Project Hazzard
A Remotely Controlled Racing Adventure Game Experiment

Chen, Hao
Nelson, Robert
Stahl, Darren
Vidanamadura, Hasith

December 9, 2013

1 Project Description

Project Hazzard is a remote controlled racing game, using Raspberry Pis controlling one (or more) RC cars. Statistics about the cars current state (speed, direction, etc.) will be displayed on a GUI rendered using the Java SWING framework on a second Raspberry Pi. A third Raspberry Pi will be used to collect controller input from the players and forward that on to the Pis controlling the car's movements. Finally, a fourth Raspberry Pi will be attached through a GertBoard to the finish gate to detect when a car finishes a lap.

The separation of the controller from the controlled cars means that there can be additional influences added into the control path to add difficulty to the game. Any number of modifiers could be applied to challenge the players in their attempts to win the race.

2 System Architecture

2.1 Deployment

Project Hazzard is composed of 5 main subsystems. The input subsystem is responsible for processing input from Xbox controllers and passing the parsed values to the second subsystem PiNet. PiNet is responsible for allowing the other subsystems to communicate in a transparent manner. The game engine is responsible for processing all game logic and sending commands to the car subsystem. The car subsystem includes the scripts required to convert PiNet commands into car control signals, as well as the circuitry required to control the physical cars. The fourth subsystem is the finish gate. The finish gate detects when a car passes through it, and sends signals through PiNet to the game engine when it detects a car. The final subsystem, the GUI, is updated through PiNet by the game engine, and displays the current game state. Figure 1 shows the full deployment diagram. Each software subsystem is circled in a dashed line, and each hardware component is circled in a dot-dash line. All of the software components can be deployed to seperate Raspberry Pis if necessary, but they can also be deployed to the same Pi if that is more convenient.

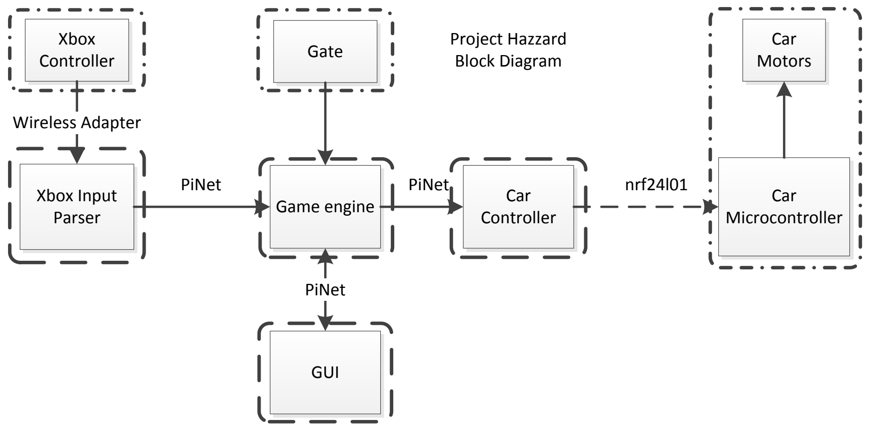

Figure 1: Deployment diagram for Project Hazzard

2.2 Communications

Each of the four deployment components needs to communicate with each other. This communication is facilitated through PiNet, the networking library the team wrote for the trial project. Its flexibility allowed it to be easily ported into this application.

There are many different types of messages that are sent between the various components. Table 1 lists all of the various messages that each component sends and receives.

Table 1: Messages sent and received by component

Component	Messages Sent	Messages Received
Input Parser	ControllerEvent	XboxEvent
Game Engine	RaceInfo CarEvent LapComplete PlayerFinish RaceFinish PlayerInfo	ControllerEvent LapFinishedEvent PingEvent
Car Controller		CarEvent
Gate	LapFinishedEvent	
GUI		RaceInfo PlayerInfo

3 Class Diagrams

Since each subsystem of the project was designed to be an independent module that performs in general, one function, a class diagram does not show much for most. The class diagram for PiNet, which is used by most modules in the program serves to show how a class interacts with the PiNet classes. It is found in figure 2

