.

School Of Engineering

EEET 2166

Real Time Systems Engineering

**Final Project Report**

Nathan Williams S3707244

Phuc Nguyen S3636112

Hieu Huynh S3634147

Submission Due Date: 18th October 2019

# **Table of Contents**

[**Table of Contents**](#_1jxprvj36drl) **1**

[**Problem Statement**](#_pdfsmaazcnd0) **2**

[**Assumptions**](#_htikgatn2vi8) **2**

[**Solution**](#_bi1u6jhngwni) **2**

[**Fault Tolerance**](#_kvoy20luprb3) **10**

[**Development Process**](#_f85lsiu15ljr) **10**

[**Future Direction**](#_rejngaobn7qe) **12**

[**Reflections**](#_klnrospco7d8) **12**

[**Contributions**](#_1u9ajtv6n7te) **14**

[**Appendices**](#_ejmmogmjujg1) **14**

[**References**](#_8pyz2092a88f) **16**

# 

# Problem Statement

A traffic light control system must be constructed to control the traffic lights and boom gates involved in a series of adjacent intersections. This system must operate in real time with node synchronicity and ensure traffic flow is always maintained, even in cases of system failure. Secondary objectives involve maximising potential traffic flow, applying separate modes for on and off peak traffic, and creating an extensive test plan that can be executed prior to system deployment. All processes must be deployed over 4+ nodes each featuring an appropriate number of threads to ensure each task can be completed within it’s time critical window. Nodes should communicate using QNX native message passing to exchange information such as states and sensor pulses. Appropriate sensors and signals must be simulated using physical hardware when deployed to the test environment for demonstration.

The design will be based on a series of intersections occuring along Bell St in Coburg, Victoria. The system will operate 2 road intersections as well as a rail-road intersection (see initial design report **{IDR: fig. 1}**). Due to the close proximity of these intersections, consideration must be taken to ensure actions do not negatively impact traffic flow around adjacent intersections. All three roads and the rail line are particularly high traffic transport lines as they support a large portion of Northern Suburb inhabitant’s daily commute.

