

**KULIYYAH OF INFORMATION AND COMMUNICATION TECHNOLOGY**

**DEPARTMENT OF COMPUTER SCIENCE**

**FYP REPORT**

**REAL-TIME WEB-BASED AGV-FLEET MANAGEMENT SYSTEM**

**MUHAMMAD AMIRUL ASHRAF BIN MOHAMAD FAUZI**

**1229237**

**SUPERVISED BY**

**DR RIZAL MOHD NOR**

**MEI 2016**

**SEMESTER II 2015/2016**

**FINAL YEAR PROJECT REPORT**

**REAL-TIME WEB-BASED AGV-FLEET MANAGEMENT SYSTEM**

By

Muhammad Amirul Ashraf bin Mohamad Fauzi

1229237

Supervised by

Dr Rizal Mohd Nor

In partial fulfilment of the requirement for the

Bachelor of Computer Science

Department of Computer Science

Kulliyyah of Information and Communication Technology

International Islamic University Malaysia

Mei 2016

Semester II 2015/2016

**Real-Time Web-Based Agv-Fleet Management System**

by

Muhammad Amirul Ashraf bin Mohamad Fauzi

1229237

A project paper submitted to the

Department of Computer Science

Kuliyyah of Information and Communication Technology

In partial fulfillment of the requirement for the

Bachelor of Computer Science

Approved by the Examining Committee:

Supervised by

Dr Rizal Mohd Nor

I

# AKNOWLEDGEMENT

All praise to Allah whose blessing enable me to accomplish this project. Without His mercy this project would not have been possible.

I wish to express my appreciation to my supervisor Dr Rizal Mohd Nor for his guidance, suggestion, pep talk and for giving me this project.

A special thanks to the IIUM Robotic Team for providing me with the necessary equipment for this project. Specifically, Dr Zul and Arul Fikri for their assistance along with other IIUM Robotic Team members who helps make this FYP a success.

Thank you for my parents for their understanding of this busy times. Also, thank you to all my lecturers and their cooperation. Special thanks to Dr Hamwira and Sr Dasila for helping me with my very late registration.

My sincere gratitude for all whom directly or indirectly help me complete this project. May Allah bless you all.

# [[1]](#footnote-1) ABSTRACT

This project present a Real-time Web-based AGV(Autonomous Guided Vehicle)-Fleet Management System made with Meteor. The system communicates with the AGV fleet using a TCP connection through Wi-Fi. The AGV uses an Arduino board which has been programmed to send sensor readings such as temperature, battery voltage and other readings. An RFID sensor is used for positioning. The system uses MongoDB as the database, which along with its integration with Meteor, provide real-time updates to the client. This prototype is a proof of concept as a part of a project to develop Proton’s AGV Fleet central management system. Proton is Malaysia’s first national automobile company.

TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| CHAPTER | TITLE | PAGE |

[AKNOWLEDGEMENT 3](#_Toc451868907)

[Table of Contents **Error! Bookmark not defined.**](#_Toc451868909)

[List of Figures 7](#_Toc451868910)

[List of Tables **Error! Bookmark not defined.**](#_Toc451868911)

[1. INTRODUCTION 1](#_Toc451868912)

[1.1 Problem statement 2](#_Toc451868913)

[1.2 Objective 2](#_Toc451868914)

[1.3 Methodology 2](#_Toc451868915)

[2. RELATED WORKS 3](#_Toc451868916)

[3. ARCHITECTURE 4](#_Toc451868917)

[3.1 Prototype AGV 5](#_Toc451868918)

[3.2 The system 7](#_Toc451868919)

[3.3 Machine Interface Server and Web Server 8](#_Toc451868920)

[3.4 Client 9](#_Toc451868921)

[3.4.1 Dashboard Page 9](#_Toc451868922)

[3.4.2 Map Page 9](#_Toc451868923)

[3.4.3 Machine Page 10](#_Toc451868924)

[3.4.4 Message Log Page 11](#_Toc451868925)

[3.4.5 Configuration Page 12](#_Toc451868926)

[3.5 Custom TCP Protocol 12](#_Toc451868927)

[3.6 Database Structure 13](#_Toc451868928)

[3.6.1 Machines 14](#_Toc451868929)

[3.6.2 Readings 15](#_Toc451868930)

[3.6.3 LocationLogs 16](#_Toc451868931)

[3.6.4 CommandQueue 16](#_Toc451868932)

[3.6.5 MessageLogs 17](#_Toc451868933)

[3.6.6 clientMachinePing 17](#_Toc451868934)

[3.6.7 clientResponseTimeLogs and clientResponseTimes 17](#_Toc451868935)

[3.6.8 GlobalStates 17](#_Toc451868936)

[3.7 Serial and Wi-Fi Configuration 18](#_Toc451868937)

[3.8 Line Following Algorithm and Manual Mode 19](#_Toc451868938)

[3.8.1 Line Following Algorithm 19](#_Toc451868939)

[3.8.2 Manual Mode 20](#_Toc451868940)

[3.9 Map and Location 20](#_Toc451868941)

[4. PERFORMANCE 23](#_Toc451868942)

[4.1 Machine Interface Server and Web Server 23](#_Toc451868943)

[4.1.1 Machine Interface Server 24](#_Toc451868944)

[4.1.2 Web Server 25](#_Toc451868945)

[4.2 Client 26](#_Toc451868946)

[4.3 Prototype AGV 27](#_Toc451868947)

[5. CONCLUSION 29](#_Toc451868948)

[4.1 Data synchronizations 29](#_Toc451868949)

[4.2 Client Processing Capability 29](#_Toc451868950)

[6. References 31](#_Toc451868951)

# LIST OF FIGURES

|  |  |  |
| --- | --- | --- |
| FIGURE NO. | TITLE | PAGE |

[Figure 1. Possible configuration 4](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884069)

[Figure 2. The prototype AGV 5](#_Toc451884070)

[Figure 3. Schematics of the prototype AGV 6](#_Toc451884071)

[Figure 4. RFID reader next to an array of infrared sensor. 7](#_Toc451884072)

[Figure 5. Prototype Circuit 7](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884073)

[Figure 7. Dashboard Page 10](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884074)

[Figure 7. Map Page 10](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884075)

[Figure 8. Machine Page 11](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884076)

[Figure 9. Message Log Page 12](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884077)

[Figure 10. Configuration Page 12](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884078)

[Figure 11. Custom TCP Protocol Handshake 13](https://d.docs.live.net/675b88c0b2dd3804/Documents/FYP/Final%20Report.docx#_Toc451884079)

[Figure 12. The database structure of the system 14](#_Toc451884080)

# LIST OF TABLES

|  |  |  |
| --- | --- | --- |
| TABLE NO. | TITLE | PAGE |

[Table 1. Machines collection fields 15](#_Toc451884081)

[Table 2. List of readings 15](#_Toc451884082)

[Table 3. LocationLogs collection fields. 16](#_Toc451884083)

[Table 4. CommandQueue item fields 17](#_Toc451884084)

[Table 5. MessageLogs collection fields 17](#_Toc451884085)

[Table 6. List of stored configurations 18](#_Toc451884086)

[Table 7. Serial configuration commands 19](#_Toc451884087)

[Table 8. Wi-Fi configuration commands 19](#_Toc451884088)

[Table 9. Manual mode Wi-Fi commands 20](#_Toc451884089)

[Table 10. Test environment 23](#_Toc451884090)

[Table 11. Metrics 23](#_Toc451884091)

[Table 12 Idle stats 24](#_Toc451884092)

[Table 13. Machine interface server CPU performance 24](#_Toc451884093)

[Table 14. Machine interface server performance with 3 server instance 24](#_Toc451884094)

[Table 15. Idle CPU usage per connected AGV 24](#_Toc451884095)

[Table 16. MongoDB CPU usage 25](#_Toc451884096)

[Table 17. Web-Server CPU load on Dashboard page 25](#_Toc451884097)

[Table 18. Web-Server CPU load on Machine Page 25](#_Toc451884098)

[Table 19. CPU usage on Machine page with two server instance 26](#_Toc451884099)

[Table 20. client response time over connection type 26](#_Toc451884100)

[Table 21. Client-Machine response time 26](#_Toc451884101)

[Table 22. Browser CPU usage 27](#_Toc451884102)

[Table 23. Prototype AGV stats 28](#_Toc451884103)

# INTRODUCTION

AGV stands for Autonomous Guided Vehicle (AGV). These AGVs are autonomous, requiring little to no user intervention in order to do their task. They are cost effective, providing transport capability within a factory floor autonomously. Generally, the AGVs are guided within the factory floor with color-based line or magnetic line, although there are researches on free-moving AGVs [1].

