TECHNICAL DOCUMENTATION

Federated Vehicle Testbed Project - Design Documentation and Techical Guide

Written By

Jacob Carroll
Madeline Cochrane
Matthew Rimkevicus
Jed Sharman
Michael Shim
Aslan Chadwick
Ryan Black
Timothy Quill
Simon Daniel

Swinburne University of Technology

Contents

Chapter 1

Introduction

1.1 Purpose of Document

This document serves as a technical report for the Federated Autonomous Vehicle Testbed Project. It contains information about design decisions and provides further information on use of the simulation engine. This report is targeted at operators i.e. people using the system in a research or development capacity. The information here is primarily focused on knowledge needed to expand existing functionality and configure the project to meet the users goals. It is best used in conjunction with the API and source code.

For users wanting information pertaining to end-use of the product, such as menu layouts and controls, please consult the User Manual.

1.2 Project Overview

The aim of the overall project was to design a simulation environment to be used for the development, testing and demonstration of algorithms for federated fleets of vehicles. The engine in its current form serves as a modular and extensible framework upon which users can create custom agents, maps and scenarios. Users can configure agents with countermeasures and gadgets to meet their needs.

When running a scenario, each agent can be controlled by an AI or a player and can communicate with other friendly agents to coordinate actions. Every action and decision is recorded by the inbuilt data and event recorder for which users can create and select custom data streams.

Chapter 2

Menus System

2.1 Overview

This chapter will outline the design of the menu system. Detailing how it was built, and how to extend it.

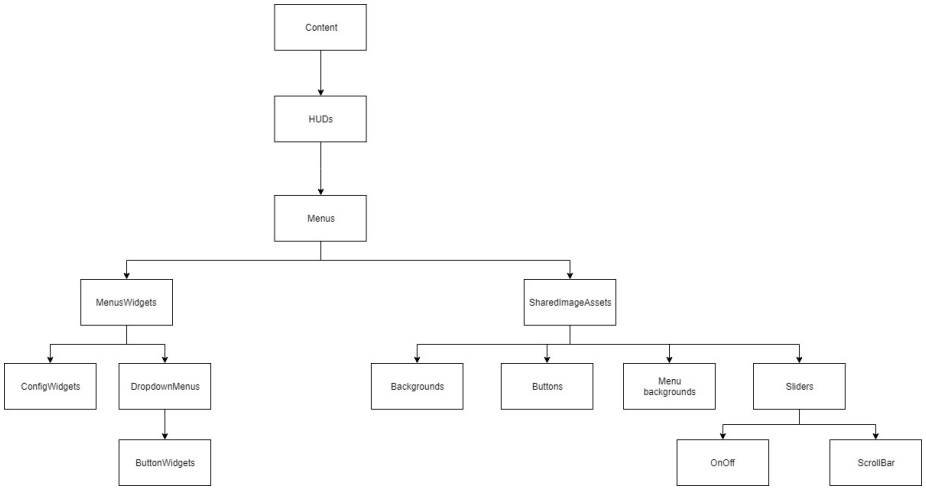

Figure 2.1: Content Structure

2.2 Assets

The menus use common assets to allow easier updatability, this is best exemplified by the button background assets. These assets are used by all buttons and allow them to display when a button is either hovered over or not, these assets are located in SharedImageAssets/Buttons.

To add any assets to the project they should be added to the corresponding directory in SharedImageAssets, if necessary a new sub directory should be created. To include a new asset the developer should go to the correct directory and then use the import tool to import the asset.