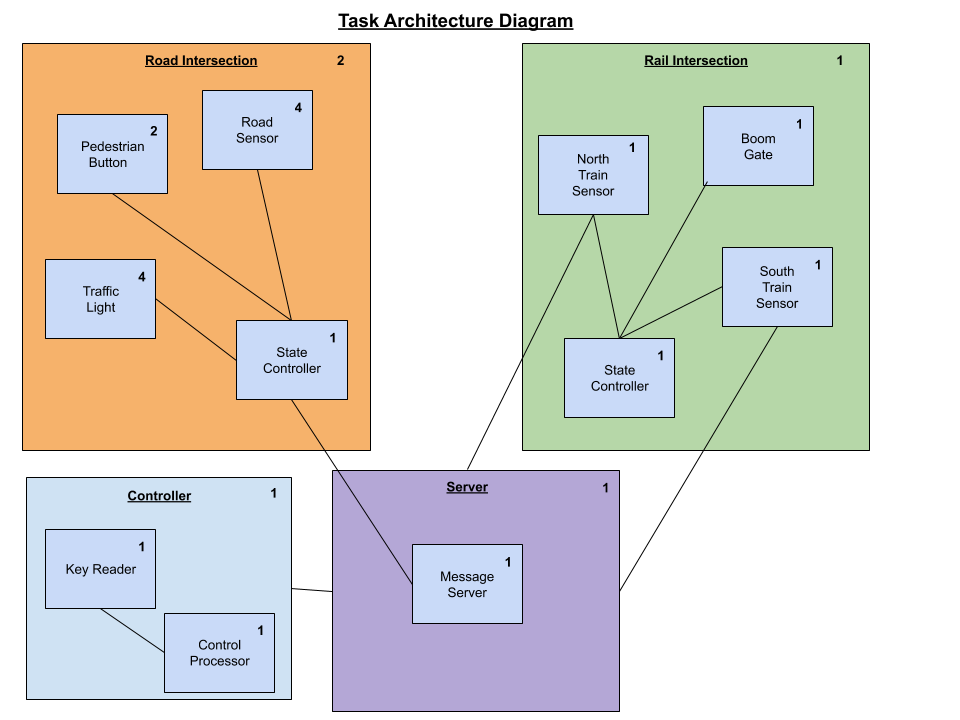
# Assumptions

* Available hardware includes multiple BeagleBone nodes and at least one i.MX node
* A system startup time will be allowed to ensure initialisation and synchronisation
* All nodes will be deployed on the same network
* The train track is bidirectional (i.e. trains can come from north or south)
* Each node’s controller has direct peripheral access to the simulated components (e.g sensor data and button triggers are directly read through I/O)
* Timer mode is designed for periods of high traffic flow in all directions equally
* Sensor mode is designed for periods of low or no traffic flow over all intersections

# Solution

**Overview**

Our design utilises 4 ARM nodes and 1 IMX node to control traffic flow over the specified intersection. Each road and rail intersection has an ARM node to control relevant lights and sensors. These nodes communicate with a server running on the fourth ARM node and can operate in timed mode, sensor mode and a range of fault sequences. A controller running on the IMX node is used to inject override commands via the server to all nodes. The layout of this system can be seen in fig. 1 and general system activity in fig. 6. This task architecture diagram has been slightly modified from the initial design report due to better understanding of message passing mechanisms and the division of the central controller into server and controller nodes.



**Figure 1: Task Architecture Diagram**

**Server**

The server utilises QNX native message passing to respond to a large set of signals and queries from all nodes. The passed message and reply structs shown in appendix A.

This structure allows an expandable framework capable of adapting to an almost unlimited series of event responses once added to code. By transmitting the clock time of the server, clients can synchronise their clocks to match the server time. Once synchronised, clients can utilise the transmitted event time to synchronise desired events. The data variable communicates which event should be executed at the event time, in response to triggering of sensors or other actions. Each event is stored locally on the server along with a modeSwitch variable. The event time is a second after the triggered sensor in majority of cases. This modeSwitch variable is for special circumstances involving a series of events e.g. Timed move of lights to N-S before switching from sensor to timed mode.

The server allows an initial cooling period for all nodes to connect and initialise however; there is fault tolerance discussed below to handle exceptions and late connections. Appendix B shows the series of events and situations the server responds to in table format. In essence, the server responds to sensors and actions with a confirmation. An event will then be created in the local store. The server expects all connected nodes to regularly poll (approximately every 100ms) for action. In response to this poll, the server transmits the stored event. If no relevant sensor was triggered then a poll is bounced back to confirm it’s been received but no event is required. Fig. 2 clarifies the general control flow of the server.



**Figure 2: General Server Flowchart**

**Controller**

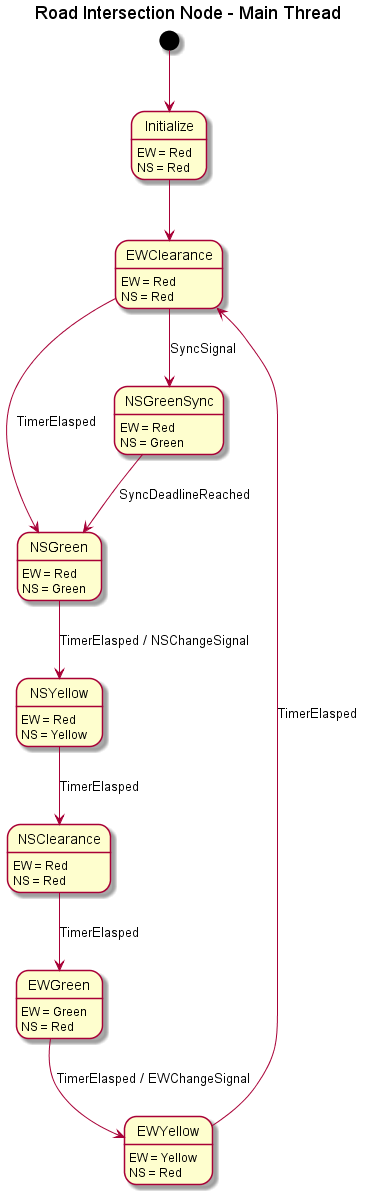
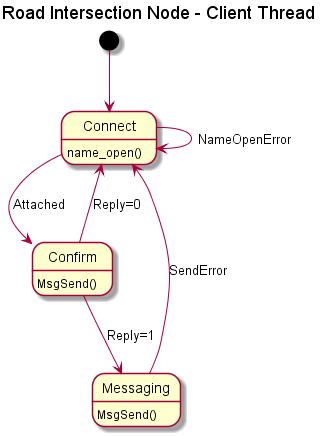
The controller is a separate node used to issue override commands directly to the server. At the moment it is used to issue commands to switch from sensor to timed mode and vice versa. This is done by reading the standard input using “scanf()”. The relevant signal is then transmitted to the server for handling.