Proton (Malaysia’s first national automobile company) have a fleet of AGV tasked with transporting car components. Currently, these AGVs does not have a central management system and rely on manual inspection. The battery is charged twice a day without determining if it need charging. Sometimes the AGV wander out of its intended track and stop. When this happen, the operator can manually control the AGV through a control panel on the AGV. However, the factory is quite large and moving from one place to the AGV can be a chore. Because of this, the practical number of AGV is limited by the maintenance cost. Having a centralized management system would allow the operator to easily inspect the performance of the AGVs and isolate AGVs with problems.

This project presents a prototype for a Web-Based central monitoring system for those AGVs. This system is intended to be a Web-Based application. Compared to standalone application, Web-based applications have several advantage:

* Can be accessed from any internet capable devices.
* Do not require installation on client devices.
* Instant update on client as it is not actually installed.
* Flexible HTML based UI.
* Because of the natural Client-Server architecture, multiple user can access it at the same time. Multiple browser window can be opened showing different part of the system at the same time.

Web-based system also faces some challenges compared to conventional single application:

* The system must be designed with multiple session at the same time, as by nature, browser can open multiple page to the server.
* Sending live data to the client is more challenging as conventional web page are based on a request-response architecture.
* JavaScript-based browser environment is limited in terms of processing power.
* Web-based application has additional latency, where the data needs to be transferred to the client.
* Web-Application generally assume that the Web-Server can be duplicated in order to increased throughput. Multiple Web-Server must serve the same data.

## 1.1 Problem statement

Conventional dedicated application is not flexible in terms of UI and client compatibility. Additionally, a dedicated application needs be installed. Real time update for a web application is more challenging compared to a standalone application because of the stateless nature of a web server and the conventional request-response architecture.

## 1.2 Objective

Develop a prototype web-based central monitoring system for the AGVs and study its feasibility and performance.

The system should be able to:

* Connect with the AGV through Wi-Fi
* Collect various readings such as temperature and battery voltage.
* Estimate the location of the AGV with a series of RFID detection
* Remotely configure the AGV.
* Remotely control the AGV.

## 1.3 Methodology

First, suitable web-based framework is chosen to help with the development. Meteor [2] was chosen as it enables real-time data updates to the client relatively easily. Meteor is a full-stack NodeJS [3] framework. NodeJS is known to be able to serve large number of concurrent user because of its event based design [4].

Then, the communication between the prototype Arduino board and the server is developed. The Wifly command interface was studied and later the best scheme of connection is implemented. Later on, the development board is swapped with a line following robot. The later development was largely incremental. More feature was added, fine-tuned and optimized.

Once the development was largely complete, the system is tested with a running robot, and load tested with multiple simulated AGVs.

# RELATED WORKS

Gregg Podnar, John M. Dolan, Kian Hsiang Low and Alberto Elfes in their 2010 paper [5] present a telesupervised fleet of water quality sensing autonomous boats. The boats send various sensor readings while moving through a river. It communicates with the server via a cellular technology.

J. Lopex, Diego Perez, Enrique Pax and Alejandro Santana in their 2013 paper [6] present WatchBot: A building maintenance and surveillance system based on autonomous robots. The autonomous robots move throughout the building as a surveillance systems and can responds to alert. It communicates through the system Wifi. The system works with a classic building automation system and exposes a web-based interface.

Yong Ming Wang, Hong Li Yin, Nan Feng Xiao, Yan Rong Jiang in their 2009 paper present an internet-based and sensor-driven architecture [7] which guarantee a non-distortion transfer of control information and reduce the action time difference between local simulated virtual robot and remote real robot.

Ahmad Aljaafreh, Majdi Khalel, Islam Al-Fraheed and Kafa Almarahleh in their 2011 paper [8] present a web-based fleet management automation system. The system uses On-Board Diagnostic (OBD), GPS, RFID and WIFI to transmit vehicular data to a server.

R. Ganesan and S. Mydhile in their 2013 paper presents a web-based remote vehicle monitoring system that is also context aware [9]. The system changes its behavior depending upon the context that ensure the challenges of context aware web services.

Dragon Stojanovic, Batislav Prediv, Igor Antolovic, Slobodanka Dordevic-Kajan (2009) [10] presents a Web Information System for Transfer Telematics and Fleet Management. The Web-Based system is based on MOVIS – a service-oriented open software platform for location-based and context-aware mobile and Web applications.

John Paul Antony and Dr. S. Rajpandian in their 2014 paper [11] present a Web-Based Control and Monitoring of Telepresence Robot. Their web user interface based on JavaScript and PHP is able to remotely control the TechRobot Create mobile robot.

# ARCHITECTURE

The system consists of four main components, the prototype AGV, machine interface server, web server and client. Communication between AGV and machine interface server uses TCP Connection. Data between machine interface server and web server is synchronized in real-time using MongoDB. The web server and the client communicate through WebSocket or HTTP polling. The machine interface server and web server are designed to work in multiple instances.

Figure 1. Possible configuration

Figure 1 shows a possible configuration for the system. The AGV should be able to connect to multiple Wi-Fi hotspot where available as long as it has access to the application server. There could be multiple machine interface server behind a TCP load balancer such as Nginx or HAProxy. The application servers could reside in multiple physical machine. As long as they connect to the same MongoDB server, data should be synchronized.

## 3.1 Prototype AGV

The Arduino based line following robot uses Arduino Mega, Wifly-RV module, 2Amp motor controller, RFID Reader (RFID-RC522), LM35 temperature sensor, Auto-Calibrating Line Sensor (LSS05) and several other components. The robot uses its EEPROM to store configurations. Some configuration can be configured through serial and other through Wi-Fi. The schematics is shown in Figure 3.

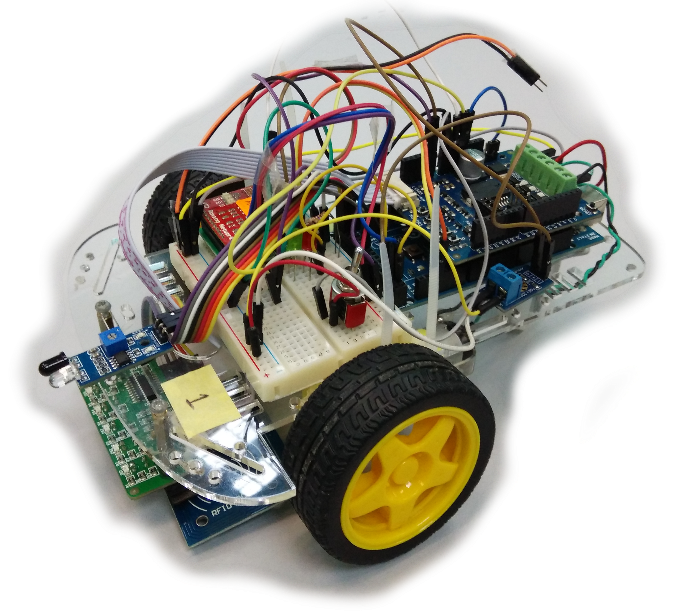


Figure 2. The prototype AGV

The prototype AGV communicate with the server through Wi-Fi using a custom TCP protocol. UDP was shown to be too unreliable with poor response time. WebSocket connection was also experimented, but due to the unreliable connection state, it was abandoned in favor of a custom TCP protocol.

