DATA3900 - Bachelorproject
Final report

Gruppe 23:
Hansen, Andreas Torres (s338851)
Nguyen, Uy Quoc (s341864)
Ottersland, Anders Hagen (s341883)

Total pages: 5

Last updated:
May 16, 2022

Abstract

Short summary of the project including the result that the group reached.

Preface

This is the report of our bachelor thesis at Oslo Metropolitan University, Faculty of Technology, Art and Design. Our project was done for Accenture, and lasted from January 2022 to May 2022. We have tried to develop a solution for traffic management with self-driving cars and server communication. A physical demonstration of our proposed solution is done with the use of a Raspberry Pi computer working as a car. In this project we have documented the functionality of our system, and our process of making it.

The report is split into 7 chapters. Introduction, research areas, process documentation, implementation, results, discussion and conclusion. At the end we have added a Moscow analysis, sprint overview and our project journal as appendices. Technical terms are in appendix for those with less technical knowledge. (Kan kanskje droppe dette)

We would like to thank everyone that has contributed to our project. We would especially like to thank:

- Ivar Fauske Aasen, Solfrid Johansen and Benjamin Vallestad, representatives from Accenture.
- Dr. Jianhua Zhang, internal supervisor at OsloMet.

Contents

Implementation

1.1 Phase 2: REST API

The aim of the project was to connect Raspberry Pi devices to a system. Thus, the group decided that the best way to establish this connection was to implement a RESTful API on the server. In addition, the question on how the data should be stored was also discussed during this phase.

Representational state transfer (REST) *application programming interface* (API) provides a way for client and server to establish communication through *hypertext transfer protocol* (HTTP). Using a REST API clients can send requests to a server to perform standard CRUD (create, read, update and delete) operations on a database (IBM Cloud Education, 2021).

Due to the time sensitivity of the application we are trying to build, it was therefore necessary to choose an appropriate type of database for our system. In this case, we chose to incorporate the time series database. InfluxDB is a time series database created by InfluxData. It provides a SQL-like syntax for querying resources that is quick and scalable, and most importantly free. Moreover, InfluxDB client library, using the influxDB v2 API, provides both ease of install and use for a multitude of langauages (Influxdata, n.d.).

During this phase the group implemented a standard REST API server with C# with the idea that the Raspberry Pi vehicles should exchange information on its velocity, acceleration and position to the server. Client should first POST itself to the server. The server will then add the vehicle to the influxDB to keep track of the vehicle's information. Then, the vehicle should be able to perform a GET request to the server to retrieve its information.

The client at this phase would be able to send a PATCH request to the server in order to update its information. In addition, the server will add a new entry into the database whenever it receives this request. The idea was that the client and server would be able to continuously communicate to each other such that the server could determine the behaviour of all connected clients.

However, a RESTful API server were not able to perform all its required task that the group wanted the server to do. Firstly, the server were only able to communicate with one client at the time, i.e. the client that sends a request. What the group wanted at this stage was that based on a request the server

should also be able to send its own request to all the other clients. In order to acheive this, the server had to send an unprompted response to other clients that is not requesting a resource from the server, which was not possible with our current architecture.

A new solution had to be in place in order to achieve our goal. Other solutions were proposed during this stage.

Long polling

Polling is the idea that the server pushes resources to the client. There are mainly two types of polling; short- and long polling.

In short polling, a client requests a resource from the server and the server responds with nothing if the resrouce is not available. The client will then send a new request in a short amount of time and the cycles repeats until the client receives the resource it has requested.

Long polling is similar to short polling, however, the server does not send anything back before the resource is available. That is, the client sends a request to the server and the server is holding this request until it has a response available to the client. In our case, we wanted every client to perform a GET request to the server on a seperate thread and instruct the server to hold onto this request until it had further instructions to the requesting client.

However, implementing a method on the server to block the response introduced more complication to the project. Also, with the asynchronous nature of the controllers implemented on the server it would also mean that the server will consume a lot of the processor which also means that the performance of the server will be heavily deteriorated.

Implement REST API on the client

Another solution was to implement the client itself as a REST API server on its own. However, in order to achieve this, each client needed to also send the server its host and port information to the server. Also, the server has to be implemented as a client in order to connect to the vehicles.