This element was implemented as a separate node for two major reasons. First, to free up cpu load and time-slices on the server; ensuring minimal response time. Second, the controller can be utilised for other functions such as modifying configuration. Therefore, we decided it was more “future-proof” to separate the server and controller into separate nodes.

**Road Intersection Nodes**

The road intersection nodes are designed to primarily move through traffic phases in either cyclical intervals (*timed mode*) or driven by events like vehicle detection or pedestrian button being pressed (*sensor mode*) to handle events specified by the use case diagrams in **{IDR fig. 4}**. The nodes periodically update their status to the server and in return ask for any available instruction/command. The server can issue an instruction for the nodes to switch to another mode using a single command code (since there are only two modes so the switch is binary). Synchronization between individual nodes are also necessary whenever two nodes become out of phase from disconnects, late startups or unsynchronized sensor events.

By default, the system runs on timed mode at fixed, hard-coded intervals for each phase and overall similar cycle times. Each phase is represented as a state in a state machine structure. The state diagram in fig. 3 illustrates the flow between states. Note; the extra NSGreenSync state is used to stall the state machine of one node while waiting for another node to reach the same barrier. This is the basis for phase synchronization between the intersection nodes.



**Figure 3: Road Intersection State Diagrams**

When in sensor mode, the node waits for an interrupt trigger on the I/O ports in a separate thread from the main state machine. Sensors are mapped to key values 1-8 on the BeagleBone node’s keypad, whose functions are laid out in fig. 4.

**Figure 4: Road Node Keypad Mappings**

|  |  |  |  |
| --- | --- | --- | --- |
| 1:  North car sensor | 2:  South car sensor | 3:  East/West car sensor | 4:  West/East car sensor |
| 5:  North pedestrian | 6:  South pedestrian | 7:  East pedestrian | 8:  West pedestrian |

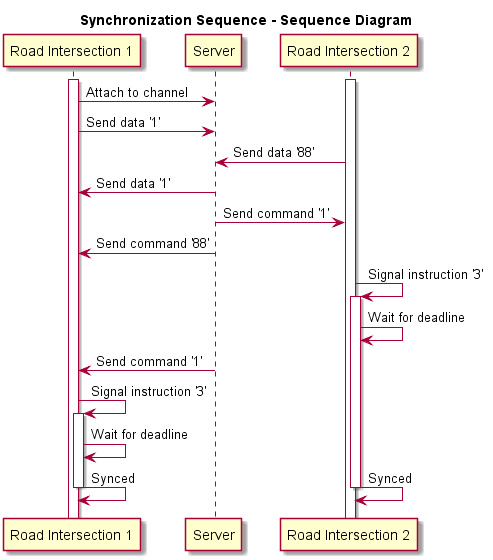
Synchronization process is briefly described in the sequence diagram in fig. 5. The controller specifies the need for synchronization with command code “1” to the nodes, when is then mapped to another set of signals (**Appendix B**) for use in the main state thread. This signal accomplishes 2 things in the main loop: reduce the timing duration of the East-West green phase by half and halt the traffic light at North-South green phase by moving to the NSGreenSync state. Synchronization requires that the clock of the server and the road intersection nodes are matched, which was difficult to handle between different boards. By making the server operate on BeagleBone Devices (same as road nodes), the real-time clock is easily synced by taking the monotonic clock value difference between them as a constant offset. This is tested to be stable overtime.