The Arduino sends several readings, namely temperature, battery voltage, loop latency and obstruction detection with the use of an infrared obstacle sensor. In addition to that, it also sends its status such as if it has been detected to wander out of its circuit, if an RFID tag was detected and if it is in manual model. The Arduino board is also programmed to return ping messages. The ping message is used to measure the AGV’s response time. Temperature, battery voltage and loop latency is sent every second. While other reading is sent when the respective condition occurs.

Currently the task of the robot is only to follow a black line in a circuit. The robot detects the black line using an array of infrared sensor mounted at the front of the robot. The readings are fed through a PID controller and translated to the motor parameters. Mounted next to the array of infrared sensors is an RFID sensor. Throughout the circuit, there are RFID tags meant to be detected by the RFID sensors. The RFID tags helps as a hint to calculate the current position of the AGV. When the AGV detect an RFID tag, it will send the RFID’s UID to the server, which will log its position. Using a combination of the last RFID log, estimated speed of AGV based on previous reading, next expected RFID tag, the server can estimate the location of the AGV. The prototype robot does not have a GPS system. Furthermore, the circuit is quite small, smaller than what conventional GPS system can accurately measure.

The prototype AGV have two LED, red and green. On startup, the red LED light ups while the Wifly module is being configured. Once it is done, the red LED will start blinking. The green LED will light up when the AGV is connected to the server.

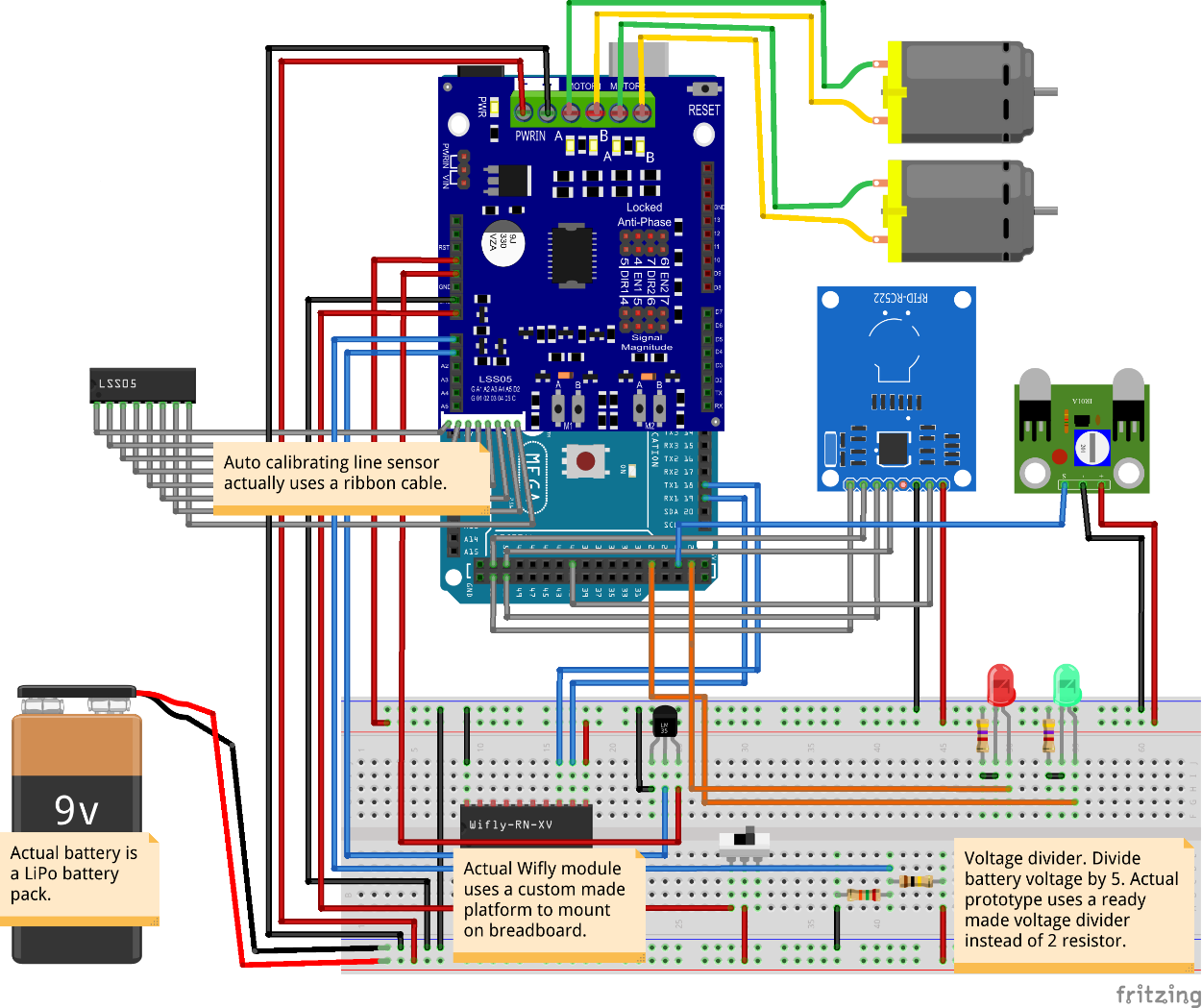


Figure 3. Schematics of the prototype AGV



Figure 4. RFID reader next to an array of infrared sensor.



Figure 5. Prototype Circuit

## 3.2 The system

The system is a single Meteor codebase which is used for three purpose, the client side Single Page Application (SPA), the web server, and the machine interface server. Meteor is a full-stack NodeJS framework. It provides many common utilities used in a modern web-based application. The main factor of choosing Meteor is because of its integrated publish-subscribe system which allow easy real-time data transfer to the client. Additionally, it allows the client and server to use a single codebase. The use of a single codebase helps to eliminate the needs to duplicate the same code across multiple codebase. Meteor also exploits MongoDB’s oplog to enable changes in the database to be reflected on the client in realtime. This features enable a real-time web-based application to be prototyped quickly.

Communication between the client and the web server uses Meteor’s DPP protocol which works on top of SocksJS [5] which works on top of WebSocket or HTTP polling where available. Communication between the AGV and the machine interface server uses a custom TCP protocol. The TCP server uses NodeJS’s built in TCP server library.

## 3.3 Machine Interface Server and Web Server

The machine interface server is basically a module within the application server. Both the listening port for the web-server and machine interface server can be configured. This allow configurations where a server process only works as the machine interface server or as the web server.

Listening for new messages from the AGVs is the responsibility of the machine interface server. When a new connection is established, the machine interface server sends an ‘identify’ command to the AGV. The AGV will then send a message containing its machine id, which registers the AGV in the system. For every incoming messages, the machine handler emits various events which is listened by other modules.

For sending message to the AGV, the system maintains a command queue per AGV in the database. When an AGV is connected, the machine interface server listens to new item in the command queue and dispatch them when they are detected. This listening, which rely on Meteor’s use of MongoDB’s oplog incur some response time of roughly 8ms. Testing with a single server and single online simulated AGV, while bypassing the command queue shows that the response time to be about 2ms. When using the command queue, and a MongoDB with oplog exposed, the response time is more volatile ranging from 4ms to 20ms. When bypassing the command queue, the application server which send the command must be the same machine interface server that is connected to the targeted AGV. This limits the number of application server instance to one, which is the reason why it is not the default behavior.

Data from the AGV to the client work in similar manner. Event listeners from the machine interface server will do some processing such as denormalizing data, calculating estimated speed, etc before saving the result to the database. Data is transferred to the client like a normal Meteor application. The client subscribes to a part of the database. When the part of that database changes, the change is sent to the client.

At any moment, there should be a single master server which listen the serial interface which detect if the Arduino is connected via USB. This enables configuration through the system. The master server also periodically stores the server time into the database which is synchronized with every server and client.

