A MODULAR AND EXTENSIBLE USER INTERFACE FOR THE TELEMETRY AND CONTROL OF A REMOTELY OPERATED VEHICLE

Tyler Morrow (Student)
Kurt Kosbar (Advisor)
Telemetry Learning Center
Department of Electrical and Computer Engineering
Missouri University of Science and Technology

ABSTRACT

This paper discusses the rover engagement display (RED), an application that integrates network communication, control systems, numerical and visual analysis of telemetry, and a graphical user interface for communicating with the embedded systems of a remote vehicle. The target vehicle is a wheeled rover participating in the University Rover Challenge, a competition that observes the performance of rovers in an environment similar to that of the planet Mars. Communication with the rover occurs via a TCP connection and messages adhere to a simple protocol. The RED user interface is visually modular in an attempt to provide additional scalability and extensibility. Control algorithms, user interface design concepts, and code architecture (C#) are discussed.

Keywords: Graphical User Interface, Remotely Operated Vehicle, University Rover Challenge

INTRODUCTION

The Mars Rover Design Team (MRDT) at Missouri University of Science & Technology was founded in the spring of 2012 to compete in the Mars Society's University Rover Challenge (URC). The URC is an annual, international competition held in desert of southern Utah in the United States that tests the next generation of rovers (Figure 1) that will hopefully, one day work alongside astronauts exploring Mars [1].

Members of MRDT contribute in ways specific to their skillsets and interests, which support the primary objective of completing tasks at URC. Completing tasks requires one or more operators to control the rover with a computer from a remote site, referred to as the base station, where the operators have no line of sight to the rover. In order to perform the tasks from a position of little information, MRDT wanted a single base station application that would seamlessly integrate telemetry and controls. In this paper, we will focus primarily on the design and implementation of the base station application, as well as the observed results from its use during development, integration, and competition. Still, it is important to mention rover systems at times as they have a direct influence on the base station application (a paper on those systems has also been submitted to the International Telemetering Conference [2]). For example, the integration of our robotic arm introduced 12 commands from which to choose in order to operate it. So while we

may introduce rover systems at times, the primary focus is how the base station application is designed in response to the design of the rover and the needs of the operators.

Figure 1. The MRDT 2014 Mars Rover

In early May 2013, the first incarnation of a base station application, dubbed *MRDT-GUI*, was started, roughly one month before the MRDT's first competition [3]. The application used for competition was rushed due to the limited time allotted. Consequently, the end product only contained the functionality needed to operate the rover at a basic level, and only telemetry that was deemed critical was transmitted. After URC 2013 (MRDT's first competition), a plan was put in place to develop a base station application that would not need to be rebuilt from the ground up each year. The current base station application used by MRDT is known as *Rover Engagement Display (RED)*. Initial development of RED had been private up until URC 2014 when it was published on GitHub, where it will remain as an Open Source project under the GNU General Public License (v2) [4]. In the long-term, RED may be able to service more than just MRDT as a telemetry and command interface for a remote, embedded system.

RED employs some of the latest technology available on the Microsoft® .NET platform. By using the most recent releases of certain technologies, we hope to increase the longevity of the project in terms of interest and support. The primary Microsoft technologies used include:

- .NET 4.5 / C# 5.0
- Windows Presentation Foundation (WPF)
- Visual Studio 2012/2013

Using the latest technologies also gives members of MRDT the opportunity to gain experience that will make them more marketable in the software development industry.

REQUIREMENTS

The primary goal of RED is to maintain a connection with the rover and give operators the ability to control, configure, and monitor the rover. The following requirements outline the essential functionality needed to accomplish this:

1. Establish a redundant network connection that allows the simultaneous sending and receiving of data on both ends of the connection.
2. A standardized format for data sent to and from the rover.
3. A user interface that displays received telemetry and input in an organized fashion.
4. Processes that capture controller input from operators and send appropriate commands to the rover.

