GCS Design Overview Document

V0.9(Working)

Table of Contents

Changelog

1. Concept
   1. Interface
   2. Network
   3. Idea – From interface we send and display information. The information is transported through the network.
2. Components
   1. GUI
      1. Widgets
      2. Other Supporting Classes
   2. Networking
      1. Internal Flow
      2. Breakdowns
      3. Intra-component Interaction Flow
         1. Interaction of signals and slots
3. Additional Info
   1. Signal/slot
   2. Threading
   3. Comnet
   4. Icons with ArcGIS/QT
4. Requirements
5. Understand how information is being passed
6. Understand how a command goes from GUI to network message
7. Understand how GUI updates from network message
8. Understand how we receive comnet messages
9. How to use C++
   1. Headers
   2. Pointers
   3. Usage in GCS
10. Qt issues
11. Comnet Issues (Install & Compile)
12. General Issues
13. Software
14. Additional Work

Changelog:

v0.5

Sections 3 & 4 initial are complete –Ethan Ito 6-15-2015

v0.7

Section 1 & 5 initial are complete

Section 2 Needs flow diagrams – Ethan Ito 6-20-2015

V0.8

Section 2 added flow diagrams

Section 4 added additional information regarding Qt and Comnet troubleshooting and GCS quirks as well as adding information about required software as well as current issues.

–Ethan Ito 6-20-2015 & 7-1-2015

V0.9 (working)

Example Description for signals/slots done for waypoint. - Ethan Ito 7-1-2015

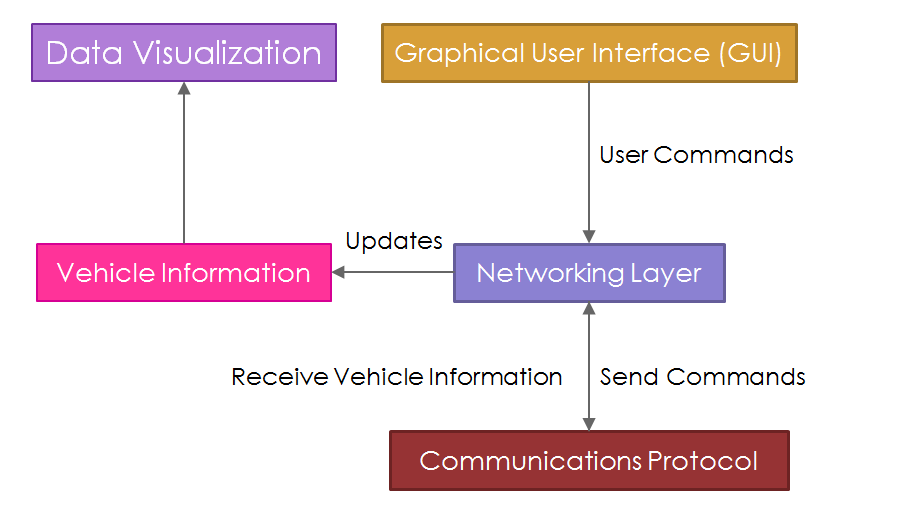
To do: Finish the remaining signals/slots

Added Additional Work, fixed some grammatical mistakes Ethan Ito 10-8-2015

**1. Concept**

The GCS’s purpose is to be able to display information related to vehicles as well as being able to send information to and receive information from vehicles.

The following diagram is a general view of how the GCS works.

**

* 1. **Interface**

The interface allows the user to send and view information related to vehicles.

* 1. **Network**

We receive information from the network protocol (comnet) and update the information being displayed. We then update our display from that received information. To send we do the opposite, from the interface we send the information through the network to the vehicles.

**2. Components**

There are 3 main components of the GCS. They are broken down into the GUI (mainwindow), network (network), and the thread that receives and sends information (rx\_thread).

**2.1 GUI**

Internally the mainwindow is broken down into several parts.