All the sensor readings and location logs are saved together with their timestamp. This enables the replay functionality by calculating previous state with these records. The system does not have any form of authentication. Authentication is expected to be handled by lower level protocol such as HTTP basic authentication, or allowing access only in a certain network.

## 3.4 Client

The client is a Single Page Application (SPA) instead of the traditional request-response HTML page. It uses ReactJS [7] as the view model of the client. ReactJS is a library made by Facebook. It is a heavy advocate of component based design. ReactJS is a dramatic departure from traditional MVC architecture. Instead of a separate template file, it uses a special JavaScript syntax which enable HTML to be embedded. In addition to that, it encourages the use of inline CSS, which is generally also stored in the same file. The result is, each component’s code, script, template and styles is contained in a single JavaScript file which helps when the codebase become large. The system also make use of MaterialUI [8] which is a set of ReactJS components which implement Google’s Material Design Language. The use of Material UI helps make the UI friendlier to touch enabled device.

The client has several pages. Through the use of FlowRouter, a Meteor’s routing library which uses JavaScript’s History API, the SPA will change page depending on the URL. The pages generally have a ‘ViewTime’ indicator at the top right of the page. This component enables the client to switch between a ‘Live’ mode and a ‘Replay’ mode. In the ‘Live’ mode, the page will display the current state of the AGVs. In ‘Replay’ mode, the page will display the state of the AGV at the selected time. A client response time indicator is available at the top left corner of the screen.

### 3.4.1 Dashboard Page

The dashboard shows multiple Machine’s current readings and an overview map. The map shows the approximate location of the machine and also change appearance depending on the status of the machines. The readings update in real-time. The card title will change color depending on the AGV’s status. Additionally, a reading charts page is also available which shows a historical chart of selected readings for each machines.

### 3.4.2 Map Page

The map page shows a visual estimation of the AGV’s position relative to the whole circuit. The map is drawn using SVG. Additional visual cues can be added through third party vector image application such as Inkscape or Adobe Illustrator. The position of each AGV is tracked using several calculations. The LocationLog inserted when an RFID tag is detected contains the current path id, current path progress and direction (forward or backward of the path). An estimated speed is also calculated based on previous LocationLogs and a speed hint stored for the path. Using the difference between the current time and the LocationLog’s timestamp, the new current progress is calculated. An SVG path’s data is stored for each path in an internal object. With the data and a built in SVG function, an X-Y coordinate is obtained which is used to determine the position of the AGV on the map. additional readings are also used to determine if the particular AGV stopped for some reason at some point.

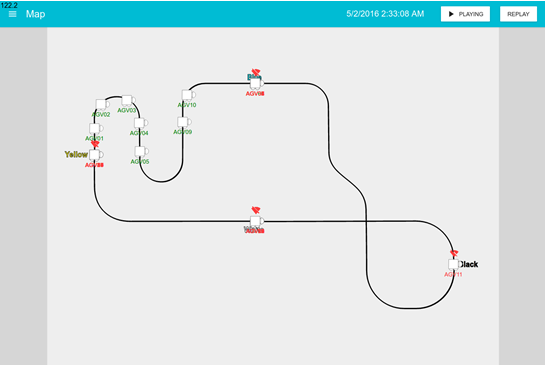
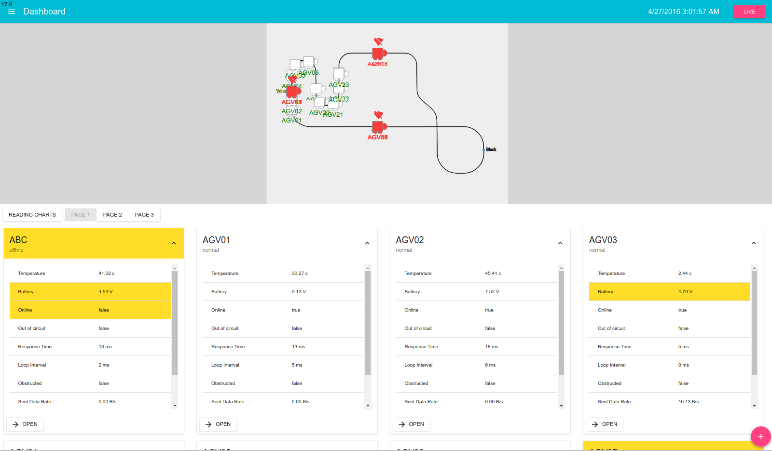


Figure 7. Dashboard Page

Figure 7. Map Page

### 3.4.3 Machine Page

The machine page shows per AGV information. It split into three tabs, which are the status tab, the message log tab and the manual tab. It can be accessed by a button on the card in the dashboard or by clicking on one of the AGV icon on the map.

The status tab shows the status of that AGV. It lists down every reading. Each of the readings can be clicked to open a historical chart of that readings. A button is also available to configure the AGV’s configuration. The configuration is sent through Wi-Fi. The message log tab shows a list of messages from the machine sorted in decreasing order of arrival. It is mostly used for debugging. The manual tab is used to control the AGV manually. When the button is clicked, a command is sent to the AGV which causes it to enter manual mode. In manual mode, the AGV will no longer follow the line in the circuit. It will now follow the command given by the four button. This allow the user to remotely control the AGV. All the tabs are responsive in design. It will change layout when the window size changes.

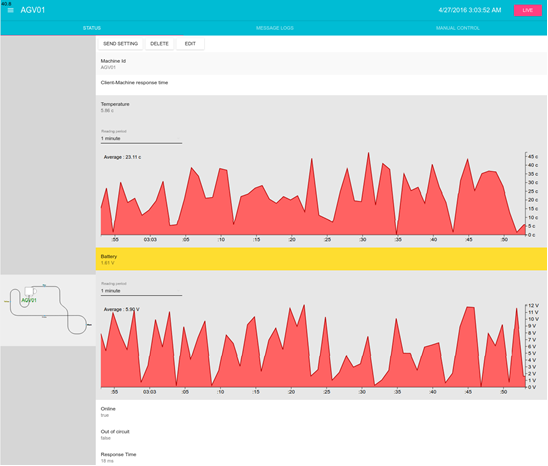


Figure 8. Machine Page

### 3.4.4 Message Log Page

The message log page shows incoming messages from all AGV. It’s very similar with the Machine Page’s Message Log tab. However, it lists down every message from every AGV.

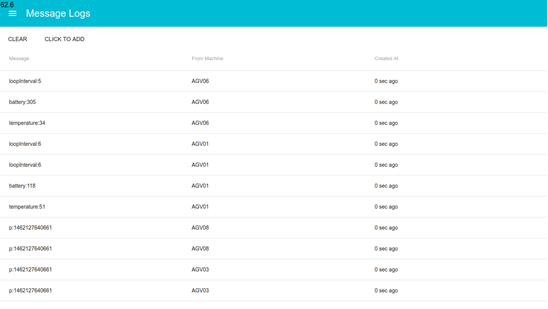


Figure 9. Message Log Page

### 3.4.5 Configuration Page

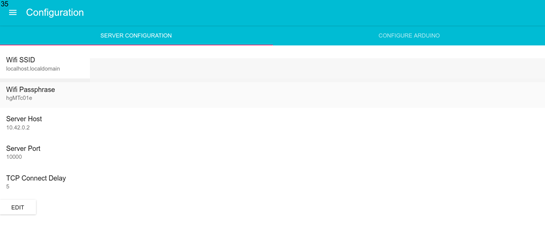
Configuration Page allow the user to set various global configuration. These configurations are generally used by the AGV. Additionally, in the Configure tab, the user can configure the AGV through serial. If the AGV is connected, a list will appear in the tab. Click on the list, then select an AGV to configure the connected machine as the selected AGV.

Figure 10. Configuration Page

## 3.5 Custom TCP Protocol

The custom TCP protocol is a simple newline-delimited message-based protocol. Messages are usually a single word or a key-value pair separated with a colon “:”. Although it is a TCP-based protocol, it assumes a connectionless underlying transport protocol. This is done for two reason. First, the Wifly module does not detect the TCP connection state reliably. Second, when the AGV is turned off, it would take a significant amount of time for NodeJS’s TCP server to consider the connection as disconnected.