RED satisfies the first requirement by acting as an asynchronous TCP server that treats the rover as a client. Resources covering network programming on the .NET Framework can be found on the Microsoft Developer Network [5]. Once the connection is established, RED parses and composes any messages it receives or sends as JavaScript Object Notation (JSON). The serialization and deserialization of JSON is handled by a third-party library called JSON.NET [6]. Messages received from the rover are routed to classes that manage specific areas of the user interface. At execution, four threads are created that are responsible for reading and interpreting controller input. Once valid input is given, one of these threads will send the correct command message over the connection to the rover. We will cover the controller a little more later on.

NETWORK COMMUNICATION

Communication can be visualized as a constant stream of bytes being passed over the network between the base station and the rover (Figure 2). Furthermore, communication can be organized as either telemetry (messages from rover to the base station) or commands (messages from the base station to rover). Currently, the direction of message flow determines the category into which a message falls, but in the future this may not be the case. For example, if a software update is made on the rover that adds telemetry or commands, the rover would ideally inform the base station the next time they connect. This could be considered the rover commanding RED to alter its user interface to display the new telemetry. Another example would be the rover sending messages to RED that prevent the operator from continuing a potentially harmful operation. This is effectively the rover deciding what is best for it, as opposed to RED analyzing telemetry to make that decision. The latter example requires a more complex rover, but it is one of many steps in the direction of autonomy.

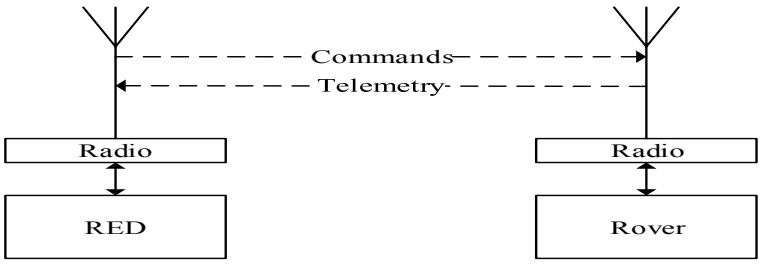

Figure 2. RED-to-Rover Communication Structure

CONNECTION MANAGEMENT

As previously mentioned, RED employs an asynchronous TCP server for managing its network connections, which is just a class initialized at application execution. Asynchronous methods allow long-running I/O operations from the network to be processed while user interface (UI) rendering and logic remain responsive [7]. From the UI, operators can initiate *listening* on the internal TCP server with a button click. Listening puts the server into a state of accepting TCP connections from clients. The UI also contains a *disconnect* button that halts listening, and closes all existing connections. The .NET framework provides numerous classes with which we can manage connections. We opted to use the TcpListener and TcpClient classes with the Asynchronous Programming Model (APM) [8] [9] [10]. Using these classes simplified the programming of the TCP server by wrapping lower-level classes and providing methods that help with common network operations. While APM itself does not necessarily make the programming any easier, plenty of examples could be found that enabled us to build off of until we achieved what suited our present needs. The most recent version of .NET introduces the async/await keywords for asynchronous operations, something that we are considering for future releases.

TCP was chosen to provide the reliable delivery of data to and from the rover. While it is important that telemetry be received and displayed in real-time on RED, it is more important that the commands, such as those for driving, arrive at the rover. If for any reason packets were to get out-of-order or never arrive, the rover may carry out actions detrimental to its well-being. Currently, due to hardware limitations and time, very few measures have been put in place to have the rover automatically detect when it is in danger. The primary protections implemented within the last year have been for the power system to ensure the safe charging and discharging of the batteries. The motors on the rover are programmed to apply the last value they received, so if no command is received with a value to stop, then the motors will not stop until the batteries are dead. With a reliable connection and by constantly surrounding the rover with people in the field, we have never experienced a scenario where the rover was damaged due to such a failure. However, it is a well-known possibility that we have full intentions of rectifying in the future with better awareness on the rover.

The rover's communication is conducted by a custom printed circuit board (PCB) containing an ARM processor, called the *motherboard*. Among other things, the motherboard is configured to always attempt to establish a connection with the base station, and once it has done that, continuously send telemetry and receive commands. The motherboard is configured to send 15 to 30 JSON messages every 100ms, providing a seemingly constant stream of data. If this stream of data ceases for a period longer than five seconds, RED will assume that the connection is bad and close it properly. If nothing fatal has occurred on the rover, the motherboard will reconnect and operation can resume.