1. The declaration/instantiation of widgets.
2. Setting up the connections between the widgets and the network and between the widgets themselves.
3. The organization of widgets to be displayed.

**2.1.1 Widget List**

1. Sidebar (sidebar.h/sidebar.cpp)
   1. Side bar for map visibility
2. Waypoint (waypointgui.h/waypointgui.cpp)
   1. GUI to enter waypoint information
3. Vehicle Element Display (vehicleelementdisplay.h/ vehicleelementdisplay.cpp)
   1. Displays Vehicle Information
4. UGV\_State (ugv\_state.h/ugv\_state.cpp)
   1. Modifies the state of the UGV
   2. The ugv has behavior code already implemented that takes these commands into account, note there is signals/slots in networking and rx\_thread that will handle the sending of the messages.
5. Vehicle Info (vehicleinfo.h/vehicleinfo.cpp)
   1. Displays information related to the vehicle
6. Telemetry GUI
   1. Starts the telemetry between the given vehicle and the GCS
7. UAV Payload (uavpayload.h/uavpayload.cpp)
   1. Sends payload commands to the UAV (i.e. manually drop)
8. Toggle Button (togglebutton.h/togglebutton.cpp)
   1. A toggle button; used in sidebar
9. Targeting GUI (targetinggui.h/targetinggui.cpp)
   1. Manual entry of the target
   2. Has signals and slots to send in networking and rx\_thread
10. Serial Port Selection Window (SerialPortSelect.h/SerialPortSelect.cpp & SerialComboBox.h/ SerialComboBox.cpp)
    1. Drop Down menu to select com port
    2. This is used to select the attached XBee unit.
11. GCS Toolbar (gcs\_toolbar.h/gcs\_toolbar.cpp & Toolbar.qrc)
    1. Upper bar in the GCS
    2. Displayed but no actual function
12. Console Log (consolelog.h/consolelog.cpp)
    1. Displays text when it receives a signal.
    2. Used to log user commands and received information
13. Attitude (WidgetSix.h/WidgetSix.cpp, WidgetADI.h/ WidgetADI.cpp, qfi\_ADI.h/ qfi\_ADI.cpp, MainWIndowADI.h/MainWindowADI.cpp)
    1. Displays attitude from given information
    2. Need to double check the display
    3. May need to merge into one .h/.cpp

**2.1.1 Other Supporting Classes**

1. Internet Test (InternetTest.h/InternetTest.cpp)
   1. Tests to see if we have internet
   2. We use the result to determine whether or not we will use an internet or locally stored map for ArcGIS.
2. MapView (MapView.h/MapView.cpp)
   1. Class that holds all of the map layers for the GCS
3. GCSGraphicsLayer (GCSGraphicsLayer.h/GCSGraphicsLayer.cpp)
   1. This is a map layer that holds the waypoint information for the GCS
4. MapSymbol22
   1. This is the symbol that is used to display an item on the map.
5. Target (target.h/target.cpp)
   1. Unused class
   2. Holds information relating to the target
6. Targetlist (targetlist.h/targetlist.cpp)
   1. Unused class
   2. Holds target information
7. Vehicle22 (Vehicle22.h/Vehicle22.cpp)
   1. Vehicle class that is used to store information
   2. Created to be thread safe
8. Waypoint22 (Waypoint22.h/Waypoint22.cpp)
   1. Waypoint class used to store information regarding waypoints
   2. Created to be thread safe