When no connection is detected, the AGV will send a ‘hello’ message every second. If a response is received, the AGV will assume that it has been connected. If there are no message from the server for more than 5 second, the connection is considered severed.

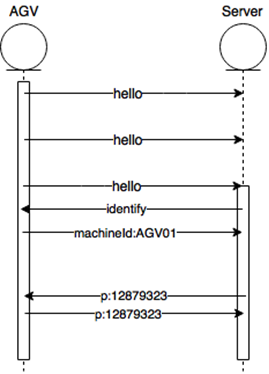
When a new connection is detected on the server, the server will send an ‘identify’ command to the AGV. The server will also keep replying ‘identify’ for every message received as long as no AGV is associated with that connection. The AGV will then reply with its machine id. The server will send a ‘ping’ message every second to the AGV to keep the connection alive. The AGV should send back the ‘ping’ message. If no message is received by the server within 5 second, the connection is considered severed. If a connection is made for a machine with the same machineId as an online AGV, the previous connection is closed.

Figure 11. Custom TCP Protocol Handshake

## 3.6 Database Structure

MongoDB does not enforce structure on the documents. However, the system does make use of a schema for most collection. Figure 12 shoes the database structure of the system.



Figure 12. The database structure of the system. Note that <reading> corresponds to each reading.

### 3.6.1 Machines

The main collection is the ‘Machines’ collection. Each AGV will have a corresponding Machines document. Each AGV is uniquely identified with a string called machineId. The machine document also contains the machine’s configuration such as its motor speed and PID parameters. The document also contains last value of each reading, along with their updated time. Table 1 list down the description of each fields.

|  |  |
| --- | --- |
| Field | Description |
| machineId | Unique machine id that identifies each AGV. |
| onlineOnServer | PID of the machine interface server process which the AGV is connected to. |
| onlineAt | The time the AGV is marked as online. |
| motorBaseSpeed | The maximum speed of the AGV motor. 255 is the theoretical maximum, but it is usually set to 100. |
| motorLROffset | Motor left right offset. Used to compensate for different motor speed. |
| motorPIDMultiplierRatio | Used within the line following algorithm. Usual value is 0.3. The higher the value, the more the AGV will turn in respond to it current PID value. |
| motorVoltageCompensation | Used to compensate for differing motor speed when the battery is low. Normally unused by setting it to 0 because the motor speed is not linear which result in an overly high compensation. |
| motorDiffRange | The maximum difference between the left and right speed of the motor. Used in motor speed calculation in the line following algorithm. |
| PID\_Kp | The PID proportional parameter. |
| PID\_Ki | The PID integral parameter. |
| PID\_Kd | The PID differential parameter. |
| lastLocationLog | A copy of the last LocationLog. |
| <reading> | The last value for each reading. |
| <reading>UpdatedAt | The time the last value is updated, even if it does not change. |
| <reading>StartUpdatedAt | The time the last value changes. |

Table 1. Machines collection fields

### 3.6.2 Readings

Readings are the state of the AGV. All the sensor readings are saved together with their timestamp. This enables the replay functionality by calculating previous state with these records. Consecutive readings with similar values are reduced to save space and bandwidth. If three consecutive readings have the same value, the middle reading is removed. Table 2 list down all readings.

|  |  |
| --- | --- |
| Reading | Description |
| online | A Boolean. True if the AGV is online, false otherwise. |
| battery | The battery voltage. |
| temperature | The temperature reading. |
| outOfCircuit | A Boolean. True if the AGV is out of its track. |
| responseTime | The response time between the server and the AGV. |
| loopInterval | The time between the loop() call of the Arduino. Useful to determine the performance of the AGV and for debugging. |
| sentDataRate | Number of bytes per second from the server to the AGV. |
| receivedDataRate | Number of bytes per second from the AGV to the server. |
| manualMode | A Boolean. True if the AGV is on a manual mode. False otherwise. |

Table 2. List of readings

These ten readings use largely the same code. Each of them have their own separate Collection, but use the same code on most functionality. The common code includes the code to determine each of these value at a particular time, the historical graph, listening for events from the connection handler and others. Each of this reading is set when a particular key-value message from the AGV is received, where the key is the name of the reading. For example, to send the loopInterval with value 15 milliseconds, the AGV would send the message ‘loopInterval:15’. Some readings are set by other modules. Those readings are online, responseTime, sentDataRate and receivedDataRate.

Because of the largely common code, adding a new reading only require adding a string to an internal array and modifying a meta object. The meta object contain the human readable reading name, the reading name type, unit, badHigh/badLow which indicate an abnormal value and a transformer that transform the value from the AGV into a correct representation. For example, the battery voltage sent from the AGV is the actual integer analog value instead of the actual voltage. The conversion to actual voltage is done with a transformer which does the necessary calculation.

### 3.6.3 LocationLogs

The LocationLogs logs the location of each AGV at a particular time. The following is the description for each field.

|  |  |
| --- | --- |
| Field | Description |
| machineId | The id of the AGV. |
| type | ‘point’ or ‘path’. ‘point’ means the AGV is at a particular point and it is not moving. ‘path’ means the AGV is on a path and is moving. Currently, the system only use ‘path’. |
| firstInterruption | The time where the LocationLog is first interrupted with an event that will invalidated position estimation. For example, the time where the AGV is out of circuit. The system should not assume the AGV continue on its path after the time recorded here. firstInterruption is recorded on the last LocationLog when either obstructed, outOfCircuit or manualMode reading is recorded. |
| pointId | Set when the ‘type’ is ‘point’. The id of the ‘point’. The id depends on an internal Map object. |
| pathId | Set when the ‘type’ is ‘path’. The id of the ‘path’. The id depends on an internal Map object. |
| pathDirection | 1 if the AGV is moving with the same direction as the path. -1 if the AGV is moving with the opposite direction. |
| pathProgress | A number between 0 to 1 which is the location of the AGV in the path. |
| nextEstimatedSpeed | The estimated speed of the AGV in progress per second. |

Table 3. LocationLogs collection fields.

### 3.6.4 CommandQueue

Each of the AGV have a corresponding command queue. The command queue is a separate collection than the Machines collection for performance reason. Each command queue has a ‘commandQueue’ field which is an array of object with the following structure:

|  |  |
| --- | --- |
| Field | Description |
| command | The message to be sent. |
| droppable | A Boolean. True if the message is droppable. A message is dropped if it takes too long to be sent. |
| createdAt | The time the message is queued. |

Table 4. CommandQueue item fields

Once an AGV is associated with a connection, the connection handler will start to listen for the AGV’s corresponding command queue. When there is new message, the command queue is emptied and the observed messages is sent. If the message’s timestamp is more than 5 seconds old and the message is droppable, the message is dropped.

### 3.6.5 MessageLogs

The MessageLogs collection logs every message received by the server. It contains three field:

|  |  |
| --- | --- |
| Field | Description |
| text | The message. |
| fromMachineId | Optional machineId that identifies from which AGV the message came from. |
| createdAt | The time the message is received. |

Table 5. MessageLogs collection fields

### 3.6.6 clientMachinePing

This collection is used to relay a ping from the AGV to the client. It is used in measuring response time from the client to the AGV and back. When measuring the client-machine response time, the client asks the server to send a ping to each AGV containing the client’s connection id and a timestamp. The AGV will then return the ping to the server which will store the message in this collection. The client subscribes to this collection and calculate the new client-machine ping when a new record is detected.

### 3.6.7 clientResponseTimeLogs and clientResponseTimes

These collections are used to measure response time between the server and the client. When a connection is made between the client and the server, the server will periodically create a new clientResponseTimeLogs document. The client subscribes to this collection. When client detect a new document, it calls a method on the server will the id of the document. The server will delete the document and calculate the client response time. The response time is calculated and added to the previousLogs fields of the document which is an array of response times. The number of element in this array is capped to 5. The actual response times is the average of this field. The client response time is stored in the clientResponseTimes collection which is subscribed by the client.