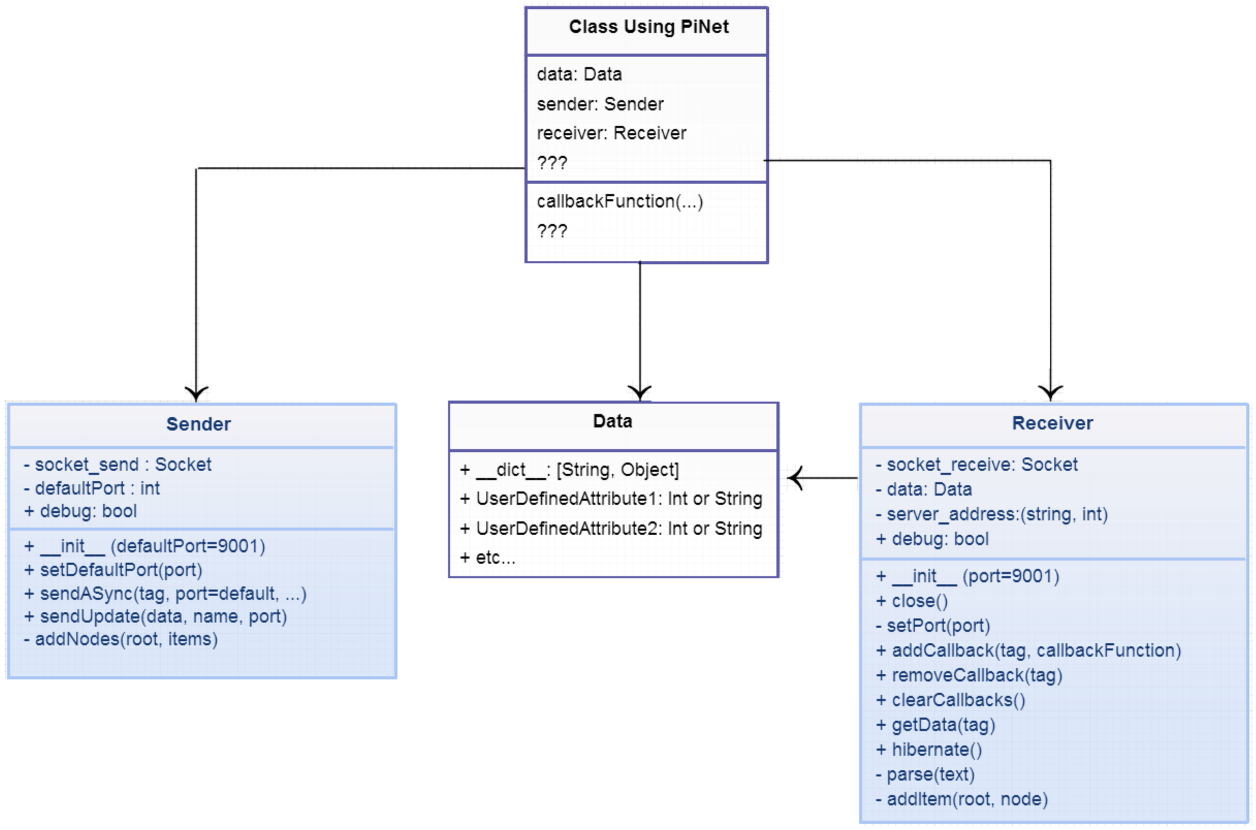

Figure 2: UML class diagram for PiNet, as used by any arbitrary class

4 Sequence Diagrams

A sequence diagram of the entire system would be far too large to be understandable, so smaller intercomponent sequence diagrams will be shown to detail the interaction between the components. Each sequence diagram will show one style of transaction that happens between the different components.

4.1 Input

The input subsystem encompases the Xbox controller, and the input controller. The sequence diagram for the behaviour of these systems can be seen in Figure 3