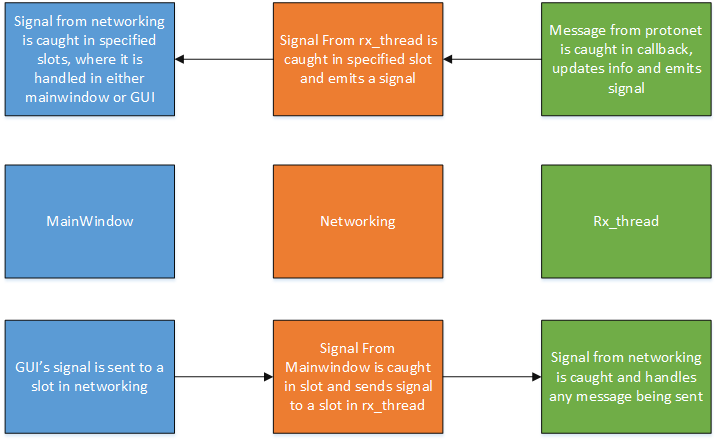
**2.2 Networking**

Internally broken into 2 parts, networking.cpp/.h and rx\_thread.cpp/.h

The networking class passes information between rx\_thread and the GUI. The networking class can be thought of as an abstraction of the rx\_thread. It starts the thread that acts as the sender/receiver for comnet messages. All messages being sent will be sent through the networking.cpp class as there is no class outside of networking that has access to the thread.

The networking thread is a thread, and it takes the appropriate measures as we are updating information outside of the thread. There are mutexes set up before and after each modification to stop any threading issues when updating the vehicle information. *Note that this has not been fully tested. Mutexes are in place, but there has been no check to see if under heavy load it is working properly.*

A general flow of how information is passed is as follows:



These are all of the signals and slots that are used with the respective .cpp that they are in and what class they are associated with. Note that GUI elements will be noted as they will internally handle the signal/slot. We will be using the name as well as what we are passing. An important part about the networking class is that it uses an internal signal to pass the information to rx\_thread. We cannot pass the information received in a slot in networking and pass that directly into rx\_thread. We must first create a new signal in networking, and then connect that to a slot in rx\_thread in order to pass the information. This signal passing may seem a bit odd, however as we invoke the thread in networking, we can only pass information to the thread from networking. The following is just an observation it can be skipped. *This may not be the optimal way of transferring information due to the amount of modifications necessary when adding/changing new signals to pass information.*

Fill out remaining signals/slots

**Waypoint**

*Class*

Signal : waypoint(Waypoint22\*,int)

*Mainwindow*

None, connects signal from waypoint class to slot in network

*Networking*

Slot: send\_waypoint(Waypoint22\*,int)

Signal: waypoint(Waypoint22\*,int)

*Rx\_thread*

Slot: send\_vehicle\_waypoint(Waypoint22\*, int)

**3. Additional Information**

**3.1 Signals and slots:**

In short signals and slots are Qt’s abstraction of callbacks. Signals emit, slots capture those emissions and the information that the signals contain. The slots can then manipulate the information emitted from the signals. In order to

**3.2 Links and Resources:**

Qt Documentation <http://doc.qt.io/qt-5/signalsandslots.html>

<http://en.wikipedia.org/wiki/Callback_%28computer_programming%29>

**3.3 Comnet:**

Concept:

Comnet handles the sending and receiving of information. This includes the various methods of sending (i.e. serial or udp), and how we are sending the information (packing of the information). It also unpacks the information. All we need to do is to determine how we handle comnet messages, where is comnet reading/sending the information from. The first is from callbacks (as declared in rx\_thread). The second is when we declare a new node (as declared in rx\_thread.cpp).

Details:

Comnet is a communications library. It handles everything related to sending and receiving messages. To give a brief overview it handles reading and parsing from both ip/port and serial. This means that you can test over localhost (127.0.0.1) on your computer without the need of the XBee units. The serial is for the XBee units. Comnet is able to take the information it receives and then break it down. The end result is a callback for the message. This is handled in rx\_thread.cpp. How callbacks work are that they are essentially function pointers. (See C++ for pointers). When comnet figures out what message it has received it calls a function. We can declare our own version of that function, all we need is the function name and formal parameters. These formal parameters hold various information; generally the id of the node that sent the message, and the information contained. Each “device” that uses comnet has a node id. This is not automatically declared, but set when we initialize the comnet node object. *We assume that we have will have unique id’s for each vehicle.* Inside of the method we can act upon the information, this includes updating vehicle information.