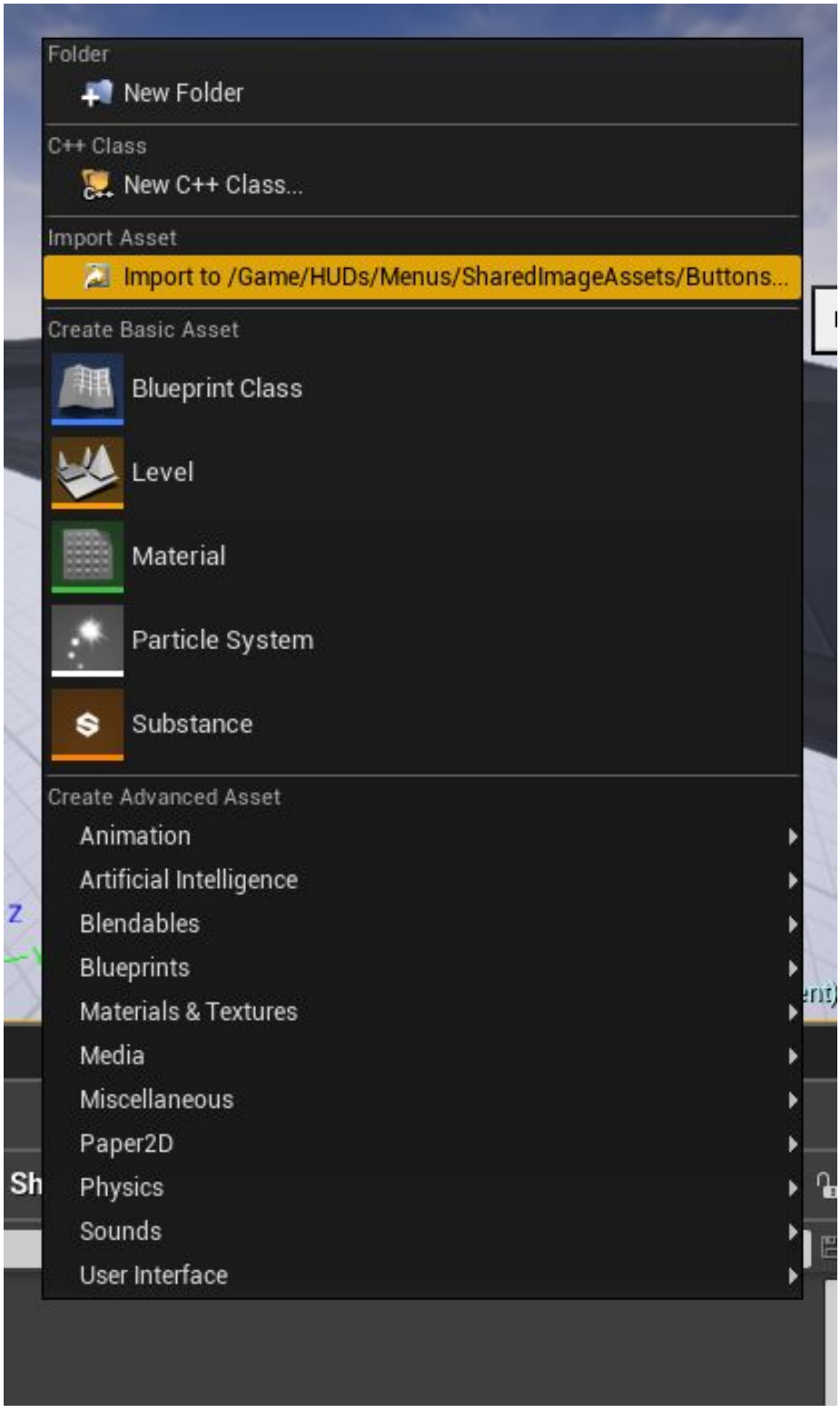

Figure 2.2: Import tool

2.3 Dropdown Menus

Unreal doesn't have native dropdown menus and so to create a dropdown menu in this project is a bit of a work around. Firstly you have to make a button widget, located in the MenuWidgets/DropdownMenus/ButtonWidgets. You then have to make a new dropdown menu widget, this widget should generate button widgets and have a function on it for the button to reference when the button is clicked. The button widget should then be updated to have a reference to the dropdown widget, calling the dropdown widgets on click function when the button is clicked. By doing this the dropdown menu can react to button clicks of dynamically created buttons.

Chapter 3

Configuration

3.1 Overview

This document describes the code behind the Configuration system. How it works, how to use it, and how to extend it.

3.2 How it Works

3.2.1 Class Structure

Configurator

This is the only class other parts of the program should interact with, other than the user interface for obvious reasons. This class houses static functions for saving and loading configurations. When saving, it creates an xml_document object, tells the configuration object to append its details to that document, and then writes the document to file. When loading, it opens the file and creates an xml_document by parsing the file, and then creates the configuration object with the correct class type.

This class also houses a static string storing the filename of the scenario chosen to be run by the user. It also contains a function to start said scenario, however this function relies on the game to already be set to the correct map before being called.

3.2.2 ConfigBase

This class is the abstract base of all configuration classes. (Config classes for short). The Unreal engine does not allow pure virtual functions in UObject derived classes so the functions in this class have been defined but essentially left empty. They utilise an Unreal engine macro to force all derived classes to implement these functions. All config objects are designed to be interacted with

from the blueprints that make up the main menu so almost all functions in this and derived classes have the UFUNCTION macro.

3.2.3 Scenario Config

This class represents the encapsulation of a scenario. It is the highest level config object. Anything that is to be used in a simulation scenario should be directly or indirectly mentioned in the scenario config object. It stores all agent configs used, where to spawn them, what communication config to use, and where to save the data and event recording output files.

3.2.4 Agent Config

This class stores all information necessary to create an agent in the simulation. It saves the name of the agent, the name of its class, the names of the classes of gadgets to attach to the agent, and whether to start the scenario in control of the agent. This class also stores the location and rotation to spawn the next agent at in the simulation. The intended use is for the scenario config to update these values with the information from the spawn location before telling the agent config object to spawn an agent that it encapsulates.

