Developer’s Guide to Comnet V2

Mario Garcia, Michael Wallace

This is a rough sketch of what is to come for Comnet V2, the next innovation to its predecessor.

* Introduction
* Design Goals
* Basic Implementation
  + Basic Architecture
  + Overall layout
  + Useful features
* Basic how to Use
  + Digest Excel, xml, sheets // IN-DEVELOPMENT
  + Useful commands for logger // IN DEVELOPMENT
  + Setting up connections
    - Communication Protocols
      * UDP
      * TCP // IN-DEVELOPMENT
      * Serial
      * XBee // NEEDS-TESTING
        + WPAN
        + ZigBee
        + Digimesh
    - Set up port
    - Set up address
  + Sending packets
  + Receiving packets
    - Callback handling
      * Return results
  + Error Handling
* Intermediate Communications
  + Loose Coupling Procedures
  + Developing manual message handling
  + Extending Abstract Packet
  + Extending Abstract Queue
  + Linking Everything to one
  + Handling “Orphan” packages
* Advanced Techniques IN DEVELOPMENT
  + Improving overall performance
  + Advanced Loose Coupling features
    - Declaring custom allocators
      * Extending Allocator
    - Declaring custom data structures
      * Extending data structure interfaces
    - Tuning the Comms engine
    - Advanced Commands for Console log
    - Security Handling,
    - Custom Comparing
    - Custom Serialization
    - APIs for Communication handling
      * Specific sets for each protocol.
      * Defining your own protocols.
  + Future Works
    - Further Polishing
    - Compressing Packets
    - Testing Toolset
    - Publish-Subscribe Model
    - To Be Continued…
  + Conclusion
  + Index
  + Notes

Introduction

Before November of 2015, there was a need to provide a protocol layer for network communications using Wireless, UDP, TCP, and raw serial. By NATO STANAG 5602, there shall be a communications modules intended on IP based protocols, however datalink abstraction plays a crucial role in networks between various clients and users. For this, the creation of Protonet was important, in that it complied with most of NATO standardization of military protocols. Much was contributed in the work of Protonet Library, a Link abstractor and protocol parser were implanted to provide dynamic and rigorous transmission, along with the help of easily providing custom made packets without the need to redefine the entire packet.

Although the Protonet library has their merit, that is not to say that it went without problems, in fact there were multiple flaws to the design of the product that deemed it difficult, and clunky to use. For starters, the design of the library was not built to adapt to various features, in fact it was built too rigid to provide enough room for change. Of course, rigid design is indeed a definite trait; it does not work well when it comes to maintainability. For this, the library itself was not written with programmer-friendly features. The most notable features of this consequence comes deeply from how its API (Application Programming Interface) was designed, in that it required external scripts to provide updating of the library. The use of python scripts enabled developers to submit requests, so that they may be compiled into the library. The problem with this scheme is that it would take days, if not weeks, to recompile and redistribute the library with a new version. Many among these features can resemble to the early days of OpenGL, which gave all the work to the GPU kernel, instead of giving users and developers the freedom to define their own custom implementation, all while supporting and adapting the kernel to newer versions and updates. No more, however, a new library was sorely needed.

There were various needs to satisfy developers with easily integrating the library into their applications, all the while giving the library the ability to apply custom made packets. Comnet nodes which supply the endpoints between communications with others were in need of proper debugging and logging of performance. There was a deep need to provide a solid foundation in not only this, but also the adaptability of a product that can be maintainable, as well as rigid, for use in the long run.

Design Goals

CommProtocol (Comnet-V2) was specifically designed to meet these needs, along with giving the users and developers the freedom to both easily defined packets, dynamically, along with giving full modular control for extensive use. The library itself is capable of allowing users to simply declare packets without the need to submit requests to integrate them into the library. This is done through loose coupling, allowing users to inherit abstraction with their packets, and giving them the control on how it should be handled upon transmission and receive. Not only is this, but the library is itself fully customizable to allow creating new forms of nodes to handle any developer’s needs and desires.

The handling and design of the library was solely made possible through the use of object orientation and software engineering practices, coupled with research in data science. The communications pipeline undergoes rather advanced processes, yet they are handled through the use of simple tight coupling features that allow for greater performance through the use of internal structures. More on this later.

Basic How to Use

Visual Studio 2013 update

*Tools > Extensions & Updates > Updates > Product Updates*

**How to make a packet:**

All packets must inherit AbstractPacket class which is located in the file called “abstractpacket.h”. AbstractPacket has three abstract function which must be overwritten which are void **Pack**(ObjectStream& obj), void **Unpack**(ObjectStream& obj), and AbstractPacket\* Create(). The functions **Pack** and **Unpack** must use the ObjectStream class to serialize and de-serialize data. ObjectStream is a stack therefore **Pack** and **Unpack** input (push) and output (pop) must use data in reverse order. ObjectStream overloads the stream operators << for input (push) and >> for output (pop). **Create**() method should return a new dynamic object of its own class by calling its own constructor. This should be deleted in the linked call back which is explained later. The base constructor which is the AbstractPacket class should take in the argument of the derived class’s name as a string which gives the packet a unique id.

