Generic Concept & Project Philosophy

By: Anthony Sasso

December 14, 2022

Contents

Introduction

Overall there are a few goals with this project. First is to enable me to use some design pattern, networking, security courses to develop a project I would likely use in the future. Another is due to how annoying/disparate networking can be with C/C++ having a preset skeleton to then build on could significantly reduce the overhead. This would also hopefully be without requiring added complexities from things like Boost, QT, etc. which many not want as dependencies in their projects.

There are also some requirements listed before the development of the official requirement matrix, currently nothing present here will limit future developments unless implicitly implemented within such a requirement document as needed.

General API Concept & Pseudo Requirements

The API as I have imagined it will consist of a few principles to be used for both the style & architecture of the project.

1. **Packet API General structure.**

   The packet API will be a template Packet with sub-templated datatype for each element to assist in its creation & use. The packet itself will consist of no more (but optionally less in the form of allocating a nullified pointer to *"TailData" or "CheckValue"* when needed) than the elements: *"HeaderData"*, *"DataBuffer"*, *"TailData"*, *"CheckValue"*, and *"SizeLimit"*. The final value of SizeLimit should also not be allocated manually instead using a predefined setter within the template aggregating a getMaxSize() function from each of the mentioned elements into a constant unsigned value.

2. **HeaderData & Struct Linking.**

   A separate .h/.hpp file including the struct contained within *"DataBuffer"* should be expected to exist on each side of the network in the form of a file named *"DataBufferStructures.h/.hpp"* which will define the DataBuffer overloads for each struct. Optionally this file may also exist as a linker for other, sub headers in the form of *"StructName_DataBuffer.h/.hpp"* and then pointed to using *"DataBufferStructures"* as a reference of sorts.

   The API should expect a HeaderData class of the minimum elements:

   (a) *sendIP*: This element will denote the sender IP as a 32-bit unsigned integer.

   (b) *recvIP*: This element will denote the receiver IP as a 32-bit unsigned integer.

   (c) *commPort*: This element will denote the operating port for the packer for checking connection authenticity.

   (d) *dataSize*: This element will denote the size of the DataBuffer present in this packet.

   (e) *tailSize*: This element will denote the size of the TailData present in this packet.

   (f) *checkValSize*: This element will denote the size of the CheckValue present in this packet.

   (g) *dataPos*: This element will denote the data position of this packet in the case of multi-packet transfers, otherwise set to 0 to denote a control packet/singular transfer.

   (h) *flagBArrSize*: This element will denote the size in bytes of the following flag binary array (for use of bit masked flag allocation), otherwise set to 0 should no flags be present.

   (i) *flagBArr*: This element will denote the binary array for all flags within the header, otherwise set to NULL along with flagBArrSize set to zero should no flags be present.

   In addition to the mentioned elements, it may be warranted to implement functions callable by the header class. The basic options would be a Constructor with a buffer/struct & individual element overloads. The header may also the use of a "check compatibility" function which checks the flagBArr element and returns an enumeration array for each value, but it may instead be simpler to use a *"checkFlag(boolean* values, int numOfElem)"* function the is then filled with the flagBArr data (the first value in the boolean array corresponding to the most significant bit in the binary array).

3. **DataBuffer Element.**

   The API should expect itemized buffers in the form of a class template of type *"DataBuffer"* which implementations can input a struct into, and overload four main functions. These functions will be *"Serialize"*, *"Deserialize"*, *"DataBuffer(struct, segmentSizes [])"* (segmentSizes will be an unsigned array, denoting the size of each structure element in bytes), and *"DataBuffer(individual elem, . . .)"*; the final two of which will be intentionally made redundant to allow both raw and buffer allocation for the data field.

4. **TailData & CheckValue Elements.**

   The Elements of *"TailData" and "CheckValue"* are optional but a CRC will be implemented within the template to avoid re-implementation. By Default *CheckValue* will operate as an automatic 64bit CRC with overloaded implementations for hash, or data-repair with *TailData*.

Overall it may be warranted to embed the CheckValue class within TailData as it does cover similar functionality. This will be decided during the requirements phase as it could still be warranted to have them separate to reduce each classes scope.

NetworkManager Controller

In addition to the previous mentioned Packet, there will be a controller to create sockets & connections in detached threads, generate & send packets, and close either specific connections, sockets, or all connections.

The general functionality and elements will likely be as follows:

1. **Elements**

   The elements within this class will be as follows:

   (a) Sockets: a vector of SocketData pointers controlled by either NetworkManager or a detached thread (one/Socket).

   (b) numOfElem: an unsigned int for the number of current sockets.

   (c)

2. **Constructor**

   The Constructor will likely have only three implementations with some mandatory and some nonmandatory elements (in the form of an overloaded Constructor). These will be as follows:

   (a) A basic constructor requiring a Port, max connection limit (or zero for no limit)*, boolean flag for either [detached multithreaded/sequential balanced] socket handling. *if >1 connections it will use the Shared Port functionality with a count of the number of active connections within that sockets class.

   (b) A constructor with an input of SocketData, which will be a struct containing the same previously declared data (this will also be created from the previous constructor when stored within the NetworkManager).

   (c) A more complex constructor of an array for SocketData for systems with multiple open sockets each of which have different settings.

3.