3.2.5 Communication Config

This class, CommConfig as it is called in the code, encapsulates the information to initialise the communication distributor class with when the scenario starts. A scenario config object can only have one of these at a time. Objects of this class store the names of SNRModel derived classes obtained using the reflection system of the Unreal engine. It stores the default propagation model to use and any additional models to use in specific ranges of frequencies.

3.3 How to Use it

3.3.1 Creating a Scenario

All config objects have various getters and setters for all variables.

Scenario Config

Scenarios are technically valid without any agents or spawn points, but it is highly recommended that some are added so that the scenario actually does something in simulation. It is required that a CommConfig is used, a map name is set, and the output locations of the data and event recording systems are set.

Agent Config

The potential values for the agent and gadget class names are found using the reflection system of the Unreal engine. It is not required for an agent to have any gadgets. Nor is it required to provide a name for the agent, however it will make it more difficult to distinguish agents in any output files because the Unreal engine will give them a name based on their agent class.

Communication Config

It is required to provide a default propagation model because the Communication Distributor class defaults to using the VoidSNR class which zeroes the SNR of all messages sent through the system. It is not required to provide any addition propagation models within ranges of frequencies. The names of the available propagation model classes are found using the reflection system as well.

3.3.2 Starting a Scenario

In order for a scenario to be started, one must be set as the scenario to use in the configurator class. There is a setter available for this purpose. Next, one must open the level that the scenario is to be run on. This starts the initialisation sequence in the Unreal engine. It is recommended to call 'StartScenario' at some stage during the initialisation process, most likely in 'PostInitializeComponents' or 'BeginPlay' of the game mode or some other high level class. This will ensure the scenario is fully initialised and ready to go before the simulation begins.

The 'StartScenario' function in the configurator is essentially a wrapper function for calling 'Initialize' on the ScenarioConfig object. This function in turn calls 'Instantiate' on each of the AgentConfig objects for each spawn point, and on the CommConfig object. If at any point something doesn't instantiate correctly, the program will bubble back up to the 'StartScenario' function in the configurator which sends the program back to the main menu.

Instantiate in the Scenario Config

All instantiate functions rely on a context object to get a reference to the current world from. The first thing the scenario config does after checking that the context object can indeed be used as a config object is checking whether the program is already on the map specified in the scenario config, immediately stopping the instantiation process if its not. It tells the CommConfig object to instantiate itself before any agents are spawned so that any communication equipment on them will initialise correctly.

Instantiate in the Communication Config

There's nothing special in this class. It sets the default SNRModel class that the Communication Distributor should use and adds any additional frequency

ranges. It has a small optimisation to only make one object of any SNRModel derived classes to save on memory usage.

Instantiate in the Agent Config

This function spawns the agent into the game world and then adds any gadgets to it. Determining how to add gadgets is currently hard-coded into this function. This function currently does not support adding communications devices to agents or initialising them.

3.3.3 RapidXML

The configuration system relies on the RapidXML library to read and write from file. We have chosen to use this library because it constructs a tree structure of the xml document and allows us to operate on it in memory, doing all of the reading and writing from file for us. There are a handful of classes and functions within the RapidXML library that have been used in this project. There are a few quirks to using them so we've provided a few tips based on what we've learned.

xml_document

This class is the basis for an xml document. It is used for creating other nodes so when passing it to other functions make sure to pass it as a reference. It is a derivative of xml_node.

xml_node

This class is the one to use for adding information to the document. It is primarily made up of a name and value.

allocate_node

This is a function of the xml_document class which creates an xml_node within it's internal memory pool. This function is the reason why xml_document objects need to be passed around by reference. Any xml_node objects will be deleted when the xml_document they were allocated with goes out of scope.

append_node

It is important to note that allocate_node does not add the node to the document. This function is what serves that purpose. It is a function of the xml_node class meaning nodes can be appended to other nodes, which is how the tree structures are achieved.

first_node

This function returns the first child node of the node it was called from. Keep in mind that the xml_document class is a derivative of xml_node. This function has an optional parameter to search through it's immediate child nodes by name. If it does not find a node with a matching name, or no node at all, it will return 0.

next_sibling

This function returns the next immediate child node of it's parent node. Like first_node, this function can also search for nodes by name.

We have leveraged this functionality in the program to not require xml documents to be rigid in the order of the major topics of the configuration. E.G. The list of agent configurations to use can come before or after where to save the output file of data recording.

allocate_string

The Unreal engine has macros to convert FStrings into char arrays, however they do this by way of pointers linked to the FString object. When the FStrings go out of scope the char arrays get cleaned up with them. The allocate_string function makes a copy of the string in the xml_document's memory pool.