**Typedef and Defines**

// Small defines for inheriting C++ AbstractPacket

#define CHAIN\_ABSPACKET(class\_name) comnet::AbstractPacket(#class\_name)

#define INHERITS\_ABSPACKET public comnet::AbstractPacket

#define ABSPACKET comnet::AbstractPacket

typedef comnet::AbstractPacket ABSPacket;

**Example**

#include <CommProto/commproto.h>

//Can be struct of class

struct VehicleAuthorizationRequest : INHERITS\_ABSPACKET {

//Constructor

 VehicleAuthorizationRequest(uint16\_t vehicle\_id = 0,

   uint8\_t vehicle\_type = 0,

   uint8\_t authorized\_services = 0,

   uint8\_t granted\_services = 0)

   : CHAIN\_ABSPACKET(VehicleAuthorizationRequest) //passing class name

   , vehicle\_id(vehicle\_id)

   , vehicle\_type(vehicle\_type)

   , authorized\_services(authorized\_services)

   , granted\_services(granted\_services)

   { }

 /\*\*

   Pack data into the stream for sending out.

 \*/

 void **Pack(**comnet::ObjectStream &obj) override {

   obj << vehicle\_id;

   obj << vehicle\_type;

   obj << authorized\_services;

   obj << granted\_services;

 }

 /\*\*

   Unpack data back into this packet when receiving data.

 \*/

 void **Unpack**(comnet::ObjectStream &obj) override {

   obj >> granted\_services;

   obj >> authorized\_services;

   obj >> vehicle\_type;

   obj >> vehicle\_id;

 }

 /\*\*

 Tells CommProtocol how to recreate the

 VehicleAuthorizationRequest packet

 when receiving data.

 \*/

 comnet::AbstractPacket \***Create**() override {

   return new VehicleAuthorizationRequest();

 }

 /\*\*

   Data.

 \*/

 uint16\_t  vehicle\_id;

 uint8\_t   vehicle\_type;

 uint8\_t   authorized\_services;

 uint8\_t   granted\_services;

};//End struct

**How to create and link callbacks:**

Creating callbacks

Callbacks are used to execute code once a message is received. Callbacks can check who sent the message and determine what actions to take. Each message type uses one call back which is linked to a communication node which is explained later.

Callbacks have the following signature:

error\_t *callBackName*(const comnet::Header& header, const VehicleAuthorizationRequest & packet, comnet::Comms& node)

//Header: useable data such as who the message is from

//VehicleAuthorizationRequest: derived class from AbstractPacket

//Comms: main communication object which can send new messages

When returning from callbacks you can return codes to let the state machine know what to do with the packet.

// Callback codes used for state machines in automating what may be

// done to the packets. Can pass more than one code via OR bitwise.

enum CallbackCodes {

 CALLBACK\_SUCCESS        = 0x1,

 CALLBACK\_FAIL           = 0x2,

 CALLBACK\_DESTROY\_PACKET = 0x4

};

For example on a successful call back you can return success and delete the dynamic message. Failed message will alert the state machine an error has occurred. **Successful packets should be destroyed.**

return comnet::CALLBACK\_SUCCESS | comnet::CALLBACK\_DESTROY\_PACKET;

Unlinked or non-destroyed packets can be manually retrieved through Comms object method AbstractPacket\* **Receive**(uint8\_t&  source\_id) which is explained in *CommNode documentation* part.

**Typedef**

typedef int32\_t error\_t;

**Example**

// Callback function that we will be using to link to Ping packet.

error\_t VehicleAuthorizationRequestCallback(

const comnet::Header &header, VehicleAuthorizationRequest &packet, comnet::Comms &node)

{

 std::cout << "=::RECEIVED PACKET::=" << std::endl;

 std::cout << std::endl << "Source node: " <<

 (int32\_t)header.source\_id << std::endl;

 std::cout << "Message: " << std::endl;

 std::cout << packet.vehicle\_id << std::endl;

 std::cout << packet.vehicle\_type << std::endl;

 std::cout << packet.authorized\_services << std::endl;

 std::cout << packet.Granted\_services << std::endl;

 return comnet::CALLBACK\_SUCCESS | comnet::CALLBACK\_DESTROY\_PACKET;

}

**Linking callbacks**

Linking callbacks will automatically execute the callback when the message is received.

To link a call back to a comms node you should already have the comms node created and preferably but not needed initialized. Assuming the created node is calls comm1 you can call its method **LinkCallBack** which takes two arguments which are AbstractPacket\* and Callback\*. To use the call back we created we must allocate a new Callback\* which takes in a function pointer. Our callback we created needs to be cast to callback\_t to be a function pointer.

