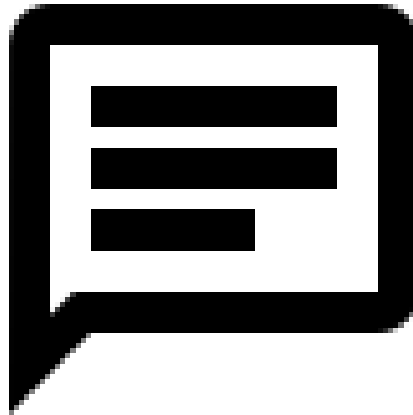


Chat Protocol v1.1

Moving Forward to v2.0



Prepared By:

Jordan Godau

Prepared For:

BCIT Data Communications Cohort

1.0 Introduction

This document will be a brief and informal overview of the protocol thus far. This includes what went well and what didn't, what features we expect to change, and what new support will be introduced moving forward. Additionally, some justifications, explanations, and examples for certain elements may be provided in order for everyone to understand *exactly* why some things (in honor of colloquialism) "*are the way they are*".

My partner and I have taken the time to implement the API into a chat application in its entirety. This was done to validate the API and protocol ahead of time as to avoid any potential strategic mistakes in the design. Following this implementation, we believe the protocol is a strong, logical, and practical approach to transmitting chat messages over a network. Nevertheless, there are certain areas that present flaws that pose a major risk to client-server interoperability.

I will attempt to outline the necessary changes and new features as best as possible, so that everyone knows what to expect moving forward. Furthermore, [this document](#) has been created so that any suggestions may be provided completely anonymously.

Thanks everybody.

2.0 Voice Chat

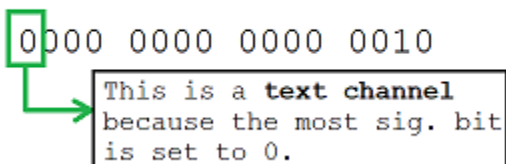
The introduction of voice chat support provides the most intimidating challenge for those implementing the next version of the protocol. At face value, this is by far the biggest identifiable change **CPT 2.0**. Nevertheless, the implementation of this *should* be as simple as the minor support provided for it (which is already written into [Protocol v2.0](#)). While it should be mentioned that the current support is only a theory of what will be necessary for implementation, we do not expect any hiccups.

The currently implemented version 1.1 maintains a 16-bit CHANNEL_ID field, enabling channel IDs between 1-65535 (because channel ID 0 is reserved for the **Global Channel**). For The Protocol's designed purpose, this number is excessive. Thus, the most significant bit (that is, 2^{15}) is being **repurposed as a voice chat flag**. Specifically, this bit will be set on or off (1 or 0) to indicate whether the channel is a *voice channel* or a *text channel*, respectively.

This means that when using version 2.0, user-created text channel IDs will ***always*** range between 1-32767, and voice channel IDs will ***always*** range between 32768-65535. To clearly demonstrate what we mean by this:

Channel ID 2 (16-bit) (Text):

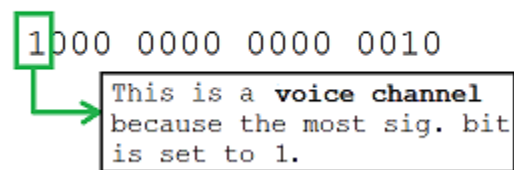
0000 0000 0000 0010



This is a **text channel** because the most sig. bit is set to 0.

Channel ID 32770 (16-bit) (Voice)

1000 0000 0000 0010



This is a **voice channel** because the most sig. bit is set to 1.

By distinguishing the channel types via the example above, this provides both a logical, and numerical distinction between the two types. Our belief is that this offers the greatest amount of control with the lowest overhead changes in the code implementation.

The justification for this was made based on two key pieces of information:

1. Implementers should be able to write code for v2.0 as an *extension* of v1.1.
2. Significant restrictions currently exist for voice chat implementation.

Currently, prevailing knowledge suggests that **only linux systems** will be able to utilize voice chat through the use of [arecord](#). Since the CPT protocol is de-coupled with the actual transmission of data over a network, the actual *sending* of voice data should happen via some transport protocol (preferably UDP).

To finally conclude what this means for our “most significant bit” design implementation, the presence of this bit (i.e. this bit field is set to 1) should tell a user-facing client program in human terms:

“Open a UDP connection, turn on my mic, and execute arecord, I’m about to start talking.”