Output to File

The RapidXML library comes with a left shift operator for basic_ostream derived streams which prints the entire xml_node (or document) to the stream.

Reading From File

file The library comes with a 'file' class which has a constructor that takes the filename as a parameter and initializes itself from that file.

parse The xml_document class has a 'parse' function which wants a character array parameter, which can be obtained with the 'data' function of the 'file' object. This function builds the xml_document with all of the information from the file, it is essentially the opposite of the left shift operator. There is more pointer foolery happening in the parse function, it is recommended to do all of the operations on the xml_document to construct the configuration object while the file object is still in scope. The parse function will throw a 'parse_error' if the contents of the file do not match the expected format.

3.4 How to Extend it

3.4.1 Adding New Information to Existing Config Classes

There are 4 places that need to be updated when adding new information to config classes:

- Getters and setters
- Saving it to the xml_document in the 'AppendDocument' function
- Initialising from an xml_document in the 'InitializeFromXML' function
- Instantiating it in the world in the 'Instantiate' function.

Take examples from the existing functionality.

3.4.2 Creating New Config Classes

All config classes are derived from ConfigBase, and must implement the three empty functions: AppendDocument, InitializeFromXML, and Instantiate. The config's Instantiate function will need to be called from within the ScenarioConfig or some other config class where appropriate. The configurator will also need to have a new entry in its LoadConfig function where it checks what type of configuration it is working with.

Chapter 4

Communications

4.1 Overview

This document describes the structure and mechanics of the communications system. It will detail how the system works and how to extend it.

4.2 How it Works

The system is designed to use radio propagation models which estimate the signal loss primarily based on the distance between the two positions. There are four main classes that anything using the communications system may have to interact with. MessageSender, MessageReceiver, CommDistributor, and Message.

4.2.1 MessageSender Interface

This interface must be implemented by any class designed to send messages. The intention is that classes implementing this interface will keep track of their maximum signal output power. The design also includes "variance", which is a measure of how widely the signal is sent onto nearby frequencies.

4.2.2 MessageReceiver Interface

Any classes intended to receive messages from the communication system must implement this interface. It is expected that part of the initialisation process of any classes which implement this add themselves to the Communication Distributor. Both interfaces are purely virtual and can both be implemented in the same class.

4.2.3 Communication Distributor

The communication distributor, or CommDistributor as it is called in the code, is the main hub of the system. It is a static class whose purpose is to act as the interface into the communications system. Messages get sent into it by MessageSender classes, and it will then send the messages out to every MessageReceiver listening on that particular frequency. Hidden away in the CommDistributor is the rest of the logic of the system. When the CommDistributor receives a message it will tell the channel object that represents the radio frequency of the message to send the message to all of the MessageReceivers listening on that frequency.

Channels

Channels, or CommChannels as they are called in the code, are the encapsulation of a frequency. They contain the frequency, the propagation model to use on that frequency, and a list of MessageReceivers to send messages to when the CommDistributor gives it one. The channel will calculate the signal-to-noise ratio and pass it along with the message for the receiver to interpret.

Propagation Models

Propagation models have been encapsulated into a class called SNRModel. This class is an abstract base class for models to derive from. The only thing it contains is a function for calculating the estimated signal-to-noise level between two points. The system has been created with polymorphism as the core concept around using these models and new models can be created and used easily because of this.

Multiple Models Used Simultaneously

The communication system also provides support for using different propagation models Simultaneously. This is encapsulated in a class called SNRModelFrequencyRange (FrequencyRange for short). This class links an SNRModel to a numerical range of frequencies. The idea is that one model will be used by default in cases where no FrequencyRanges cover a particular frequency, but the model in the FrequencyRange will be used when it does. If multiple FrequencyRanges overlap the CommDistributor will use the model from the first one in the list.

4.2.4 Message Class

The system has been designed to emulate sending a stream or packets of data through a network. Receivers need to know what to listen for in order to interpret messages properly. As such, the communication system has been designed to be brittle when it comes to interpreting messages. Each receiver will need to

know how to interpret each type of message it is designed to operate with. The messages can't interpret themselves.

A template message class has been provided to simplify the use of this design. Rather than making a new message-derived class you could instead create an enumeration or something like that and pass it through the communication system using the message template class.

4.2.5 How it All Works Together

When the simulation starts, the CommDistributor is initialised with whatever the configuration system pushes into it. A default model to use and any FrequencyRanges. Part of the process of initialising MessageReceivers involves adding themselves to the CommDistributor. This involves creating a new CommChannel if one doesn't already exist, and checking if any FrequencyRange objects exist for the particular frequency the receiver is being added for.