**Typedef**

typedef error\_t (\*callback\_t)(const Header&, AbstractPacket&, CommNode&);

typedef std::function < error\_t (const Header&, AbstractPacket&, CommNode&) > CallbackFunc;

**Example**

comm1.**LinkCallback**(new VehicleAuthorizationRequest(),

new comnet::Callback((comnet::callback\_t) VehicleAuthorizationRequestCallback));

If no call back is inserted for the second argument of **LinkCallback** the packet will be sent to the receive queue and must be handled by the user manually. This first argument is for associating the packet type with the linking process and does not need valid arguments in the constructor.

**How to create and initialize a node:**

The only header file you need to use CommProtocol is commproto.h which is inside Commproto folder.

#include <CommProto/commproto.h>

CommNode and Comms is located inside comnet namespace. Comms derives CommNode as its base class. Comms constructor takes in one argument which is the node number to identify different platforms.

**Example**

// CommNode 1

comnet::Comms comm1(1);

When using encryption you must load the key into Comms which can be done from **LoadKey**(char\* key) or **LoadKeyFromFile**(char\* keyFileName). It is recommended not to hard code the encryption key therefore you should use **LoadKeyFromFile.**

**Example**

comm1.**LoadKey**("NGCP project 2016");

comm1.**LoadKeyFromFile**("key.txt");

When you initialize a node you choose what type of connection you are using. The types of connections are UDP\_LINK, SERIAL\_LINK, and ZIGBEE\_LINK. Depending on what connection type you must also supply argument. The initialization method has the following signature.

**InitConnection**(transport\_protocol\_t conn\_type,

                          const char\* port,

                          const char\* address = NULL,

                          uint32\_t baudrate = 0)

**Example UDP**

// 1338 is the port data is received on and 127.0.0.1 the nodes own IP

comm2.**InitConnection**(UDP\_LINK, "1338", "127.0.0.1")

**Example Serial**

// COM5 is the Microsoft port and baudrate is 9600

// Linux comport using FTDI is "/dev/ttyUSB0"

// Serial does not use an address hence NULL is used

comm1.**InitConnection**(SERIAL\_LINK, "COM5", NULL, 9600)

**Example Zigbee**

// COM3 is the Microsoft port and baudrate is 57600

// Linux comport using FTDI is "/dev/ttyUSB0"

// Zigbee uses a MAC address which does not need to be inputed

comm1.**InitConnection**(ZIGBEE\_LINK, "COM3", "", 57600)

**How to add connections to a node:**

Adding connections to your node allows you to communicate to other platforms easily. To add a connection to your initialized node, you must call the **AddConnection()** function from your node:

/\*\* The address entered will be paired for communication by destination ID

Adding address can be a UDP IPV4 or hex MAC address for zigbee

Adding an address is not need for serial and will default to ""\*/

bool **AddAddress**(uint8\_t dest\_id, const char\* address = NULL, uint16\_t port = 0) override;

**Example UDP**

/\* Assuming a UDP connection, you must require the port and address, along with the unique id, of whoever you are trying to connect to \*/

comm1.**AddAddress**(2, “127.0.0.1”, 1337);

**Example Serial**

/\* Assuming a serial connection, there is no need for adding connections, serial acts as a broadcast. \*/

**Example Zigbee**

/\* Make sure you know the MAC address of the xbee you wish to communicate to, along with the id of the node you are trying to connect to. \*/

comm1.**AddAddress**(2, “0013A2004067E4AE”);

The dest\_id is the unique id that is given for all platforms. It is a unique identification number solely identifying the vehicle to connect to. You will be using this number for sending, as we will discuss later.

**Running a Node:**

Running a node is simple, as well as VERY important. There are threads within the node that do not run unless stated so by the user. This requires that node call the Run() function:

comm1.**Run**();

This will essentially tell the node to start working, and running the receive and send handlers within. Without this call, the node will not process any information for sending or receiving.

**Sending From a Node:**

What’s important about these CommProtocol nodes is that they are used as the backbone, or gateway, for sending and receiving data universally, regardless of the protocol. Of course, we need to be able to send information easily, and without worrying about the format of the data that is to be sent. Luckily, CommProtocol handles this much with it’s AbstractPacket inheritence, so all we need to do is send the information. This is done so with the **Send**() function:

/\* Sends the specified packet to the receiving dest\_id.  \*/

bool Send(AbstractPacket& packet, uint8\_t dest\_id) override;

**Example**

/\* Create the packet you wish to send to, and fill it with data to send.    \*/

VehicleAuthorizationRequest request;

/\* Send the request out to destination id 2, which is the id we connected to, using AddConnection() \*/

comm1.**Send**(request, 2);

This function call is abstract, so it is not specific to any protocol.