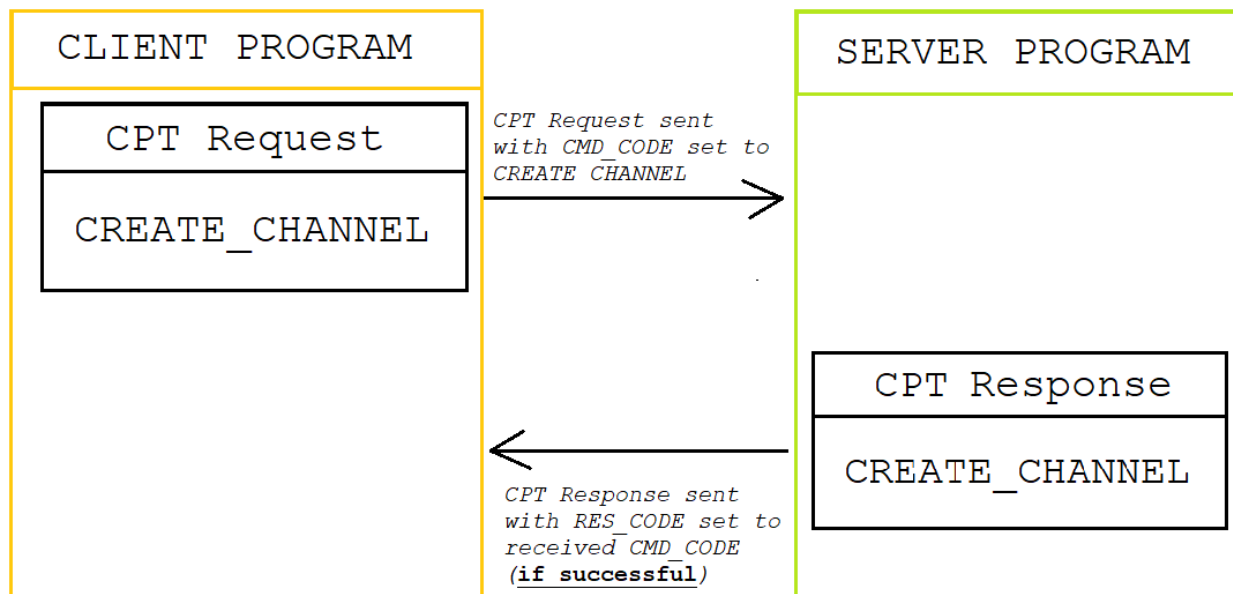
3.0 Strict Response Code Enforcement

Of all our findings during a thorough implementation of the Protocol and API, this is viewed by myself as the largest oversight by a landslide. Currently, the Protocol maintains a very weak enforcement of response codes, with little to no guidance on it whatsoever.

This inevitably leads to issues during implementation - primarily:

1. Interoperability between different client programs is significantly compromised without a standardized definition of how to interpret a server response
2. If implementing threading or multiprocessing, a message may be intercepted in the wrong part of the program control flow. With no way to determine what type of response you are receiving, this causes unexpected behavior in the implementation.

As a solution, a successful response to any CPT request message shall contain an identical CMD code to the request sent. To demonstrate below:



The above implementation will help in distinguishing between not only successful server responses, but the type of response you are getting - a missing element in Protocol v1.1.

A failed response should include one of the appropriate response RESPONSE ERROR CODES provided in either the API or according to the protocol documentation.

Protocol v2.0 currently does not have these constraints formally documented, however implementers should expect to see this soon.

4.0 Message Response Sub-packet

One of the most common questions I received after the first implementation was about the “Msg Response Sub-packet”. Prevailing wisdom would conclude that this particular section of the document confused people. This comes as no surprise, as this was a loosely generated idea that was included in the documentation hastily (then removed, then re-included).

At this time, I have gone back-and-forth on whether or not something like this is necessary. Above all else, we must always circle back to asking ourselves “What *value* does this constraint provide, if any?”. Still this begs the question: how does one *know* who a message is coming from when it is received, with no user id in the existing response packet.

My belief still stands that messaging should be handled within the Response Packet, while the actual *formatting* of the sent message should be handled by the user-facing application. That is, if an implementer wants *their* implementation to send their name and any other information as part of their message, they may do this. However, the absence of this additional information has no effect on the compatibility between different client elements between a server program, since these messages will almost always be immediately sent to some file stream.

Ultimately, this must be discussed with the entire CPT protocol team (something that I have, as of writing this document, not discussed with anyone). Nonetheless, it is important for anyone who will be, or has already implemented this sub-packet to know that this feature may change in both the API and the Protocol.

As it stands, the current idea is simply to present a *suggestion* on what kind of information should be sent, rather than a prescriptive sub-packet.

5.0 Conclusion

Overall, we believe the initial design to be a success. Of course, this is something that none of our team members has ever tried before. Thus, there were issues and oversights, and this is to be expected.

Nonetheless, it was fun and exciting to hear peers saying “Cpt Request Packet”, and “Global Channel”, knowing how hard we all worked to make this protocol and API truly flexible, simple, and practical. Many hours and legitimate sleepless nights were spent fine tuning the protocol and API, and we believe this final version will encapsulate the last missing elements to be truly agnostic between different client implementations.

With that being said, a summary of the expected upcoming changes are presented below:

- **Voice Channel Support**
 - **Definitely happening**
 - Should not require anyone to change code they have already written, but definitely require them to add more features
- **Stricter Response Code Enforcement**
 - **Definitely happening**
 - Successful response must have a RES_CODE that matches the CPT Request CMD_CODE field.
 - Unsuccessful response codes should contain an appropriate error code.
- **Re-work of the CPT Response Sub-packet**
 - **Probably happening**
 - Present as a suggestion rather than a prescription
 - Enable implementers to send whatever they like in their chat messages.

Thank you everybody, please let us know if you have any questions or concerns moving forward.