When a message is sent, the CommDistributor looks for the CommChannel that matches the frequency being used and tells the CommChannel to pass on the message to all MessageReceivers it knows about.

4.3 Classes That Need to be Defined

4.3.1 Transceivers

There is currently a PerfectTransceiver class available in the code. This is only meant to be used as a way to test and demonstrate that the communication system works. It is expected that a proper transceiver-style class (one that implements both MessageSender and MessageReceiver) would be created. There is also a TransceiverControllerComponent class whose intended function is to behave as the player, or offer them the means to interact with the PerfectTransceiver. The workload is split in this way so that the transceiver is responsible for interpreting the SNR related activities while the controller is responsible for interpreting and sending messages. Think of it like a person and their radio.

4.3.2 Propagation Models

Any number of propagation models can be created and used with no changes needed anywhere in the code. The configuration system uses the reflection system in the Unreal engine to find all subclasses of SNRModel so there'll be no trouble.

Chapter 5

Data Recording

5.1 Overview

This sectio outlines how the data recording system was design. How it works and how to extend it.

5.2 Design Concept

The data recording system utilises a subscriber design pattern. Data collector objects are attached to the recorder which polls them at a set rate. The advantages of this design makes it agnositic to errors in the data it is recording.

5.3 Creating Data Collectors

To create a custom data collector you have two options. You can create a custom datatype and then use the templated methods of the provided data recorder to collect it. The only requirement is that you correctly utilise the operator¿¿ override method for writing out to csv.

Otherwise you can derive from the Data Collector class and override the methods you require.

```
template <typename T>
class VEHICLETESTBED_API DataCollector : public DataCollectorBase
{
        DECLARE_DELEGATE_RetVal(T, FTypeDelegate)

public:
        FTypeDelegate FGetDelegate;

        ///<summary>Default constuctor, calls base class default</summary>
        DataCollector();

        ///<summary>Name constuctor, calls base class name ctor</summary>
        DataCollector(FName name);

        ///<summary>Full constuctor, calls base class full ctor</summary>
        DataCollector(FName name, bool bEnable);

        ///<summary>virtual destructor</summary>
        virtual ~DataCollector();

        ///<summary>Collects the datavalue if delegate is bound</summary>
        ///<returns>Unique pointer to <see cref="DataValueBase"/> object with value</returns>
        virtual std::unique_ptr<DataValueBase> Collect() const override;
};
```

Figure 5.1: DataCollect Code

5.4 Binding the Delegates

The data collectors uses the Unreal Engine delegates system to bind the methods to the collectors. Information on this system can be found in the Unreal Engine docuementation.

```
// Add collectors to data recorder
DataCollector<int32>* myCollector = new DataCollector<int32>();
myCollector->FGetDelegate.BindUObject(this, &AVehicleTestbedGameModeBase::GetNumPlayers);
dataRecorder->AddCollector(myCollector);
```

Figure 5.2: Delegates example

Chapter 6

Event Recording

6.1 Overview

This document describes the design intentions and workings of the event recording system, and most importantly how to use it.

6.2 Design

The event recording system needs to be dynamic enough to record any information from anywhere in the system. To suit this, the event recorder has been designed as a static class, accessible from anywhere. Because it is accessible from anywhere, the recording system needs to be self-sufficient. Therefore the recorder offers functions for recording events which only use basic Unreal engine classes: UObject, FString, and TMap.

6.3 How the Recorder Works

Events are pushed into a thread-safe queue from anywhere in the program. The system uses the 'async' library to create a thread which continuously writes the queue to file. The recorder is started and stopped with appropriately named functions. Each time the event recorder is started, it creates a new file with the date and time appended to the name. This means if the event recorder is started in the game mode class for example, it will produce a new file each time a new level is opened. The event recorder technically doesn't need to be stopped as it will stop on its own when the program is closed, though it may improve performance to stop it when not needed.

6.4 How to Record Events

Simplicity is quality. There are three functions to record events with, two accessible from within Unreal's blueprints.

The first takes a string to describe the event and a UObject from which its name and the game time are extracted. In many cases this function will be all that is necessary. Take care in which UObject is passed in. In the case of UActorComponent derived classes it may make more sense to pass in the owner of the component as the component's name will not distinguish which actor it is part of. E.g. "Jackal2 received a message" rather than "Radio received a message".

The second function takes the same parameters as the first but with an additional map of FString pairs. The purpose of this map is to add any additional information, such as the SNR of a received message from the communications system.

The third function takes a FRecordableEvent object directly. More on that later. The other two functions actually create a FRecordableEvent object from the parameter information and call this third function. This function is not available to blueprints.

6.5 Recordable Event Class

The event recorder uses Unreal's smart pointers when dealing with FRecordableEvent objects which means they will need to be created with the 'new' keyword but do not need to be deleted as the smart pointers will do that.