During a train event, the intersection nodes follow the sequence:

* Receives a ‘12’ from server, set the internal signal to ‘5’
* If the state machine passes to N-S Green first, reduce the light duration by half to hurry to the E-W Green. If the E-W Green is first passed, then wait at this state.
* At E-W Green the state is stalled until client receives a ‘13’ from server to indicate the train passing and boom gate closing, which maps to internal signal ‘6’
* The state proceeds to N-S Green again, at which it waits until client receives a ‘14’ from server to indicate the train has passed completely and the gate is opened. This event is mapped to signal ‘7’.
* Proceeds as normal

Overall, this node requires 5 threads:

1. Main thread: cycle the state machine
2. Client thread: handle the message passing with server, map the given commands to appropriate internal signals (protected by mutex)
3. Input thread: block waiting on interrupt from the keypad, map the pressed key values to appropriate message data to send to server (**Appendix B**)
4. LCD thread A: write to the first line of the LCD every 200ms
5. LCD thread B: write to the second line of the LCD every 200ms

****

**Figure 5: Road Intersection State Diagrams**

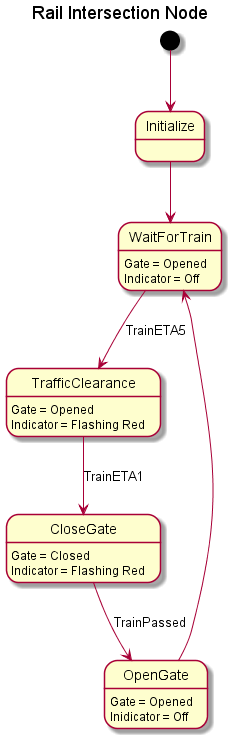
**Rail Node**

The train node ulitilises 1 BeagleBone board, which inherits the message data structs regulated by the server node. Also, this node has its own unique data struct that can be passed around the threads running on the node, which is shown in appendix A. The rail node has the following threads:

* Client thread: use to establish and communicate with the server
* Hardware interface threads:
* LCD thread: display the current state and corresponding guiding message
* LED thread: represent for the signal light, strobing light to indicate the events of train coming and leaving
* Keypad thread: receive the signal from outside factor (train)
* Main thread: run the state machine

Unlike the intersection nodes, the train node has higher priority, which can not be controlled by the server nor the operator. The train node sends message to the server only when an event happens. The event can be when the train is coming, crossing and has passed. The event is controlled via hardware interfaces, where button “15” denotes for incoming train event and button “16” denotes for train has passed event. Then, depending on the event sent to server, the server has corresponding commands sent back to the intersection nodes client.

Basically, the E-W traffic lights at the two intersections turn red for the road crossing the railway. The traffic lights returns to normal operation only when the train has passed. The two events are separate (not a timed sequence) since the train can stop at the middle of the boom gates (delays can happen). Also, corresponding message is displayed on the LCD to guide the drivers. The state machine is designed based on the model illustrated in the initial report:



**Figure 6: Rail Intersection Sequence Diagram**

In fig. 6, after the initialization, the system stays in the WaitForTrain loop and wait until it receives a signal (a train is coming, denoted by pressing button “15”). Then, the system transition to TrafficClearance state, where it sends a message of incoming train to the server and delays for a while (10 seconds for demonstration) before moving to CloseGate state. In this state, then system only move to the next state if the train has passed (pressing button “16”). Train passing event can not be simplified by a delay of some seconds since the movement of the train can not be predicted (it can be delayed and stopped in the middle of boom gate in rush hours). After the train has passed, the system moves to OpenGate state, which opens the boom gates and goes back to WaitForTrain state. At every transition of state, the node updates its status to server via message sending.