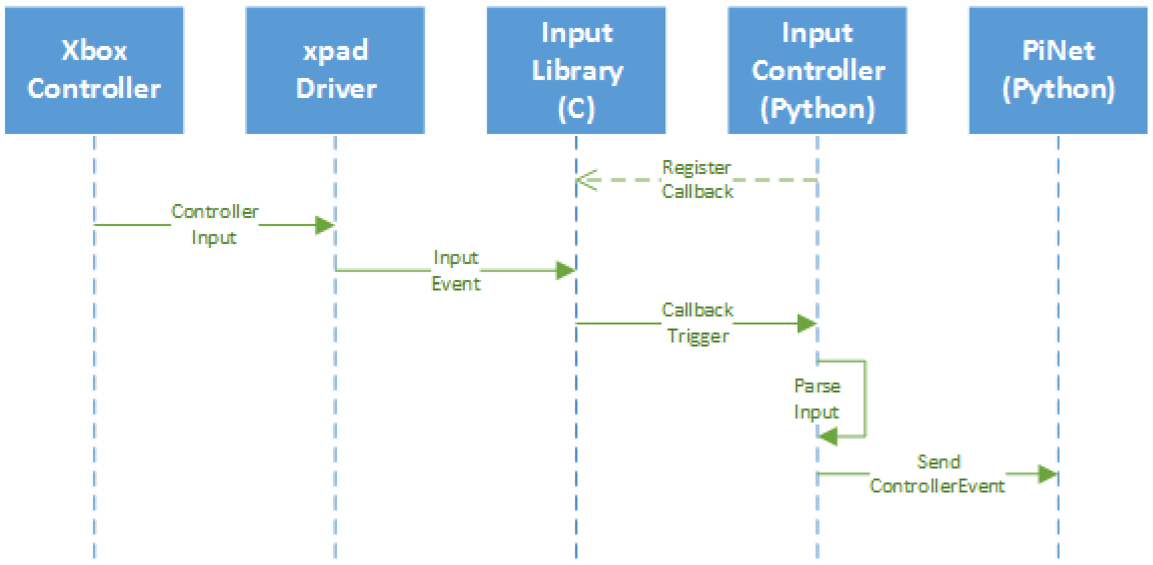

Figure 3: Sequence diagram for input subsystem

Input on the xbox controller is processed by the xpad driver, and converted into input events. These input events are read by the input library, and passed into the input controller. The input controller processes the events, performs rate limiting, and eliminates any unwanted events. Once only the desired events remain, the input events are converted into ControllerEvents and sent out over PiNet to the Game Engine.

4.2 PiNet

PiNet includes two methods of sending data to and from clients. The first is asyncronous data, the sequence diagram for which can be found in Figure 4 Note: While both the following sequence diagrams show a single receiver, there can be any number of receivers.

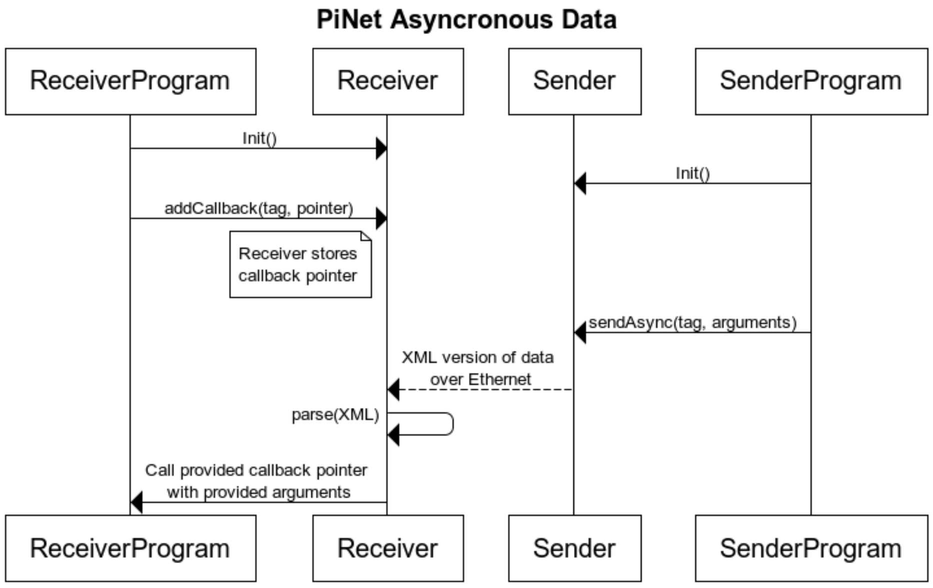
PiNet Asyncronous Data
ReceiverProgram
Receiver
Sender
SenderProgram
Init()
Init()
addCallback(tag, pointer)
Receiver stores callback pointer
sendAsync(tag, arguments)
XML version of data over Ethernet
parse(XML)
Call provided callback pointer with provided arguments
ReceiverProgram
Receiver
Sender
SenderProgram

Figure 4: Sequence diagram for PiNet's async interface

The other method PiNet provides is syncronous or update data. The sequence diagram for updates can be found in Figure 5.