The event recorder is designed to use its FRecordableEvent class by default, but it was designed with polymorphism in mind if a use case for that appears. The FRecordableEvent class is responsible for its own xml output format. It has a getter function for the xml output as an array of FStrings.

Note: The xml output should not include line endings as they will be appended by the Unreal engine.

6.6 Writing to File

The event recorder's output location can be set with the configuration system. Writing to file is handled by the Unreal engine because it handles the different file endings used by different operating systems. The event recorder collects the xml output of each recordable event and gets the Unreal engine to write them to file. Take note that the file will not exist until an event is written.

6.7 Performance

In our testing we did not find any concerns regarding performance. The recorder has a simple optimisation built in to only write to file when it has collected

more than 1000 lines of text or if the event queue is empty. Even on a 5400rpm mechanical hard drive the writer thread did not lag behind when testing with thousands of events per second.

Chapter 7

ROS AI and Integration

7.1 Overview

This section provides implementation details for the AI components used in the project. There are three parts to the AI: the ROS plugin, which enables communication between Unreal and ROS; the ball tracking node, which identifies and locates incoming tennis ball threats; and the threat response AI, which rotates the Jackal to face a shield towards incoming threats. Each of these components will be explained in their own subsection.

A basic understanding of ROS is considered assumed knowledge for this section. For more information on ROS, consult the ROS tutorials or Jason O'Kane's Gentle Introduction to ROS.

7.2 ROS Integration Plugin

Within this project, an open source plugin was used to enable communication between ROS and Unreal. The plugin is provided with the project source code, but is also available on GitHub. The rosbridge package is used by the plugin to facilitate communication with a ROS master.

The ROS master can be run using either the provided Docker container, a custom Linux container or a physical computer running on the same network as the computer running Unreal. It is essential that the container or machine have the rosbridge package installed. This can be installed using the command below in a terminal:

```
# sudo apt-get install ros-kinetic-rosbridge-suite
```

Note that the above command uses ROS Kinetic. At this point in time, we have not been able to make the plugin work with other ROS versions. In particular, ROS Indigo was tested quite extensively without success. More information on this can be found in Section 7.2.4.

7.2.1 Launching the provided Docker container

You may choose to use the provided Docker container as your ROS master machine. This container may be useful if there is no access to a physical Linux machine. In order to run the container, you must have Docker installed on your machine. It can be downloaded from the Docker Website.

Once Docker is running, the container can be started by entering the following commands into a command line interface from the directory storing the container:

```
# docker run -it --rm -p 6080:80 -p 9090:9090 roswscontainer
```

The container will start up a VNC server connected to a machine with ROS Kinetic and rosbridge. You can connect to the container by navigating to 127.0.0.1:6080 in any web browser.

In its current state, this container contains the catkin workspace used on the Jackal. You can download your own code to the container using git clone, or similar techniques. If you make changes to the container and wish to save your setup, you can do so using the following method:

1. On your physical machine, open a new tab or window of your CLI and type the command `docker ps` to find the container ID of your container.
2. Use the command `docker commit <container id> <chosen container name>` to commit the container to memory.
3. Save the container by using the command `docker save <chosen container name> > <chosen file name>.tar`
4. Load the container with the command `docker load -i <chosen file name>.tar`
5. You should now be able to stop and re-run the container by substituting roswscontainer with your chosen container name in the docker run command above.

7.2.2 Configuring the plugin

There are several steps required to configure the plugin for first-use. Many of these steps should be completed as part of the provided project source code, however it is worth working through these steps at least once to check the configuration is correct.

1. Open the Plugins settings in Unreal (Edit → Plugins) and search for the ROS Integration plugin. Check that it is activated, clicking the `Activate` button if it is not. A restart may be required to fully activate the plugin if it was not previously active.
2. Open the Maps and Modes settings (Edit → Project Settings → Maps and Modes) and check that the Game Instance class is set to "MyROSIntegrationGameInstance".

3. Open the MyROSIntegrationGameInstance blueprint (Content → MyROSIntegrationGameInstance and check the correct settings of the ROSBridge Server Host and Port. Rosbridge servers are by default started on port 9090, so you should not need to change that value. The host may need to change depending on your setup: if using a Docker container, the host should be 127.0.0.1, if using a seperate machine it should be the IP address of that machine.

The current form of the project provides a player controller (Testbed ROS Player Controller) that subscribes to the `/cmd_vel` topic to get the linear and angular velocity for the controlled agent. Use of this control has not quite been fully integrated into the control system and so requires an additional configuration step. The ROS player controller can not be switched to automatically and must instead be configured in the Unreal Editor. This can be done in the World Settings (on the right side panel).