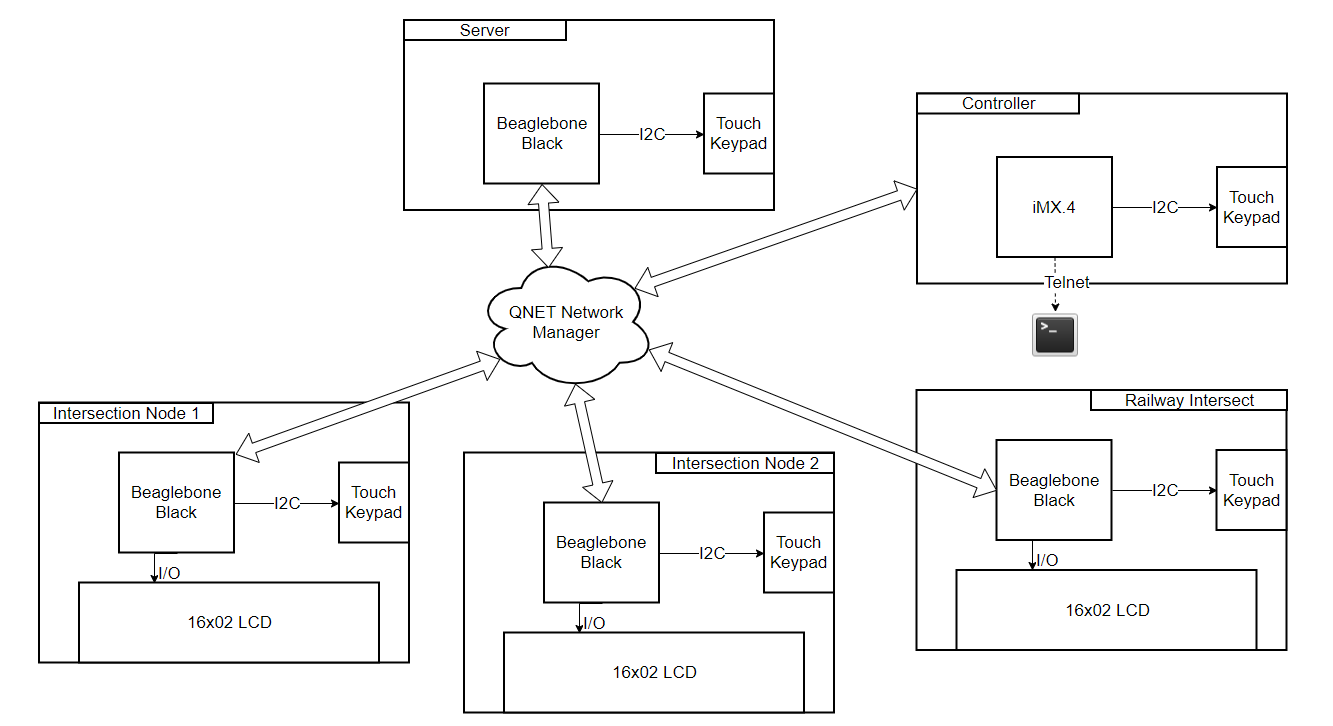
**Hardware**

Modified versions of the code given by Dr Ippolito is used for hardware interfaces (the LCD, LED and keypad on the BeagleBone board). However; for convenient management, the codes are modified into functions, stored in header files (LED\_init.h, LCD\_init.h and KEYPAD\_init.h) so that it can be used in a similar manner between the nodes. In order to make the hardware work, some hardware (port) configurations have to be made. Only then, the I/O data of the ports on the BeagleBone board can be read and written. The hardware interfaces functions are used for train node and intersections nodes.

For the 4 LEDs, 4 bits are needed to be written to GPIO1\_21, GPIO1\_22, GPIO1\_23 and GPIO1\_24 to make the LED lights up (active high). Also, to make the LED strobe, the corresponding bit only needs to be flipped 1 to 0 and 0 to 1.

For the XC4602 Jaycar Keypad, the communication between the module and the board is Serial Protocol Interface [1], which shifts 16 bits at every clock edges (generated by the BeagleBone board) if the module senses the interaction with human hand (touch sensor). The 16 bits correspond to the 16 buttons on the module, and, based on the shifted bits, the data can be decoded into the corresponding button.

The LCD communicates with the BeagleBone board via I2C communication protocol [2]. A string can be written to the LCD using I2cWrite\_() function (requires <hw/i2c.h>). Then, depending on the node, 4 threads or 2 threads can be utilised to write important data onto the LCD at 4 or 2 different areas parallelly. An overview is shown in fig. 7.