### 3.6.8 GlobalStates

GlobalStates is used to store singleton throughout the system. Each of the document have different schema. Currently the global states collection is used to store the current server time and the list of serial device connected with the same machine as the master server.

## 3.7 Serial and Wi-Fi Configuration

The AGV stores several configurations in its EEPROM. Some of the configuration can be configured through serial and some through Wi-Fi. The following is a list of all configuration stored in the EEPROM.

|  |  |
| --- | --- |
| Field | Description |
| wifiSSID | The SSID of the Wi-Fi. |
| wifiPassphrase | The Wi-Fi passphrase |
| serverHost | The address of the server. Usually the IP address of the server. |
| serverPort | The TCP port the machine interface server listens to. The default port is 10000. |
| tcpConnectDelay | How many second before the wifly module tries to reconnect to the server. The default value is 5 second. |
| machineId | The machine id. |
| motorBaseSpeed | The maximum speed of the motor. |
| motorLROffset | Motor left right offset. |
| motorPIDMultiplierRatio | Used in the PID calculation to translate from the PID value to the motor speed. The default value is 0.4. |
| motorVoltageCompensation | How much the baseSpeed is increased or decreased when battery voltage fell below or above 7V. |
| motorDiffRange | Maximum difference in speed between the left and right motor. |
| PID\_Kp | The PID proportional parameter. |
| PID\_Ki | The PID integral parameter. |
| PID\_Kd | The PID differential parameter. |

Table 6. List of stored configurations

The serial configuration protocol uses the same key-value convention used in the custom TCP connection. All commands end with a newline. Additionally, the AGV also respond to some command through serial. The following are the commands used to configure the AGV through serial.

|  |  |
| --- | --- |
| Command | Description |
| dump | Print all configuration through serial. |
| save | Save the current setting to the EEPROM. |
| load | Load the setting from the EEPROM. |
| reconfigure | Reconfigure the Wifly module according to the current settings. |
| wifiSSID:<SSID> | Set the Wi-Fi SSID. |
| wifiPassphrase:<pass> | Set the Wi-Fi password. |
| serverHost:<host> | Set the server address. |
| serverPort:<port> | Set the server port. |
| machineId:<machineId> | Set the machine id. |
| tcpConnectDelay:<delay> | Set the TCP connect delay. |

Table 7. Serial configuration commands

The configuration page can automatically send the configurations.

The Wi-Fi configuration follows the same protocol as the TCP connection. The following are the messages that is used to configure through Wi-Fi.

|  |  |
| --- | --- |
| Command | Description |
| saveSettings | Save the current setting to the EEPROM. |
| motorBaseSpeed:<speed> | Set the motor base speed. |
| motorLROffset:<offset> | Set the motor LR offset. |
| motorPIDMultiplierRatio:<ratio> | Set the motor PID multiplier ratio. |
| motorVoltageCompensation:<speed> | Set the motor voltage compensation. |
| motorDiffRange:<range> | Set the motor diff range. |
| PID\_Kp:<val> | Set the PID proportional parameter. |
| PID\_Ki:<val> | Set the PID integral parameter. |
| PID\_Kd:<val> | Set the PID differential parameter. |

Table 8. Wi-Fi configuration commands

The system will automatically send the commands when the edit machine form is saved.

## 3.8 Line Following Algorithm and Manual Mode

The AGV prototype operate in two modes, the line following mode and the manual mode. By default, it is in the line following mode. If the manual mode flag is set, it will bypass the line following algorithm and execute the manual mode procedure. To set manual mode on or off, the server will send the message ‘enterManual’ or ‘exitManual’.

### 3.8.1 Line Following Algorithm

The line following algorithm uses an array of infrared sensor that can detect the color of the surface under it. There are 5 infrared sensors in total. From the five sensor, a value is determined ranging from -4 to 4, each correspond to how much the AGV should turn. A value of -4 means the AGV should turn left. A value of 4 means the AGV should turn right. If the value is 0, the AGV should go straight.

The value is inputted into a PID Controller which parameters is obtained from the configuration. The PID Controller will then output another value from -5 to 5, which also correspond to the direction the AGV should turn. The left and right motor speed follows Equation 1.

Equation 1. Motor speed calculation

Where *base* is the motorBaseSpeed after voltage compensation, *diff* is the motorDiffRange, *multiplier* is motorPIDMultiplierRatio and *value* is the value outputted by the PID controller.

If the sensor pattern does not correspond to any value, the AGV will skip the line following algorithm, with the motor still using the previous speed. If the situation continues for more than 2 second, the AGV is considered out of circuit. It will stop the motor and send a message to the server notifying it that it is out of circuit. This prevent the AGV from wandering far outside its intended track. If the AGV is moved physically, and a valid sensor pattern is detected, it will continue on the path and notify the server that it is no longer out of circuit. If the obstacle sensor detects an obstruction, it will stop and notify the server of its situation. When the obstruction is removed, it will continue its movement and notify the server that it is no longer obstructed.

### 3.8.2 Manual Mode

The manual mode enables the user to remotely control the movement of the AGV. The following are the commands from the server that control the movement in manual mode.

|  |  |
| --- | --- |
| Command | Description |
| manualLeft | Rotate counter-clockwise |
| manualRight | Rotate clockwise |
| manualForward | Move forward |
| manualBackward | Move backward |
| manualMotorL:<speed> | Set the left motor speed to <speed> |
| manualMotorR:<speed> | Set the right motor speed to <speed> |
| manualStop | Stop the AGV. |

Table 9. Manual mode Wi-Fi commands

Internally, the AGV store the current left and right motor speed along with the last time the speed is set. If the last command is received more than 1 second ago, the AGV will stop. This means the client needs to repeatedly send movement command as long as the AGV should not stop. Currently, the interval between a repeated command is 500 milliseconds. The repeated command prevents the AGV from continuously moving when the client or the AGV is disconnected.

## 3.9 Map and Location

The system keeps an internal Map object for the use of drawing and positioning. The coordinate system of the Map is a plane of width 1200 units and height of 900 units. The Y coordinate goes from top to bottom. The Map coordinate is analogous to SVG coordinate, as that is what used to draw the map and trace the AGV’s position.

The Map contain three items, list of points, list of path and an extraSVG string which can store arbitrary SVG to be drawn on the client. Each point contains three attribute, id, visualX and visualY. The visualX and visualY is used to position the AGV when its location is at that particular point. Each path contains three attribute, id, svgPathD and machineSpeed. The svgPathD is the content of the ‘d’ attribute when the path is drawn on the SVG plane as a path element. The machineSpeed is the estimated speed of the AGV in units per second on that path.

To get the position of the AGV when it is on a path at a particular time, first the client calculates the current progress of the AGV at that path. The calculation roughly follows the following formula:

Equation 2. Path progress calculation

The *progress* is capped between 0 to 1. *logProgress* is the pathProgress on the last location log*. logDirection* is the direction on the last location log. *firstInterruption* is the first interruption (an obstruction, outOfCurcuit or manualMode reading) after the last log and logTime is the time the last location log is logged.

Once the progress is obtained, the client uses a built in SVG function that return X-Y coordinate based on how ‘far’ in the path is the AGV, to get the position to draw the AGV graphic on the map. To animate the AGV, the client simply recalculates the position several time every second.

The estimatedSpeed is precalculated on the last location log. It uses the previous LocationLogs to estimate the current speed. If no data is available, it uses the built in path’s estimatedSpeed. The estimatedSpeed is calculated by iterating on the previous 20 location logs. If two consecutive log is on the same path, and there is no interruption between them, then the estimatedSpeed can be calculated by using the difference of progress between the two log and the difference in timestamp. On average, there should be about two valid speed estimated using this technique. Because the estimatedSpeed depends in previous speed, the AGV needs to run through the circuit before the speed become accurate.