If you do not wish to use the player controller, you can also test functionality of the ROS connection with the very basic TestPublisher and TestSubscriber classes. These can be added to the project by navigating to C++ Classes → ROS Integration within the Unreal editor and dragging the objects into the viewport.

If you wish to create additional publishers and subscribers, you can use the example publisher and subscriber in the ROS Integration README as a template.

7.2.3 Using the plugin

Once the ROS machine and Unreal have been appropriately configured, it should be possible to run the plugin. On the Unreal side, all that is required is to launch the level. The ROS machine requires a command to start the rosbridge server. Simply enter the following command into a terminal:

```
# roslaunch rosbridge_server rosbridge_tcp.launch bson_only_mode:=True
```

If the plugin is working correctly, you should see the rosbridge server correctly launched on port 9090. A message should appear when the Unreal client connects (and disconnects). If there are any subscribers in the Unreal code, a print statement in the ROS terminal will indicate when they have subscribed.

If the Unreal code is publishing messages to ROS, this can be confirmed by opening a second terminal tab or window and typing the command `rostopic echo /<topic name>`. You should be able to see messages appear as they are sent from Unreal.

7.2.4 Troubleshooting

Throughout development, we encountered a number of issues with the plugin that we managed to resolve. If the plugin is not functioning as expected, we recommend starting the troubleshooting process by looking through this section to see if the problem is described here.

Issue 1: Client connects to rosbridge, but no messages are sent or received

- After launching an Unreal level, a connected ROS machine shows a message like "[Client 0] connected. 1 client total".
- The usual message, "[Client 0] Subscribed to /example_topic" does not display.
- Running `rostopic echo /example_topic` will produce the message "WARNING: topic /example_topic does not appear to be publshed yet".
- /example_topic is not shown in the list of topics generated by the command `rostopic list`.
- Wireshark analysis confirms that messages are being sent from one machine to the other.

Solutions We have found two possible causes for this issue. The first is that the user missed `bson_only_mode:=True` in the roslaunch command. If this is the case, relaunching the rosbridge server with this extra part should resolve the issue.

This issue was also encountered when using ROS Indigo. As the manifestation of the issue is the same as when not using `bson_only_mode`, we suspect that the root cause of the problem here is a difference in handling BSON between the versions of rosbridge for ROS Indigo and Kinetic. The `bson_only_mode` flag was added in a later version of rosbridge, however it does seem to have been implemented for ROS Indigo. It is possible that the repository has not been updated to include this version of rosbridge.

If ROS Indigo must be used, our recommendation is to start by trying to install the latest version of rosbridge from source, rather than using `apt-get`.

Issue 2: Unreal client is unable to connect to rosbridge

- After launching an Unreal level, no connection messages are seen on the rosbridge machine.
- There is no sign that any messages have been received from the Unreal side.

Solutions There are many possible causes for this error. The best method for troubleshooting would be to work through the launch and configuration steps described in Sections 7.2.1 and 7.2.2. In particular, focus should be given to the following areas:

- Ensure that the plugin has been enabled within Unreal
- Check that the ports and host are correct within the game instance settings

- Make sure that the Game Instance is set to MyROSIntegrationGameInstance
- Check that there is something within the level (either a player controller or agent) that publishes and subscribes to messages.
- If using a Docker container as the ROS machine, make sure that port 9090 has been exposed by including `-p 9090:9090` somewhere in the run command.
- If using a physical ROS machine, ensure that you are on the same network by attempting to ping the IP address you have provided in the rosbridge host settings.

Issue 3: The program connects and sends/receives messages, but Unreal crashes shortly after beginning.

- After launching the Unreal level, the usual connection and subscription messages can be seen on the ROS machine.
- Messages may be sent and received as expected for a short time.
- Unreal crashes within about a minute of launch.
- Problem occurs consistently.

Solutions A potential explanation for this issue could be that the ROS topic is being garbage collected by Unreal. When the topic goes out of scope, it is deleted and causes an unhandled exception to be thrown.

If this is the problem, it can be resolved by declaring the topic as a class property and placing "UPROPERTY()" on the line immediately above its declaration. This is discussed within the related issue on the plugin's Github page.

Of course, there are many other reasons that Unreal could crash - the issue may be totally unrelated to use of the ROS plugin. Further troubleshooting may be required to locate the source of the crash if this is the case.

7.3 Threat Detection AI

The threat detection AI detects and locates a tennis ball in an image. This functionality has been provided in the form of a ROS package which contains AI nodes, tests, configuration and launch files. This section will outline the design and use of this package.