**3.4 Images with ArcGIS/QT**

Importing images into Qt is quite simple. Create a resource file in the resources folder in your project and then add the image to the resource. To access the resource file to add your images, right click the resource file and click “Show in editor”. This opens up a new window that will allow you to add image files.

WARNING: Be sure to Run QMake after adding/deleting resources so that the Qt Creator will recognize the new/removed images.

If the images have any kind of transparency, space that is not taken up and we will see the background, be sure to “flatten” the image in the editor. Please see the documentation for the editor.

Example: In GIMP go “Image”-> “Merge Visual Layers”. When exporting export with all settings. At this point Ethan has not determined what settings are the correct for the export of images.

**4. Requirements**

**4.1 Understand how information is being passed**

Files to look at:

mainwindow.cpp

mainwindow.h

rx\_thread.cpp

rx\_thread.h

networking.cpp

networking.h

vehicle\_list.cpp

vehicle\_list.h

Details: We pass information between our display and our network communications protocol (comnet) via two methods. The first being signals and slots. The second is the vehicle list object and the update signal it sends to signify that a vehicle has updated information.

**4.2 Understand how command goes from GUI to message**

Files to look at:

mainwindow.cpp

mainwindow.h

rx\_thread.cpp

rx\_thread.h

networking.cpp

networking.h

vehicle\_list.cpp

vehicle\_list.h

(Any GUI class .h & .cpp)

Concept: From our GUI we can emit signals that are passed to the thread. In the thread we take the values if any and then send the message.

**4.3 Understand how GUI updates from network message**

Files to look at: Same as 4.1

Concept: We receive the message from the thread, pass it through signals and slots to the mainwindow. In mainwindow, we update the information to display.

**4.4 Understand how we receive comnet messages**

Files to look at:

rx\_thread.h

rx\_thread.cpp

Detail: We use callbacks to handle the messages, matching the function name and formal parameters to handle comnet calls. (See comnet)

**4.5 Quick C++ Overview**

**4.5.1 Headers**

C++ uses headers, these files hold information. Most are used in conjunction with a .cpp file. In the GCS all of the .h files hold the function prototypes and the dependency declarations for the .cpp files. As C++ is compiled from the start of the file to the end, if you declare a function but use it earlier in the file, the compiler will throw an error. This is where function prototypes come in. They hold all reagent information that the compiler needs, the output, the function name and the formal parameters. Thus with all of the functions being declared in the header file, we can order the implementation of the functions in the .cpp in any order we like.

**4.5.2 Pointers**

To be put simply, pointers are a variable that holds an address in memory. What is stored at that address is determined by the type that the pointer is. i.e. a pointer to an integer would be declared as int \*j; J is a pointer, and points to an integer. To assign a “value” to j, we need an address of an integer. This can be done via the & symbol. This symbol when attached to a variable returns the value of the address that the variable is stored at. Thus if we have an integer o, the following j = &o would have j point to o. Note that any new instance of an object using the *new* keyword has to be set to a pointer to that object.

Function pointers are a special type of pointer as they are pointers that point to a function. These are used in callbacks for comnet. Notice that there is a \* symbol before the name of the method. This makes the name be a pointer for the method that is declared.

An important thing to note, if we want to have a value/object be able to be seen across the entire object’s functions, we need to declare the variable in the header file. This applies to pointers as well.

**4.5.3 Usage in GCS**

Most classes in the GCS use pointers and references. They are generally declared in the header file when we need to access the values in multiple functions. Understanding how pointers and references work will be very beneficial in understating how many parts of the GCS work.

More information regarding C++ can be found online:

<http://www.cplusplus.com/doc/tutorial/program_structure/>

<http://www.learncpp.com/cpp-tutorial/01-introduction-to-these-tutorials/>

**4.6 Qt Issues**

Most issues with Qt arise from code not updating properly when you are building. If you add/remove any file from the project, you must clean, run qmake then build the project. Cleaning the project removes any files that were created when compiling the code. Qmake makes sure that your project is up-to-date and ready to be built (uses your .pro file to accomplish this). If you are still running into issues, manually delete the contents of your build folder. This is located in your project settings.