Typically, the network connection with the rover failed due to damaged cords, power issues, or poor signal quality. Damaged cords were the result of insufficient wire routing or stress from activity from operating the rover. Any power issues resulted from improper management of discharging power by the PCB dedicated to this task. Lastly, poor signal quality came into play

when the rover's distance exceeded what the antennas could handle, or when the rover broke line of sight with the antenna behind an obstruction of considerable size.

MESSAGE FORMAT

Messages are serialized as JSON prior to being sent over the network where they have to be parsed by receiving entity. These messages contain only two things: (1) and ID number and (2) a value. IDs are unique integers that correspond to a particular piece of telemetry or command. Once the message is identified, the value is taken and applied to either updating the UI (on RED) or commanding a subsystem of the rover. JSON is a simple format, and third party libraries exist that provide simple serialization and deserialization support [6]. Regardless, the rover's limited hardware resources should be used sparingly. In order to ensure that more telemetry and commands do not hinder performance, we will be exploring an alternative, custom format that reduces processing time during serialization and deserialization, as well the amount of data transmitted over any network connection.

CAMERAS

It is important to note that RED does not manage the video streams to the several IP cameras on board the rover. These cameras receive unique IP addresses and are viewed through a separate application on dedicated monitors at the base station. This allows the connection through which telemetry and commands pass to be independent from the connections for video streams. This isolation has been particularly valuable in the field and when troubleshooting. If one camera is damaged, other cameras, telemetry, and commands will still function, and when that is not the case, it is due to a much larger issue such as power failure or a bad radio signal.

USER INTERFACE

RED utilizes Windows Presentation Foundation (WPF) to render a user interface for operators to view and interact with telemetry. RED integrates this user interface with many background threads that are responsible for tasks such as reading controller input, managing the network connections, serializing outgoing messages, deserializing incoming messages, and conducting the business logic that is ultimately visualized on the UI. All of this functionality was organized in conjunction with Model-View-ViewModel (MVVM) design pattern (Figure 3) leading to a modularization of functionality.

MODEL-VIEW-VIEWMODEL (MVVM)

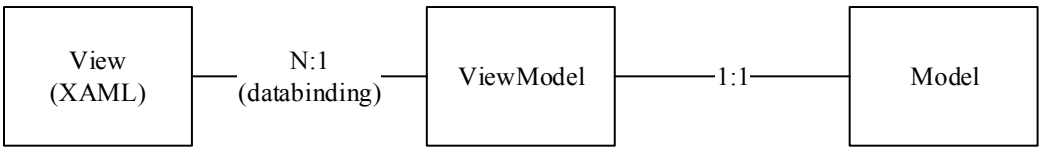

Figure 3. Model-View-ViewModel Relationship Structure in RED

The View portion of MVVM corresponds to the markup code associated with the user interface known as Extensible Application Markup Language (XAML) [11]. This markup is rendered by WPF into what users see on their monitor. XAML offers an alternative to programmatically

setting up the UI in code-behind. Views are just classes (such as a Window) that derive from a long list of .NET classes where they obtain many different properties, event handlers, and methods. One of these properties is the DataContext which we use to dynamically set a class that serves as the business logic for a particular View. In addition, the same ViewModel instance can serve multiple Views. The ViewModel can be seen as the mediator between the data objects, the Models, and the Views. Having this separation can be beneficial when attempting to share the same data between multiple Views. Two separate Views can set the same ViewModel instance as their DataContext, giving them the ability to display the same information in multiple locations on the UI. In practice, this is not the exact implementation we use, but rather we have modified this approach by having ViewModels contain other ViewModels as members in a process called Composition. Dependency Injection is then implemented to share ViewModel functionality across ViewModels.

DATABINDING

The ability to display data on View from its DataContext is achieved through a process called *databinding* [12]. Views are composed of controls, such as a TextBox, which bind to public properties in the View's DataContext (a ViewModel). These properties encapsulate a portion of the Model by providing a getter and setter to the View. Anytime the property's setter is executed, an event is raised that notifies the UI thread of a change. The UI thread then has every control bound to that property call the property's getter for the updated value. Upon doing so, the UI updates to reflect the change in value. Logic can be used within the getters and setters of these properties to format the data. Setters can also be removed from these properties when a read-only experience is all that is needed.