Webhooks

Webhooks, according to Atlassian, 2019, is a user-defined callback over HTTP. In our case, implementing webhooks to post notifications on clients based on events sent to the server. This was a good contender to solve our issue. However, implementing webhooks includes extensive research into a system the group had never heard of, in addition to scarce information on how to create such a system. The group decided that the time constraint of this project did not justify the time it would take to implement such a system.

1.2 Phase 3: SignalR

After exhaustive discussions on how to solve the two-way communication discussed in Phase 2: REST API, the group agreed that websockets would be a

good solution to our problem. Websockets is a protocol that provides a bidirectional communication between clients and server by establishing a single TCP connection in both direction (Fette & Melnikov, 2011). Hence, using websockets both client and server can transfer data whenever they see fit. However, Microsoft, 2022b discourage developers from implementing raw websockets for most applications, and recommends using SignalR instead.

ASP.NET SignalR is a library that at the top layer provides real-time communication using websockets while also provides other transport methods such as long polling as fallback (Microsoft, 2022a). Furthermore, SignalR API supports *remote procedure calls* (RPC) using hubs, meaning we can invoke subroutines on the client from the server and vice versa (Microsoft, 2022a).

During this phase we disregarded our old REST API server and InfluxDB completely. First, setting up an echo server using SignalR, while simultaneously implementing the client code. The client code is required to be implemented independent from the Raspberry Pi code because our goal was to create a communication module that could be reused through inheritence for other devices, e.g. traffic lights, should it be required to set up a new hub with other devices.

After successfully implementing all the necessary methods on the client. The vehicle class that represented the Raspberry Pi device was created. Vehicle class inherits the client class which gives it the ability to connect, listen and send data to the server. Furthermore, the client can also subscribe to events that the server can trigger using RPC.

After witnessing a successful connection between the Raspberry Pi vehicle and our SignalR server we started to implement necessary functionality on the server. A simple scenario was first taken into consideration when we first developed new functionalities. The client will inform the server whenever its velocity has been changed. In this case, the server should informs every vehicles behind that vehicle on the same road to adjust their own velocity accordingly. As a result of this functionality, we are required to continuously keep track of the vehicle´s position. Thus, raising a new issue on how the vehicle information should be stored.

InfluxDB could in theory be used to store the vehicle's position however, since the position is constantly changing it would require the server to continuously read and write on Influx. Hence, the group concluded that in theory this will impact latency on server responsiveness. Thus, unanimously we determined that a live-in-memory database using lists would be better. Using simple mathematics the server could recalculate the vehicle's position based on its previous velocity and a stopwatch whenever it retrieves the information of a vehicle instead.

Expanding further on this concept, new functionalities on the SignalR server was developed to handle vehicles approaching an intersection. With a more complex topology it is also required to expand the database to account for the new road network. Hence, road models and intersections models were created to represent these concepts. Furthermore, the vehicle model on the serverside now also composed of a route planner to represent what it means to be approaching an intersection.

With these improvements, SignalR server proves successful in establishing communications with clients. In addition, with the implemented functionalities the server is able to command the clients to adjust their velocity to avoid collision between vehicles approaching an intersection simultaneously. By reducing velocity of some vehicles it also became apparent that traffic flow is improved, in contrast to stopping a vehicle.

Bibliography

Atlassian. (2019, August 13). *Webhooks*. Retrieved May 16, 2022, from https://developer.atlassian.com/server/jira/platform/webhooks/

Fette, I., & Melnikov, A. (2011). The websocket protocol. https://doi.org/10.17487/RFC6455

IBM Cloud Education. (2021, April 6). *Rest apis*. Retrieved May 16, 2022, from https://www.ibm.com/cloud/learn/rest-apis

Influxdata. (n.d.). *Influxdb is the time series platform for developers*. Retrieved May 16, 2022, from https://www.influxdata.com/products/influxdb-overview/

Microsoft. (2022a, March 26). *Overview of asp.net core signalr*. Retrieved May 16, 2022, from https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-6.0

Microsoft. (2022b, March 26). *Websockets support in asp.net core*. Retrieved May 16, 2022, from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-6.0