The estimated speed calculation works with how the RFID reading is interpreted. The system assume that the AGV is moving through the circuit continuously in one direction. When an RFID tag is detected, the AGV send the RFID’s UID to the server. Based on the UID, the server will log two LocationLog. One to indicate that it is at the end of a path and the other to indicate that it is starting a new path. The combination of these two enable the previous speed estimate. In theory, to implement a more complex movement system where the AGV may move to more than one direction, only this module needs to be modified. The rest of the system should work as it is.

# PERFORMANCE

The following are various metrics that represent the performance of the system. Unless otherwise stated, all tests were done using a machine with the following specifications:

|  |  |
| --- | --- |
| CPU | Intel i5 4690K Overclocked to 4.3 Ghz (Quad code, no hyperthreading, turbo boost disabled) |
| RAM | 16GB 1600Mhz DDR3 |
| Drive | 120 GB Kingston V300 SSD |
| OS | Ubuntu 16.04 |
| Software Versions | MongoDB 3.2, Meteor 1.3.2.4, NodeJS 4.4.3, Nginx 1.10, Google Chrome 50. |

Table 10. Test environment

All tests were done using a production build of the system. The TCP and HTTP reverse proxy used is Nginx 1.10 bundled with Ubuntu. Only one MongoDB instance is used. Unless stated otherwise, all AGV are simulated AGV running on the same machine. The simulated AGV will send random temperature, battery reading and loop interval every 1 second. To simulate movement, the simulated AGV cycle between RFID readings and send it every 5 second. At any moment, 50 AGV is registered at the system. The following are description of various metrics presented.

|  |  |
| --- | --- |
| CPU Usage | Per core CPU usage. Presented values is the estimated average, guessed manually from the readings of the ‘htop’ utility. |
| Memory Usage | RAM usage used. Presented values is the estimated average, guessed manually from the readings ‘RES’ column of the ‘htop’ utility. |
| Response time | Response time between the server and the AGV is measured by periodically sending a ping to the AGV and calculating the time difference when the ping returned. |
| client response time | Response time between the server and client. Measured by periodically saving a ping document which is subscribed by the client. The client will call a server method when a new ping is detected. The server method then, calculate the time taken for the method to be invoked. |

Table 11. Metrics

## 4.1 Machine Interface Server and Web Server

Machine interface server and web server are the same application server. The application server can run on a different physical server as long as it can connect to MongoDB. There is no significant difference between the memory usage of the machine interface server and web server. Because the server is JavaScript based, it always run on a single thread. The following are the idle metrics for the server, given that no AGV or client is connected.

|  |  |
| --- | --- |
| Idle CPU Usage | 0-1% |
| Idle Memory Usage | 70-100MB |

Table 12. Idle stats

### 4.1.1 Machine Interface Server

The machine interface server shows good scaling over multiple server instance. However, every server instance is limited to about 40 AGVs before it starts to drop messages.

|  |  |  |
| --- | --- | --- |
| No of AGV | CPU Usage | Response time |
| 1 | 3.8% | 8.0ms |
| 10 | 28.0% | 16.5ms |
| 30 | 77.0% | 30.0ms |
| 50 | 100.0% | 5760.0ms |

Table 13. Machine interface server CPU performance

Using three server instance shows better response time, but higher overall CPU usage.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No of AGV | Estimated AGV per server instance | Average CPU Usage for all server instance | Total CPU Usage | Response time |
| 1 | 0.33 | 2.0% | 6% | 11ms |
| 10 | 3.33 | 14.0% | 42% | 12ms |
| 30 | 10.00 | 39.0% | 117% | 14ms |
| 50 | 16.66 | 61.0% | 186% | 30ms |

Table 14. Machine interface server performance with 3 server instance

The higher overall CPU usage can be explained by Meteor’s use of oplog in which each server instance need to process the writes for every MongoDB operation. This can be shown by the idle server CPU Usage over number of AGV. The AGV is connected to other machine interface server, however, the idle server still shows an increase in CPU usage. RAM usage does not show any clear pattern. None of the server uses more than 200 MB of RAM throughout the test.

|  |  |
| --- | --- |
| No of AGV | CPU Usage |
| 1 | 1% |
| 10 | 6% |
| 30 | 14% |
| 50 | 22% |

Table 15. Idle CPU usage per connected AGV

The following is the MongoDB CPU Usage. As the number of connected AGV increase, the CPU usage increase. The CPU usage does not seem to be affected by the number of connected server, rather it is proportional to the number of writes operation, which is triggered by every online AGV.

|  |  |
| --- | --- |
| No of AGV | CPU Usage |
| 1 | 2.4% |
| 10 | 13.5% |
| 30 | 35.0% |
| 50 | 50.0% |

Table 16. MongoDB CPU usage

The CPU usage of MongoDB seems to suggest that the maximum number of connected AGVs it can process is about 100 AGVs. At that point, the MongoDB is the bottleneck as the writes operation of a MongoDB database is limited to a single thread. MongoDB can use multithreading for read operations. Its multithreading capability is not tested due to the lack of processing capability of the test machine.

### 4.1.2 Web Server

The web server load largely depends on the particular page the client is on. This is because some page request more data than the others. Unlike traditional web page, the SPA continuously receives new data as it becomes available. The Dashboard in particular is very heavy in bandwidth. The following metrics is done with 1 machine interface server with 10 AGV running. To reduce browser’s CPU load, the tab is hidden, but not closed.

The first test is the server load when opening the Dashboard page. The web server CPU load usage is recorded over number of open page. Each page uses about 25KB/s of bandwidth.

|  |  |
| --- | --- |
| No of open page | CPU Load |
| 1 | 12.5% |
| 10 | 27.0% |

Table 17. Web-Server CPU load on Dashboard page

At no of open page greater than 10, the browser’s CPU usage is too high for the server to give accurate result. (More on the client’s CPU usage later).

The second test is using the machine page. The machine page is less demanding compared to the dashboard page as it only need to load one AGV’s data. The page uses about 5KB/s of bandwidth.

|  |  |
| --- | --- |
| No of open page | CPU Load |
| 1 | 8.0% |
| 10 | 14.6% |
| 20 | 25.6% |

Table 18. Web-Server CPU load on Machine Page

The following is the same test with the machine page, with number of open page opened equally to two web server.

|  |  |  |  |
| --- | --- | --- | --- |
| No of open page | No of open page per server | Average CPU Load per server | Total CPU Usage |
| 1 | 0.5 | 7.0% | 14% |
| 10 | 5.0 | 11.0% | 22% |
| 20 | 10.0 | 16.5% | 33% |

Table 19. CPU usage on Machine page with two server instance

The web server usage seems to show a good performance. Based on the dashboard page’s performance, it is easy to expect a server instance to handle at least 40 concurrent users. This is without considering idle CPU usage taking up some of the performance reading. The two server test also shows that the load could be distributed evenly among server instances.

## 4.2 Client

The client is served by a Single Page Application. The total payload is about 2.5MB. However, by default it is compressed, reducing the size to about 408KB. The client communicates with the server with a compressed WebSocket connection, or HTTP polling. HTTP long polling is not enabled in Meteor due to issues in some browser. The following are the client response time according to connection type. The client response time is measured on the Dashboard page.

|  |  |
| --- | --- |
| Connection type | client response time |
| Compressed WebSocket | 4-14ms |
| Uncompressed WebSocket | 4-14ms |
| HTTP Polling | 20-50ms |

Table 20. client response time over connection type

There do not seem to be a significant difference between a compressed WebSocket connection and uncompressed WebSocket connection. The browser is run on the same machine as the server, which means bandwidth is not limited. HTTP Polling however, shows consistently worst result.

We also measure the client-machine response time of the system. client-machine response time is recorded by sending a timestamp from the client to the server which then passes it to the AGV. The AGV will return the message. A module on the server then, save the message which is subscribed by the client. Once the client receives the message, the difference between the current time and the timestamp is calculated as the client-machine response time. client-machine response time is not enabled on previous test as it is very resource intensive.

|  |  |
| --- | --- |
| Client-Machine Response Time | 10-40 ms |

Table 21. Client-Machine response time

The client SPA consume high amount of CPU time. Profiling shows that the issue is due to multiple reason.