MODULES

RED's primary window is called the Control Center, which is itself a View. By design, nearly all operator functions can be performed from this screen. Nonetheless, appearance and system settings are not configurable from the Control Center and must be accessed by clicking the *settings* link at the top-right of the Control Center. The Control Center contains five rectangular placeholders the can contain *modules*. Modules are just Views that can be loaded dynamically at runtime into the placeholders. Recall that Views have access to ViewModels which contain the business logic and bindings for that View. Therefore, modules can also be thought of as both the View and its corresponding ViewModel. Modules are loaded by using the Module Manager (Figure 4) located on the far right column of the Control Center. To load a module, the user selects a module from the list of available ones. The user then assigns it to a placeholder by clicking one of the buttons below the list. These buttons are visually structured to correspond to one of the placeholders. Once loaded, the placeholders can be adjusted in size by clicking and dragging the lines that separate each placeholder. The right column also contains the Message Center (Figure 5), a read-only console that serves as a debugging tool at run-time. A list of basic information pertaining to the state of the system can also be found on the Control Center.

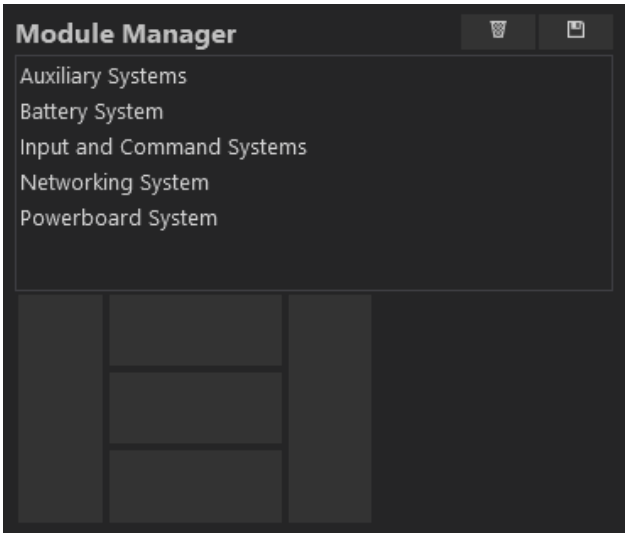

Figure 4. Module Manager

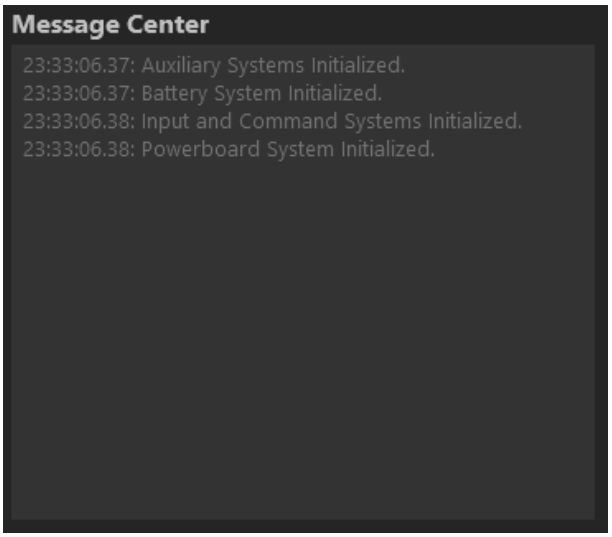

Figure 5. Message Center

The Module Manager also provides a way to save and delete a particular set of modules, as well as the size of their containing placeholder, that have been loaded between application executions. This serves to save the operators from having to manually load each module every time they use RED. Users must designate a name for a particular layout before saving. When a layout is saved, each placeholder's contents and size are saved to a local XML file along with the given name. Once saved, the user will see a button with that name appear at the bottom of the Control Center. Clicking that button will adjust the placeholders to the layout specified in the XML file.