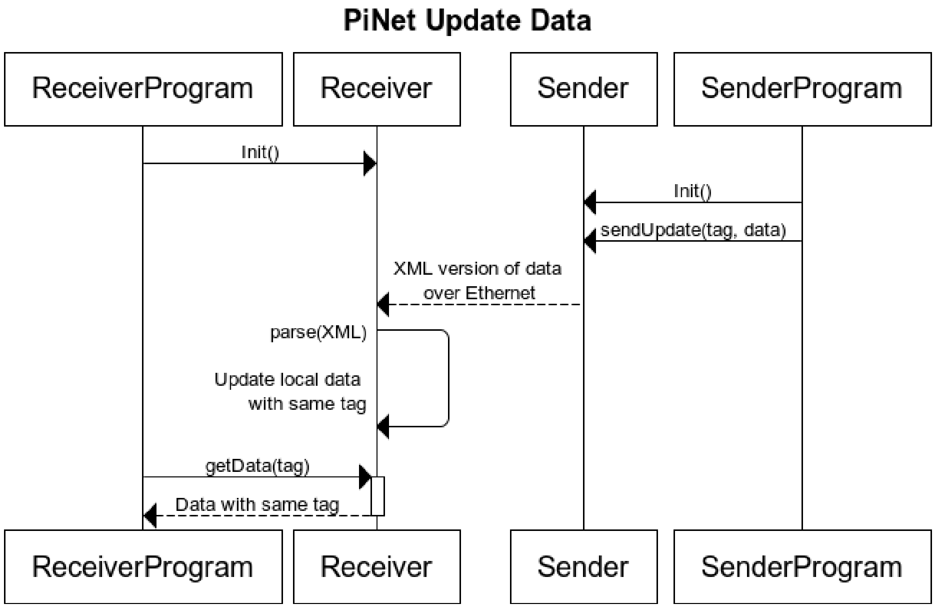

Figure 5: Sequence diagram for PiNet's update interface

4.3 Game Engine

The game engine has many different sequences that it procedes through, but for simplicity, only the most important one, the sequence that takes place during the race will be shown. The other sequences in the game engine are simply variations on Figure 6, with some additional loops or checks.

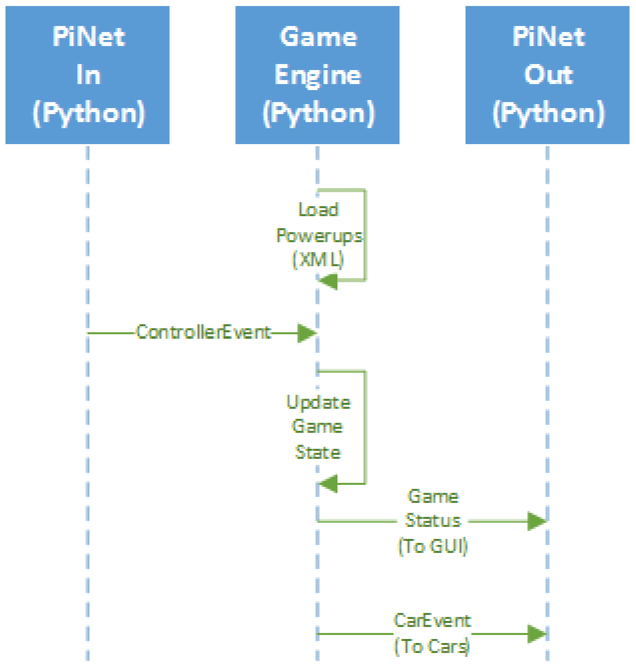

Figure 6: Sequence diagram for the Game Engine during a race

4.4 RF Link

The cars are connected to the main Raspberry Pi based network via a 2.4GHz-band radio network based on nRF24L01P modules. The modules provide auto-acknowledgement, and are reliable enough to perform at 1Mbps. The relevant documentation is given in the cuLearn book, and the sequence diagram (Figure 7) showing the sequence of events when a radio packet is sent and processed by the car controller.

