Generic Concept & Project Philosophy

By: Anthony Sasso

February 21, 2023

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.

   * The maximum size of a given packet will be set to 64Kib with any sized above automatically triggering MPT (multi-packet transfer). This max will be over-loadable within the packet object constructor if warranted.

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) *netFmt*: This element will denote if the IP is IPv4 (32-bit), or IPv6 (64-bit) address for the following IPs as an enumeration of [IPv4 | IPv6].

   (b) *sendIP*: This element will denote the sender IP as a sockaddr_storage datatype.

   (c) *recvIP*: This element will denote the receiver IP as a sockaddr_storage datatype.

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

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

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

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

   (h) *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.

   (i) *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.

   (j) *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. This object will use a 'bitset<flagBArrSize>' object cast to unsigned long during serializations, giving a max number of 32 flags.

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, of *"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* (note a template FEC using the leopard library will optionally be implemented as well).

   The CheckValue object will trigger CRC, FEC, or other over-loaded functionality through three functions within a DataHandler Class contained within TailData, these will be as follows:

   (a) *Start*: This function will initialize the data handler and will be used for some FEC APIs or as a backup constructor for APIs without the need for this function.

   (b) *Encode*: This function will wrap and implement the encoding for sending data to another device.

   (c) *Decode*: This function will wrap and implement the decoding for sending data to another device.

   The TailData class will also contain and serialize all necessary information for the functionality of this DataHandler, as required by its implementation. In the case of inline FEC, there CRC could be NULL & TailData is utilized as a caller without any data in it.