The separation of the UI and business logic with WPF and the MVVM pattern is conducive to this modular UI approach. Once the basic framework for loading modules was established, new features could be implemented as modules. Once modules are completed, they integrate immediately with the Module Manager. This integration is ensured by implementing our IModule interface on ViewModels used by Views that are intended to load into placeholders. Placeholders are ContentControls (C# Class) whose content is bound to properties of type IModule, the interface implemented by module ViewModels. When these properties are set to a ViewModel implementing IModule, the placeholders update their content as a new instance of a View associated with that ViewModel. The DataContext for this new View is the ViewModel implementing IModule that was just set. This is a WPF feature known as Data Templates. Any ViewModel that implements IModule has the following:

- A *Title* property of type string
- An *InUse* property of type Boolean
- An *IsManageable* property of type Boolean
- A *TelemetryReceiver* method that takes an IProtocol<T> as a parameter.

All module ViewModels are initialized one time when RED is executed, as opposed to their associated Views which are initialized every time modules are loaded. The module ViewModels are stored in a List structure where they can be retrieved as needed. The Title property of IModule provides a way to look up the module ViewModels within that List. Selecting only the Titles for every ViewModel in that List generates a list of names that are used in the Module

Manager's list of available modules. However, only module ViewModels that have their IsManageable property set to true will appear in the Module Manager. The InUse property is used to enable a key part of the Module Manager's logic when loading modules. Any module that is currently loaded to a placeholder will have its InUse property set to true. By checking this property when loading a module, we prevent the same module appearing in more than one placeholder simultaneously. Lastly, the TelemetryReceiver method is the primary method that allows data to flow from one module to another. The most important example of this is the flow of data from the Networking module to every other Module that displays telemetry from the rover. When a message is received and parsed by the NetworkingViewModel as all incoming messages are, it is routed to the TelemetryReceiver of the appropriate ViewModel by accessing the aforementioned List. At design-time, all possible telemetry is known and module Views are designed to have dedicated controls, such as a TextBlock, for each piece of telemetry. Since these controls bind to properties in the ViewModel, the TelemetryReceiver implementation only needs to set the appropriate property to the new value received in order for the UI to update.

The IProtocol<T> interface is the contract used to ensure that telemetry and commands maintain the Id/Value format. Classes implementing IProtocol are serialized into the JSON that gets sent over the network to the rover. Data received from the rover must be deserializable as an IProtocol in order for it to be routed to a module. With many messages being received every second during a connection, care was taken to gracefully handle the deserialization of messages that were corrupted or improperly formatted. JSON.NET deserialized every message as a Protocol<T> class, which implements the IProtocol<T> interface. The IProtocol<T> interface contains the following:

- An *Id* property of type integer
- A *Value* property of type T

Having IProtocol be generic allows RED to serialize and deserialize JSON messages containing any primitive C# data type.

CONTROLLER

An Xbox controller is used by RED to get command input from an operator. Reading and interpreting this input is managed by the Input ViewModel (InputVM) and the XInput class in DirectX [13]. This input is visualized on the Input View in order to verify that all buttons and joysticks are working. When the InputVM is initialized, it creates a thread whose sole job is to read the current state of the controller every 30 milliseconds and map those values to properties within InputVM. Following this, three more threads are created that interpret that input and pass commands through the Networking ViewModel so they will serialized and sent to the rover. Each of these three threads are Timer classes that execute at regular intervals, each of which may be configured to be different. Every time each thread fires, it reads the current state of values read from the controller and decides whether or not sending a command is necessary. The three threads correspond to sending commands for driving, the robotic arm, and robotic arm gripper. The difference in intervals between the three threads was seen as necessary to control the rate at which messages are sent if necessary. For example, driving commands need to be sent the fastest because we needed as much responsiveness when trying to slow down or speed up rapidly

for a short period of time. In contrast, the robotic arm needed to seem less responsive to controls (or just slower) in order to provide the illusion of precision as we attempted to take advantage of six degrees of freedom.

VISUALIZATIONS

Due to limitations on time, mainly caused by the need to ensure successful integration, minimal efforts were applied to the implementation of visualizing and analyzing telemetry. Still, the battery management system was reporting throughout integration and competition. We took the time to keep the last ten messages received for each of the 15 telemetry messages for the battery management system. All of this was displayed on the Battery Systems module, and the historical aspect was represented as a series of vertical bars progressing across the screen as telemetry was received. This was enough to alert the operators to spikes in current or sags in voltage.