7.3.1 Design description

The ball tracking node is the heart of this package and its functionality. This node subscribes to an incoming ROS image message, processes the image to

Parameter name	Description	Default Value
calibrate	Toggles calibrate mode	False
record_video	Toggles record mode	True
image_topic	Name of the image topic	"/test_image"
frame_width	Width of the camera frame (pixels)	1280
frame_height	Height of the camera frame (pixels)	1024
focal_length	Focal length of the camera	1
known_width	Width of the tennis ball (metres)	0.065

Table 7.1: Ball tracking parameters

find a tennis ball and, if found, publish the location of the tennis ball to the `/threat` topic. The ROS image message is converted to an Open CV frame format through the use of the ROS CV Bridge package.

The technique used for identifying the tennis ball was adapted from the method shown on the Py Image Search blog. Open CV is used to identify contours matching the green shade of the tennis ball. The centre of the tennis ball is found by summing the moments to find the centroid of the contour. Distance can be estimated based on the number of pixels the tennis ball fills, although it was found that using the raw number of pixels worked just as well for the AI.

If required, the image can be processed to place a circle around the identified tennis ball, and a mark at the centre. This image can be recorded to a video file, or, if a monitor is connected, displayed in real time while running. The feature can be toggled using a parameter and is on by default.

The output of the node is a custom message type, `threat_detection/Threat`. The message contains the width and height of the image, the x-y co-ordinates of the threat's centre point and the distance to the threat. Currently, the distance is actually the radius as it was found the AI node responded better to that than the distance measurement.

7.3.2 Parameters

The node has a number of parameters that enable the ball tracking node to be used in a variety of situations. Parameters all have default values and can be changed at run-time using either launch files or other parameter setting methods within ROS. The parameters used are summarised in Table 7.1.

7.3.3 Usage

The package is provided with a number of launch files to make using the node as easy as possible. Parameter files and launch files are provided for both of the cameras on the Jackal. Testing found that the node functions better with the Axis camera due to a wider field of view and better focus. The launch file for this camera can be run by entering this command into a terminal window:

```
# roslaunch threat_detection ball_tracking_with_axis.launch
```

This file launches an additional node. This node converts the image format from the axis camera (WFOV_Image) to a normal ROS sensor_msgs/Image message.

One launch file sets up the node to work in calibrate mode with the Axis camera. Calibrate mode is used to find the focal length of the camera so that distances can be calculated. To calibrate the camera, hold the tennis ball 1m from the camera and write down the focal length shown on the screen. The value of the focal length and the width of the tennis ball in metres should be added to the launch file. Calibration mode can be run using the following command:

```
# roslaunch threat_detection calibrate_with_axis.launch
```

The final launch file in the package can be used to test tennis ball identification with static images. It may be worth running this launch file if changes are made to the node, to ensure that identification still works as expected. The unit tests in the package can also be used for the same purpose.

7.4 Threat Avoidance AI

The threat avoidance AI is a node within ROS which makes the Jackal rotate to face its shield towards an incoming threat. At this point the behaviour of the AI is very basic, but could be upgraded in the future.

In its current form, the node subscribes to the `/threat` topic published by the ball tracking node. It compares the distance of the threat to previous threat messages to check whether the distance is decreasing (i.e. the threat is approaching). If the threat is approaching, the node will look at the x position of the ball to find the angle of the ball from the Jackal's centre. Based on that angle, the Jackal will rotate itself so that the shield on the right side of its body is facing the incoming threat.

The calculated movement is sent to the Jackal using the `/cmd_vel` topic. This consists of a geometry_msgs/Twist message specifying the linear and angular velocity for the Jackal. This is passed to the Jackal's inbuilt motor control software.

7.4.1 Usage

The threat avoidance AI requires the use of the ball tracking node to identify threats. At this point, the AI node has not been integrated into a launch file with the ball tracking node. This could be done easily in the future to make launching the nodes easier.

For the moment, the AI node is launched with the following command:

```
# rosrun jackal_ai ai.py
```

7.4.2 Limitations of AI

As noted earlier in the section, the AI is currently quite basic. There are a number of limitations in its behaviour. One of the big issues is in identifying

whether an incoming tennis ball is a threat. As the current method is based only on the number of pixels the ball takes up, it will trigger whenever the ball first appears in the Jackal's field of view regardless of whether or not it is heading towards the Jackal. As a result of this, the Jackal can react to balls that are not a threat, or even move into the path of a threat. This could be fixed by using the distance and x value together to see if the ball is moving into the path of the Jackal.

Another issue is that the resolution and field of view of the camera are quite limited. This means that the tennis ball needs to be quite close to the Jackal to be detected. The ball therefore needs to be thrown quite slowly to give the Jackal enough time to react to the threat. It also means that the Jackal can only identify threats that are in front of it. For this reason, the Jackal currently only rotates to the right as that will always be the shortest path to any threat the camera is capable of detecting. If a better camera is used, more sophisticated algorithms will be required to find the optimal path to safety.