**Figure 7: Hardware Configuration**

# Fault Tolerance

Our system has fault tolerance to cover a large range of situations. These fault tolerance mechanisms can be broken down into the following:

* **Road Disconnect:** If a road node disconnects from the system, the adjacent road node(s) will be forced into timed mode and continue to cycle through states until a second road node has been connected. If the power is still on, the disconnected road node will also go into timed mode and continue to run.
* **Rail Disconnect:** In the case the rail node disconnects from the server, it will continue to operate boom gates ensuring no collisions. During this period it will simultaneously and periodically attempt to reconnect to the server until successful. During this period, road nodes will be forced into timed mode to ensure E-W traffic is not backed up while boom gates are unknowingly closed
* **Reconnect:** If a node has disconnected and then attempts to reconnect, it will trigger a holding period. During this period, the other nodes have time to transition to a certain state and are held in this state until it has expired. This ensures nodes are synchronized upon re-connection and also does not cause any major disruption to traffic flow.
* **Late Connect:** If a node does not connect during the initial cooling period it will trigger a similar holding period to the “Reconnect” sequence with the addition of initialisation. This ensures synchronisation and added tolerance for unexpected boot operations.
* **Server Disconnect:** If the server disconnects, nodes will be forced into timed mode sequences while repeatedly attempting to reconnect to the server at set intervals. Once reconnected, they should be roughly synchronised due to the nature of the code (i.e. they were synchronised before disconnect and perform the same operations) and will be exactly synchronised upon the next event. The server can then continue as normal as soon as nodes are reconnected.
* **Acknowledge:** The server sends acknowledge signals after each poll or sensor sent. This allows nodes to check connectivity if an acknowledge is not received and force fault sequences if necessary.
* **Unknown Error:** Each node has set error codes for unhandled signals and operations. Nodes are programmed to handle unknown signals received and generated, as well as print debug information if necessary.

# Development Process

At the beginning, the project was separated into 3 parts corresponding to 3 team members. The 3 parts are:

* Server and client communication (connection establish)
* Intersection node (state machine)
* Railway node and hardware (state machine and hardware interface)

For connection establishment, a simple model of server and client was built and synchronisation primitives were attempted to develop a method of synchronisation. The first attempt was to use iMX as server since its CPU has more core and higher frequency compared to BeagleBone’s. Then, the clock count difference between the server and client was calculated then a few seconds was added to the clock difference to set deadlines for the clients to execute features. However, the CPU’s speed difference was too significant, hence, the clock difference was too high and could overflow. As a result, the server was run on a BeagleBone instead. Once operations were shuffled to the appropriate node, the development process continued to the next iteration.

For the intersection node, the state diagram needed to be framed using the model from the initial report. Then, the system was tested using keyboard input and the number of state and state transitions were ensured to meet the basic requirements of the two operational modes, sensor mode and timed mode. The state machine for the timed mode came up first since the idea of fixed timing was more obvious compared with other mode. The sensor mode behaviour was designed based on assumptions and practical observation, and this required the agreement between the group’s members. Particularly, a number of thought experiments were designed to reach consensus on what action should be taken in a number of scenarios.

The railway node was the most simple node in the overall system and was therefore added last. The railway was initially assumed to have high priority. That is, it could run on its own without the commands from server and it sent event messages to server so that the other node had corresponding reactions. Also, the size of state machine of the boom gate was relatively small compared to that of intersection node from the initial design. The node simply reacted to the train’s movement and sent messages to server. However, synchronising the signals from this node was a challenge and was only attempted once full functionality was achieved in all other network nodes.

The testing phase was done without hardware interaction first and utilising standard input. The hardware interface code for the LED, LCD and keypad was given by the lecturer. The hardware behaviour was examined separately from the project to learn how the hardware work. After the examination, the code was modified (simplified) into functions and header files so that every node could use the provided hardware by including the header files. The hardware interfaces ran on different threads, hence, data between the threads must be protected and shared.

Then, once the hardware and software worked, the client code was merged into the node’s code, where the node could run the code and send its status to the server. The server synchronized the nodes by setting the deadlines for the nodes and replying the appropriate messages when the client updated its status to the server. The status code can be viewed in appendix B.

Overall, it was an extremely iterative process going through a number of crucial feature additions. The first iteration began with a simple clock synchronisation and more nodes and features were progressively added until desired full functionality was achieved, testing a set number of scenarios at each milestone. This design allowed us to account for unexpected errors and debugging by ensuring we could always submit the last working revision in a disaster scenario.