CONCLUSION

RED is developed to provide MRDT with a user interface that effectively manages and visualizes communication with the remote rover. RED's design contributed to a successful year at URC 2014. During competition, no significant issues arose from RED's design that impeded the operator's ability to operate the rover. Commands sent by RED were received by the rover, and all telemetry sent by the rover was received and displayed properly. All of this was done while maintaining a responsive user interface. Network issues caused by non-fatal events on the rover always led to a connection being re-established and operation continuing as normal.

Slight latency (less than half a second) in camera feeds was an issue that made controlling the rover difficult. The limited perspective and picture quality of the IP cameras we used only added to that difficulty. Nonetheless, there were enough cameras that operators could perform the tasks. It is also worth noting that the desert conditions took a toll on the base station computer and its peripherals. We foresaw this and brought spares for some things which turned out to be a good idea. Glare from sunlight also made viewing RED difficult when its dark theme was enabled. Switching to its light theme made viewing RED much easier in the desert sun.

MRDT intends to add more telemetry-generating components to the rover, giving operators a better picture of what is going on. Doing this presents a new challenge for those of us developing RED to create new ways of representing data, as well as storing it long-term. Implementing autonomous behavior on the rover is a long-term goal of MRDT, something that will require the generation, acquisition, and processing of much more data than we currently receive. There are over 60 telemetry and command Ids currently in use by RED, with many more well on the way.

Reducing any latency with communication, as well as increasing telemetry, enables the possibility of higher precision controls. In particular, haptic feedback on the robotic arm and suspension translated to vibrations on the controller are a simple, yet powerful manner in which to give operators insights into the environment around the rover.

By using C# and WPF, along with the MVVM design pattern, RED is in a good position for expansion. Currently, the telemetry and command Ids are hard coded as enumerations. While changing them is not difficult, eliminating that task would open the door to allowing those who are unfamiliar with RED's code to determine the commands it sends and telemetry it displays. Ultimately, we hope to empower the user to design new modules from within RED, a feature which would hopefully see RED serving more than just the needs of MRDT.

ACKNOWLEDGMENT

The Rover Engagement Display would not be possible without the contributions of the entire Mars Rover Design Team at Missouri S&T. MRDT provides the rover which make RED's existence meaningful. Recognition should also be given to Joshua Reed and Derek Allen for their contributions to the early Networking and Module code.

REFERENCES

[1] The Mars Society, "Requirements & Guidelines," [Online]. Available: http://urc.marssociety.org/home/about-urc.

[2] K. Johnson and K. Kosbar, "Telemetry Processor Design for a Remotely Operated Vehicle," Submitted to *2014 International Telemetry Conference*, San Diego, CA, USA, Oct. 2014.

[3] Missouri S&T Mars Rover Design Team, "MRDT-GUI - GitHub," [Online]. Available: https://github.com/MST-MRDT/MRDT-GUI.

[4] Missouri S&T Mars Rover Design Team, "Rover Engagement Display - GitHub," [Online]. Available: https://github.com/MST-MRDT/Rover-Engagement-Display.

[5] Microsoft, "Network Programming in the .NET Framework," [Online]. Available: http://msdn.microsoft.com/en-us/library/4as0wz7t(v=vs.110).aspx.

[6] J. Newton-King, "JSON.net," [Online]. Available: http://james.newtonking.com/json.

[7] Microsoft, "Parallel Processing and Concurrency in the .NET Framework," [Online]. Available: http://msdn.microsoft.com/en-us/library/hh156548(v=vs.110).aspx.

[8] Microsoft, "TcpClient Class," [Online]. Available: http://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient.aspx.

[9] Microsoft, "TcpListener Class," [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/system.net.sockets.tcplistener(v=vs.110).

[10] Microsoft, "Asynchronous Programming Model (APM)," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms228963(v=vs.110).aspx.

[11] J. Smith, "WPF Apps With The Model-View-ViewModel Design Pattern," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

[12] Microsoft, "Data Binding Overview," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms752347(v=vs.110).aspx.

[13] Microsoft, "XInput Game Controller APIs," [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ee417001(v=vs.85).aspx.