**4.7 Comnet Issues (Install and compile)**

Most comnet issues come from compiler version, missing or non-updated files. Note that after any comnet change (you have changed the dll file), you must clean your project before your compile. Otherwise you may end up with the previous version of the library. As the GCS is using the VS2012 compiler, when there is a comnet update, you must get the vs2012 compiled version. You can do this by pulling the comnet code, and under the project settings for comnet change the compiler to VS2012. (To access this, right click the comnet project and go to properties. Note that you can just build the comnet project, not the entire Visual Studio project. Again right click and you can build from the menu that comes up.)

Another important thing is that you have the version of the dll that the GCS is being compiled in. If you want to compile in debug, be sure to have a debug compiled version of comnet. You can distinguish the versions by naming the debug with \_d at the end of the name. Note that you will have to make sure that the .pro file for the project has the correct name and location of the dll file.

If all else fails, run a “clean” comnet tester to make sure that everything is working properly. This can be done via a visual studio project (add project to repo). You will have to change the file locations that are being referenced in the project by going through the project settings; the way to access is the same way as accessing the compiler settings. This allows for an independent check to make sure that the protocol itself is working properly and that the GCS is not the issue.

**4.8 GCS Issues**

Currently the GCS is using the VS2012 compiler. We can update to the VS2013, however this requires a complete rework of the ArcGIS components of the project.

Environment : Qt 5.4.1 VS2013 compiler openGL, ArcGIS 10.2.5

The current status of that is that it is crashing at runtime. This is most likely due to the reference that was added to the point constructor. It is crashing on line 28 in mainwindow.cpp with the line that is setting the graphic for v46. Most likely it is that the mv’s spatial reference is not the correct reference to be using for the point.

The ArcGIS parts of the project dealing with the graphic display will need to be fixed. This means all waypoints, vehicle graphics, etc. will need to be updated accordingly. The methods are already in place, however as stated previously it needs to be updated in order for the program not to crash.

**4.9 Software**

The 2014-2015 GCS is using Qt vs2012. Other required software is ArcGIS 10.2.3. Getting this software installed will be an issue as ArcGIS no longer offers it as a downloadable. The final piece is a vs2012 compiler. There are various ways of getting the compiler, the easiest is installing vs2012 express. (Although there is a download available for the compiler only.)

**5. Additional Work**

(The following should be considered the opinion of Ethan Ito as what he thinks should be worked on if the current GCS were to be changed.)

There are a few aspects of the GCS that need to be changed. The first and the “easiest” would be the GIS software. Although ArcGIS gets a lot of work done, its major pitfall is the nature of the software. Not only can it change the API, it also removed the older version from being able to download. This makes it be a constant updating process if the installers are not saved. Additionally the installers are unique to each account thus copying the same installer may result in some unforeseen issues. An alternative would be Marble <https://marble.kde.org/>

<https://techbase.kde.org/Projects/Marble> (Steps to integrate). The only issue with marble is the time needed to get the compilation of the project to work. As the official documentation found is using a linux system. However it provides all of the abilities that ArcGIS does with drawing on the map. Another benefit would be the offline capabilities of being able to zoom in closer on the map than ArcGIS. The amount of work required to change it will be restricted to the classes that work with the display of the map.

The second aspect of the GCS that would take longer overall would be the revamping of the movement of information. As it is currently all commands and the vehicle update are passed individually, resulting in a large chuck of code reserved for the signal/slot connections. This also becomes an issue when updating and adding new elements. *If* redone this would mean a complete re-working of the GCS. There have been no tests done on when the GCS is under heavy strain to have the information for a comparative analysis if there was a change. Thus this change is optional. It is only listed due to the readability issues presented during the 2014-2015 year when new members were trying to get their code integrated.