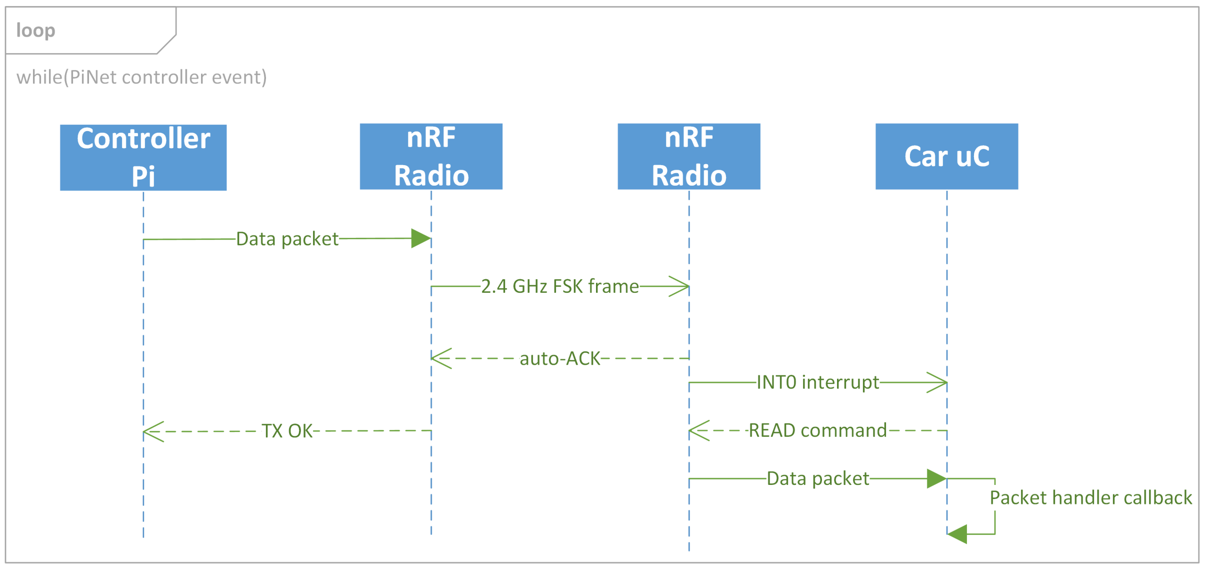

Figure 7: Sequence diagram for Raspberry Pi to Car Microcontroller communication

4.5 Car Microcontroller

The cars are controlled by custom firmware running on AVR microcontrollers. These decode the RF link packets into commands to be forwarded to the actual car hardware. The microcontroller additionally supports the generation of the IR beacon for identifying cars. The sequence diagram in Figure 8 identifies the decoding and beacon processes. For further information, please refer to the cuLearn book.

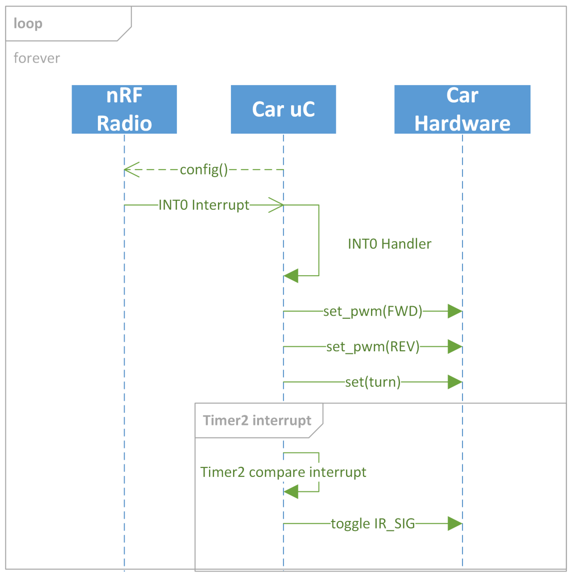

Figure 8: Sequence diagram for Car Microcontroller activity

5 Testing

5.1 Testing the Input Subsystem

The input subsystem was difficult to test effectively due to its inherent dependency on the Xbox controller hardware. This made it impossible to write automated tests to check the input library code. Eventually, it was decided that since the code was relatively simple, a functional verification would be the only test. A test application was written to measure the rate that input events were received. Table 2 shows the data collected during this test.