# Future Direction

There are several additional features that we planned to implement if the scope of this project was larger. These features are:

* **Right Turn Signals:** While our system does account for vehicles turning right when safe, it does not currently display specific lights at each intersection signalling a right turn. It would be more realistic to add specific signals and increase the intersection states to include “right turn only” states.
* **Extra Nodes:** Our system is programmed to expand to any number of nodes in a certain layout. Adding extra nodes to simulate a larger network of intersections would be a great feature to add and test. It would also be good to adjust the program to allow extra nodes in any layout for more portability.
* **Reinforcement Learning:** Our system could utilise reinforcement learning to change states of each intersection autonomously. This would be done by setting the reward to maximise traffic flow. This is a current area of traffic research and would require a lot of experimentation adjusting the structure of inputs and reward functions.
* **Automatic Mode Change:** It would be more convenient and useful to change traffic mode based on traffic flow and other measurements rather than requiring a manual override. This would be implemented by designing an equation based off sensor inputs however; it would also require more simulated sensor input to be added
* **Test Suites:** Two test suites could be added. First, to test the software before deployment when changes have been made. This allows greater certainty that bugs have not been introduced with the latest code revision. Secondly, a test suite that runs on launch and tests components. Any faulty results could be used to adjust the system appropriately e.g. boom gate broken so flash warning lights

# Reflections

Our group developed a number of skills over the duration of this project. Firstly, each member had to develop a number of soft skills to coordinate collaboration between 3 group members, especially since our project went through several iterations. Each member of our group had experience in vastly different areas of engineering and combining these efforts to create our final product involved negotiation, communication and cooperation.

Our combined understanding of developing programs for real time systems has also advanced significantly over the course of this project. While we each developed separate components of the system, a detailed understanding of the entire design was required to integrate all components. This project also forced us to extensively practice general C programming, coding practices, code management, embedded system development, design, networking and a number of other skills.

Overall, our team thought the project was a great way to develop real time system development skills however; testing was difficult due to the limited number of permanently deployed beaglebones in the lab. Individual reflections are discussed below:

**Nathan**

Most of my time was spent developing the server. In doing so, I had to develop a rock-solid understanding of QNX message passing. Our system revolves heavily around the server executing commands to the client nodes. As such, I had to perform plenty of testing on the server to ensure an incorrect message was never transmitted. In case this failed, I had to ensure errors were handled appropriately. I also had to synchronise events which forced me to learn a lot about system clocks and common methods of performing this task.

C is not the language I commonly program in either, so I had to develop my skills and utilise stack overflow to translate common functions I knew in other languages. Overall, I was completely new to real time systems design so this project taught me a lot of the necessary macro and micro skills.

**Phuc**

My portion of this project focused heavily on the road intersection nodes. The operation of the node require multiple threads and I had to handle a lot of shared memory problems. Resolving the issues gave me more understanding of how to manage mutual exclusions, when and when not to use condition variables and how to use construct like signal variable to pass message quickly between internal threads. The process of synchronization was also particularly tricky, as the state machine needs to be able to run continuously without much interruption while still be constantly on the look out for signals from the client thread to synchronize probably. Deadlocks are particularly common mistakes during development, from simple mistake like forgetting to unlock a mutex to misused condition variables. Resource utilization time is also important, as improperly designed threads may hog a shared memory a lot more than other threads which means many workarounds to reduce the time spent inside critical region.

Another issue that kept springing up is code management. The initial structure of code was important to easily adapt overtime, and I was struggling with adapting as the message passing between threads start getting bigger and more complicated. By simplifying the server’s instructions to a series of coded signals, the problem has been much simpler to the end of the project phase. Overall, the experience has taught me a lot on concurrent programming, client-server messaging and managing my codebase.

**Hieu**

The project’s requirement is very practical and new for a non-local person like me. I had to do some research and observation in order to resolve the possible situations that can happen. In term of technical, I learnt and understand very well about QNX message passing after the project. I understand how a server runs and synchronized the connected client by having the same deadlines. This project shows the importance and strength of interprocess communication. This course is heavily software based, however, after the project, I can see how powerful and useful software can be when it is correctly utilised with hardware.