* MaterialUI’s method of calculating its style is very inefficient.
* Currently, inline CSS perform worst compared to using a stylesheet.
* ReactJS shadow DOM render function are run quite often, especially because the map re-render every 200ms to animate the AGV.
* Meteor’s minimongo are surprisingly inefficient. Minimizing loaded data seems to fix some issue.

None of this issue would be a significant problem in a normal web application. However, this system involves streaming tens of updates to the client per second. Additionally, calculating the position of the AGV can be CPU intensive. This shows the limitation of a web-based system where JavaScript based environment is inherently handicapped compared to a statically typed environment.

Various optimization has been made to mitigate this issues such as precalculating estimated speed on server and denormalizing readings and last location log into the machine document. Additionally, the update rate has been limited. Aside from that, pagination was added to the Dashboard page to limit the number of updates the page is subscribed to.

The following are the CPU usage of various page. The number of AGV is varied to show the effect of having online AGV. The dashboard is limited to 12 machine per page.

|  |  |  |  |
| --- | --- | --- | --- |
| Page | 0 online AGV | 5 online AGV | 30 online AGV |
| Dashboard | 30% | 40% | 70% |
| Map | 40% | 42% | 62% |
| Message log | 3% | 72% | 102% |
| Configurations | 3% | 3% | 12% |
| Machine Page | 8% | 18% | 20% |

Table 22. Browser CPU usage

The Map page does not have pagination and it will still need to render offline machine, which would explain its high CPU usage when no AGV is online. The total number of AGV in the system is 50. The Dashboard page have a mini map with similar behavior as Map page. However, the page also shows the statuses of the AGVs. It also has pagination. The Machine Page only subscribe to a single AGV, which is the reason why the difference between 5 online AGV and 30 online AGV is not much. However, there are consistent increase in CPU load. The exact reason is still unknown.

Due to the high CPU usage, most of the pages are not suitable to be used with a mobile device. Using a more efficient UI framework would improve the CPU usage dramatically. Additionally, a more fine-grained subscription scheme would help. In the end, the limitation is still there. A web based application simply will not be able to compete with a native application in terms of client-side efficiency. However, as web technology becomes more important, further improvement in web technologies would reduce the gap.

## 4.3 Prototype AGV

The following are some of the recorded metric from the prototype AGV.

|  |  |
| --- | --- |
| Average sent data rate (to the AGV) | 16 B/s |
| Average received data rate (from the AGV) | 72 B/s |
| Average response time through Wi-Fi | 50ms |
| Loop interval | 27ms |

Table 23. Prototype AGV stats

Average response time is recorded with the AGV connected to a laptop acting as a Wi-Fi hotspot. The server is connected to the laptop also through Wi-Fi. Loop interval is the time between the Arduino's 'loop' function call.

# CONCLUSION

The current system can handle moderate number of AGV. There are several limitations that prevents it from being suitable for a larger scale:

* Meteor's lack of support for sharded MongoDB configuration with oplog support.
* High idle CPU usage on heavy writes operations.
* MiniMongo's poor performance.
* MaterialUI's inefficient styling system.
* The use of ReactJS for rendering map is CPU intensive.

Using alternative tools instead of specified above could alleviate some of the issues, but may incur additional development time. This system suggest that a web-based AGV fleet management system can be developed successfully. However, some aspect of the system need to be considered.

## 4.1 Data synchronizations

A web-based system conventionally depends on horizontal scalability to provide capacity. Because of this, there needs to be some way of synchronizing data between server instances. In a normal web-based application, this synchronization is accomplished through the use of a centralized database system. This works well with a traditional request-response web system.

In real-time application however, there needs to be a way to actively send the data to other server and to the client to provide low response time. In this prototype system, this is accomplished through Meteor's exploit of MongoDB's oplog and through the use of the DPP protocol. However, MongoDB's oplog is limited to a single primary server. This limits the write capacity of the system. Additionally, each server instance would need to process all write operation of the database, even if it is irrelevant to them.

Inter-server notification may be better served through IPC call, however, that would complicate the system's configuration. A simpler solution would be to use (or partly use) a database system that officially support the publish and subscribe pattern. Redis, for example, officially support publish and subscribe pattern which would allow server to be notified when a particular data the subscribed to has changed [9]. The server can subscribe for each connection and receive notification when new data arrived. Then, the connection handler can gather other necessary data before sending it to the client. Another alternative database is RethinkDB which is designed with real-time updates in mind.

## 4.2 Client Processing Capability

It is no secret that the web environment is growing in importance. Because of this, major IT company invest in improving browser's capability. Node.js impressive performance is in no small part due to its use of Google's V8 JavaScript engine. However, there are still significant performance gap between a web environment and a native environment.

The prototype shows that creating a real-time capable Single Page Application could mean high CPU usage. Particularly if the page needs to be updated frequently. Therefore, the application needs to be carefully designed. Additionally, certain client has significantly lower processing capability then others, notably mobile devices. The system design needs to take this into consideration.

A native version of the application may be considered to alleviate this issue. However, that would mean increasing the cost of development and at the same time, nullify some of the benefit of a web-based system.

# References

|  |  |
| --- | --- |
| [1] | K. Schilling, M. M. Arteche, J. Garbajosa and R. Mayerhofer, "Design of Flexible Autonomous Transport Robots for Industrial Production," in *Industrial Electronics*, 1997. |
| [2] | "Build Apps with Javascript | Meteor," Meteor Development Group Inc., [Online]. Available: https://www.meteor.com/. [Accessed 2 May 2016]. |
| [3] | "Node.js," Node.js Foundation, [Online]. Available: https://nodejs.org/en/. [Accessed 2 May 2016]. |
| [4] | M. Fasel, "Performance Comparison Between Node.js and Java EE," 23 October 2013. [Online]. Available: https://dzone.com/articles/performance-comparison-between. |
| [5] | G. Podnar, J. M. Dolan, K. H. Low and A. Elfes, "Telesupervised Remote Surface Water Quality Sensing," 2010. |
| [6] | J. Lopex, D. Perez, E. Pax and A. Santana, "WatchBot: A building mantainance and surveilance system based on autonomous robots," *Robotics and Autonomous Systems 61,* 2013. |
| [7] | Y. M. Wang, H. L. Yin, N. F. Xiao and Y. R. Jiang, "Internet-based remote manipulation and monitoring of an industry robot in advanced manufacturing systems," 2009. |
| [8] | A. Ahmad, K. Majdi, A.-F. Islam and A. Kafa, "Vehicular Data Acquisition System for Fleet Management Automation," 2011. |
| [9] | R. Hanesan and S. Mydile, "Design and Development of Remote Vehicle Monitoring System using Context Aware Services," 2013. |
| [10] | D. Stojanovic, B. Prediv, I. Antolovic and S. Dordevic-Kajan, "Web Information System for Transport Telematics and Fleet Management," 2009. |
| [11] | J. P. Antony and D. S. Rajpandian, "Web-Based Control and Monitoring of Telepresence Robot," *Internation Journal of Innovative Research in Science, Engineering and Technology,* 2014. |
| [12] | "SockJS-client," [Online]. Available: https://github.com/sockjs/sockjs-client. [Accessed 2 May 2016]. |
| [13] | "A JavaScript library for building user interfaces | React," Facebook Inc, [Online]. Available: https://facebook.github.io/react/. [Accessed 2 May 2016]. |
| [14] | "Material-UI," [Online]. Available: http://www.material-ui.com/. [Accessed 2 May 2016]. |
| [15] | "Pub/Sub - Redis," [Online]. Available: http://redis.io/topics/pubsub. [Accessed 2 May 2016]. |
| [16] | "Replica Set Oplog," [Online]. Available: https://docs.mongodb.org/manual/core/replica-set-oplog/. [Accessed 2 May 2016]. |

1. [↑](#footnote-ref-1)