Table 2: Number of input events observed during various input conditions

# of Controllers	No Input (Evts/s)	Normal Input (Evts/s)	Heavy Input (Evts/s)
1	0	120	750
2	0	250	1600

These test results led to the conclusion that the number of input events would exceed the capacity of PiNet if all of them were transmitted. The only recourse was to reduce the number of events by not sending any unnecessary events. The logical type of event to trim was the sensitivity of the analog values. Each analog stick had an X and Y range of -32767 to +32767. The cars only supported on/off turning, so these ranges were reduced to three values: -32767, 0, and +32767. All other analog events were ignored. This drastically reduced the number of events that needed to be sent to the game engine.

5.2 Testing the RF link

The major concern for the radios was reliability. Given that the modules themselves do not have external low-noise amplifiers or power amplifiers, the range of the modules was tested. To test this, the radios were plugged into the car microcontrollers, and disconnected from all outputs except serial for debugging. The idea was to monitor a continuous test (contained in ping_test.c) that attempted to contact a target, and vary the distance and collect data.

5.2.1 Effective Distance

The **effective distance** is defined as the distance data can be transferred by a module transmitting at 0 dBm (full power) with zero packets lost. The test application from the previous section was reused, and the module tested both in line of sight **(LOS)** and through-walls **(TW)** configurations.

Line of sight: The maximum testable line of sight distance was about 5m, and zero packet loss was observed.

Through-walls: The maximum distance testable was about 10m with two floors and walls between the source and the sink. Zero packet loss was observed even in that configuration. At 12m, noticeable packet loss (66%) was observed.

Based on the above testing, the module meets the requirements. Since the project requires the cars to be visible to the players, the LOS configuration is more applicable, and the range in that configuration is sufficient for the purposes of this project. Further testing data including effects of the power rail noise is included in the cuLearn book.

5.3 Testing PiNet

PiNet was one of the few automated tests that could be done on the system, since each part relied so heavily on user input, or on physical output. PiNet/PiNetTest.py contains all of the testing code for PiNet. It starts by testing the small integral functions to make the system work, and slowly ramps the testing up to test more and more complicated parts. It eventually tests the system as a whole with fake inputs from a socket. It finishes by testing the system with the coresponding networking library partner.

5.4 Testing the gate

The gate is very reliant on its external surroundings, as it is based on light. Since it was built in an isolated location to get confirm its operation, even something as simple as running it in the lab was testing it. It was found that the flourescent lights triggered the sensor at 10KHz, which masked any cars driving though. It was required then that we block all light except for IR light. As a result of this test, an IR filter (magnetic disk from a floppy drive) was placed on the sensor, this prevented accidental trigger by the lights.

6 Discussion

6.1 MVC

The MVC model used in Project Hazzard is decentralized. It contains elements from multiple physical Raspberry Pis, as well as an abstract network packet. The controller is the Game Engine. The game engine is responsible for agregating the data from many sources, and compiling this data into a RaceInfo packet (the model), which is shared with the GUI (the view) over PiNet. This slightly deviates from the MVC model, in that a PiNet instance on the GUI does some networking and parsing of data, but it conforms to the spirit of MVC.

6.2 XML

XML is used in a number of places in the project. It is used to create a unified format for data in PiNet, and is used for configurations in the game engine.

6.2.1 XML in PiNet

In PiNet, XML is used to create a known format for sending the arguments to a callback, and a unified, extensible format for packaging classes. The callback uses a single element as the callback, and any number of attributes as the arguments. This is to limit the complexity of arguments, as they do not need to be extensible. The update data passing method uses all elements to store the data. This is because the class may contain multiple different types of data, which can be represented as a "type" attribute, or, may contain sub elements, which can be respresented as childen elements. This increased the load on the network, as there is a fairly large text overhead for using elements over attributes, but we felt that the ability to extend the update framework was worth the network overhead. A schema is not used, as the contents of the XML are unknown until execution time.

6.2.2 XML for Powerups

In the game engine, the powerups are stored as an XML configuration file (game_engine/powerups.xml). This file follows a schema (game_engine/powerups.xsd) which allows the game engine to ensure that each powerup follows the proper definition. GenerateDS, a tool which generates a python XML parser from an XML schema was used to create a class based parser for the powerups. Unfortunately, since the generated parser uses the SAX model to perform the parsing, the powerups file cannot be validated against the schema. Therefore, since the schema validation was an important step, a second XML library *lxml* was used to ensure that the powerups were in a valid format.

