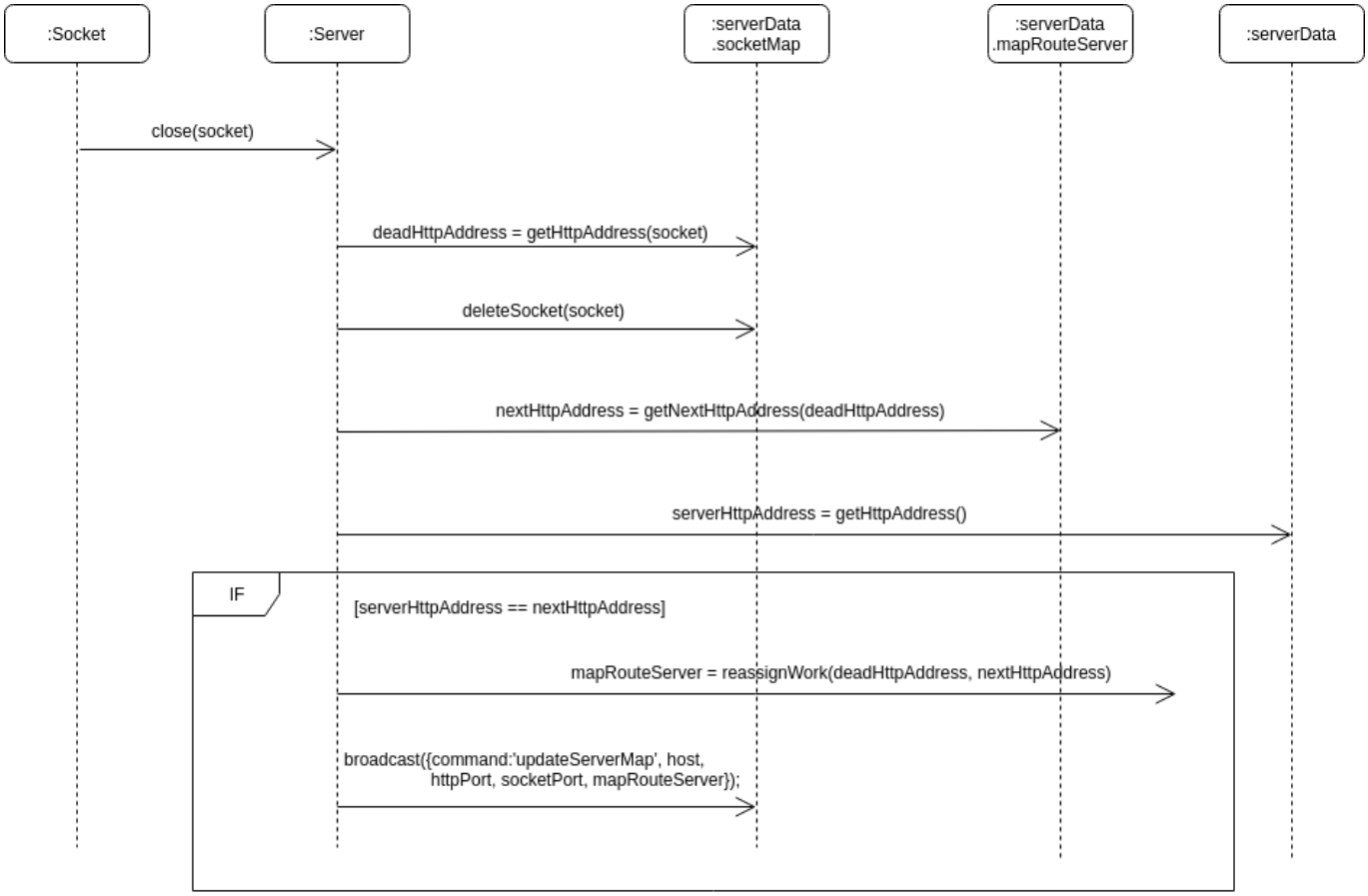

Figure 4: A server stops running

6.2.2 Server Spawning