# Contributions

*Each member contributed an equal 33% to this project*

**Nathan**

* Code for server node and all server operations (e.g. server-side fault tolerance)
* Code for control panel node
* Defined message struct standards
* Report contributions:
  + Formatting
  + Problem statement
  + Server solution
  + Fault tolerance
  + Future directions

**Phuc**

* Road intersection node
* Adding LCD to road intersection nodes
* Test plan
* Report Contributions:
  + Road solution
  + State and sequence diagrams

**Hieu**

* Boom gate node code
* Hardware interface research and simplification
* Report contribution:
  + Rail node
  + Hardware interface
  + Development process

# Appendices

**// ----------server data struct----------**

**typedef struct**

**{**

**struct \_pulse hdr; // Our real data comes after this header**

**int ClientID; // our data (unique id from client)**

**int data; // our data**

**} my\_data;**

**typedef struct**

**{**

**struct \_pulse hdr; // Our real data comes after this header**

**int data;**

**double clock;**

**double event;**

**} my\_reply;**

**//used to store next event and other details for server**

**typedef struct{**

**int signal;**

**double eventTime;**

**int client;**

**int modeSwitch;**

**char currMode; // 's' or 't'**

**} event;**

**//----------Client data struct----------**

**typedef struct {**

**sendData msg;**

**replyData reply;**

**double clkdiff; // clock difference between server and client**

**int data\_ready;**

**pthread\_mutex\_t mutex;**

**pthread\_cond\_t cv;**

**int finish;**

**} clientThreadData;**

**//----------Train node data struct----------**

**enum Boomstates {**

**wait4train,**

**trafficClearance,**

**closeGate,**

**openGate**

**}; // states of the boom gates**

**typedef struct{**

**enum Boomstates state;**

**char input;**

**} modState; // include boom gate state and input to the state machine**

**typedef struct**

**{**

**int fd;**

**uint8\_t Address;**

**uint8\_t mode;**

**pthread\_mutex\_t mutex;**

**} LCD\_connect; // used for interacting with the LCD**

**typedef struct{**

**int trainSENSOR;**

**char state[20]; // 2 string buffer used for LCD displaying**

**char mode[20];**

**LCD\_connect td;**

**modState curState;**

**clientThreadData clientData;**

**} TrainNodeDATA; // data type used for boom gate client node**

**Appendix A: Struct Definitions**

**TABLE: Data values sent to server**

|  |  |  |
| --- | --- | --- |
| **Data** | **Action** | **Notes** |
| 88 | Poll | Transmit current stored event (empty if none) |
| 1 | Initialise | Used to trigger initial connect or reconnect sequence |
| 5 | Switch Mode | Manual override switch mode |
| 8 | E-W Signal | Signal from E-W sensor as intersection |
| 9 | N-S Signal | Signal from N-S sensor as intersection |
| 12 | Train Coming | Train approaching, clear E-W |
| 13 | Train Passing | Boom gate down |
| 14 | Train Gone | Boom gate up |
| 0 | Error/Unknown | Each node programmed to handle unknown |

**TABLE: Mapping of server’s command to node’s instruction as internal signals**

|  |  |
| --- | --- |
| **Command** | **Instruction** |
| 1 | Signal 3, wait for deadline, then signal 4 |
| 5 | Signal |
| 8 | Signal 1 |
| 9 | Signal 2 |
| 12 | Signal 5 |
| 13 | Signal 6 |
| 14 | Signal 7 |

**Appendix B: Signals and node’s instructions**

# References

[1]Jaycar.com.au. (2019). *Arduino Compatible 16 Key Touch keypad module | Jaycar Electronics*. [online] Available at: https://www.jaycar.com.au/arduino-compatible-16-key-touch-keypad-module/p/XC4602 [Accessed 17 Oct. 2019].

[2]MIDAS Display, (2019). [online] Available at: https://au.element14.com/midas/mccog22005a6w-fptlwi/lcd-alpha-num-20-x-2-white/dp/2425758?ost=2425758&selectedCategoryId=&categoryNameResp=All&searchView=table&iscrfnonsku=false [Accessed 17 Oct. 2019].