The powerups format is very extensible, most data within a powerup is stored as elements. This provides the ability to add new elements in the future if for example more effects needed to be added. The name of the individual powerups however will always be required, and so that value is stored in an attribute on the top level *powerup* element.

7 Technical Recommendations

While the individual components performed well in isolation, several issues cropped up during the presentation phase. After careful analysis, the following technical recommendations were agreed upon by the team.

1. The choice to use inexpensive RC cars as the hardware platform for the project was a mistake, as the cars required extensive retrofitting (to the level that none of the original electronics were retained), and performed less than expected. A revisiting of this project would replace the current RC cars with more dependable performance and reliability characteristics.

2. Time was wasted on attempting to port PiNet to Jython, so as to ease the interfacing with the (written in Java). Since Jython uses the underlying Java classes it was thought that this would be a simple way to interface the GUI with PiNet. Once the code had been converted to Jython, it was found that Jython did not support Multicasting, as the developers felt that it was not an important feature. This took a while to find out, and resulted in a large time sink when we needed it the least. Once Jython was working by mixing Java and Python networking types, it still failed to integrate with the GUI. At this point, we had no choice but to cut our losses and rewrite PiNet in Java.

3. The order for the car controller PCB was placed in late October, as soon as the requirements for it were finalized. The delivery delays meant that the PCBs did not arrive until two weeks before the final presentation, unfortunately this caused a lot of the milestones to be missed, because they were scheduled based on the PCBs arriving when expected. The recommendation is therefore to overestimate how long it will take for PCBs to arrive, and plan accordingly. That way, if the boards arrive earlier than expected, there is extra time, but plans are in place if the boards are delayed.

8 README

8.1 Starting the System

The system has several components, in order for the GUI to function properly it must be started before the game engine. In addition, the RC cars must be started before the car controller. All other components can be started in any order and the system will function properly.

8.1.1 Starting the GUI

To start the GUI, the java files must be compiled using the commands:

```
> cd GUI/src
> env CLASSPATH=com/hazzard/gui/dom4j-1.6.1.jar:com/hazzard/gui/jaxen-1.1.6.jar:.
        javac com/hazzard/gui/*.java
```

Then to run the GUI:

```
> env CLASSPATH=com/hazzard/gui/dom4j-1.6.1.jar:com/hazzard/gui/jaxen-1.1.6.jar:.
        java com.hazzard.gui.Start
```

8.1.2 Starting the Car Controller

To run the car controller, the nrf library must be built.
Run:

```
> make -C lib/bcm2835/
```

To build the library. Then, the car controller can be started by running:

```
> cd car_controller
> ./car_controller.py
```

One thing to note is that the RC cars must be on before the car controller is started.

8.1.3 Starting the Game Engine

To run the game engine, simply run:

```
> ./game_engine.py
```

8.1.4 Starting the Input Controller

To run the input controller, the xbox library must be built.
Run:

```
> make -C xbox
```

and then the input controller can be started by executing

```
> cd xbox
> ./input_controller.py
```

8.1.5 Starting the Gate

To run the gate, the shared library must be compiled

```
> cd gate
> make
```

then, the Python wrapper must be started, it must be run as root to gain access to memory for GPIO actions; execute

```
> sudo python wrapper.py
```

8.2 Repository Layout

```
/
├── GUI
│   ├── build
│   ├── dist
│   ├── nbproject
│   └── src
├── PiNet
│   └── examples
├── car_controller
├── doc
├── game_engine
├── gate
├── lib
│   ├── avr
│   └── bcm2835
├── minutes
├── weekly_status
└── xbox
```

- **GUI** contains the project for the Graphical User Interface.
  - The source is located in **GUI/src/com/hazzard/gui**
  - The Netbeans project is located in **GUI/nbproject**.
- PiNet sources are all located in **PiNet**.
  - In **PiNet/examples** there are two examples, one of how to use a Sender and one of a Receiver.
- **car_controller** contains the Car Controller source.
- **doc** contains the block diagrams of the system, and the presentations given.
- **game_engine** contains the central game engine and associated tests.
- gate code is located in **gate**. It contains a make file, as well as sources.
- **lib** is where the code for the car microcontroller, and the nrf24 radios is located.
- **minutes** contains the meeting minutes for the meeting we have had.
- weekly statuses are contained in **weekly_status**, though there was only a few required.
- **xbox** is where the code for parsing the xbox input resides.