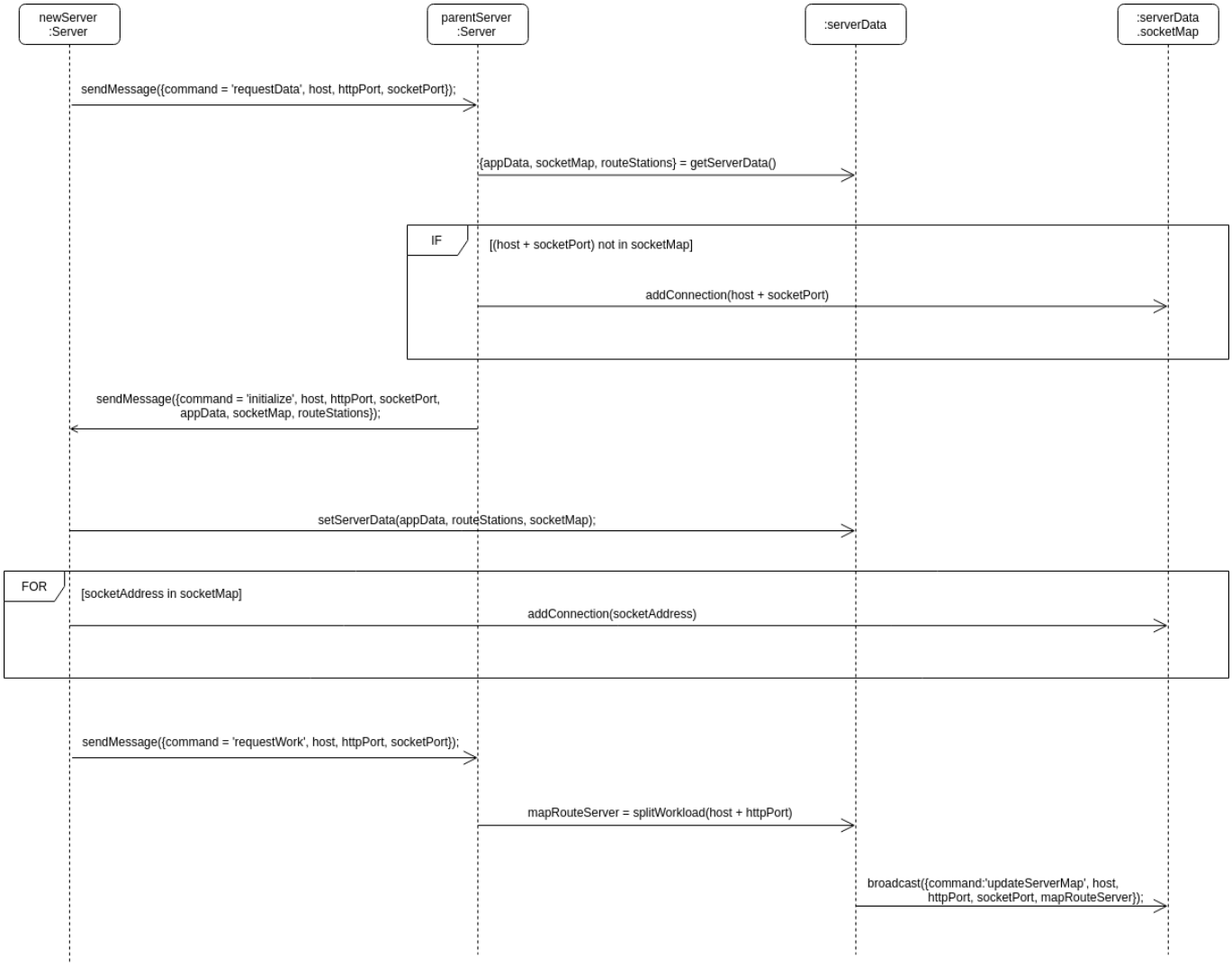

Figure 5: A new server is spawned

7 Class Diagram