Network Working Group
Request for Comments: 9898

Scott Rubey
November 2020

Chat Room Instant Messaging Protocol

Status of This Memorandum:

The purpose of this memorandum is to define a new protocol for the Internet community. Suggestions are expected and welcome. Please note, this protocol is purely experimental, being for educational purposes, and may remain incomplete. Distribution is unlimited.

Abstract:

The CRIMP is an internet protocol developed for the purpose of supporting single-server to many-client interaction in a multi-user chat room environment.

Table of Contents

1. INTRODUCTION

The CRIMP (Chat Room Instant Messaging Protocol) is an experimental protocol used for sending messages between server and clients that manage multi-user chat rooms. (Such client-originated messages shall hereafter be referred to within this document as “commands” in order to differentiate from “messages” created by the user for posting to a chat room.) The protocol supports creating one or more chat rooms, joining chat rooms of the users’ choice, and posting messages to one or more chat rooms, along with other assorted utility functionality (i.e. connecting/disconnecting from the server, leaving a chat room, etc.). The following document describes the CRIMP as it currently stands.

1.1 Server

It should be emphasized that this protocol only supports the use of a single server at this time. The server stands as a central node through which numerous users may communicate with each other in the context of a chat room. It processes protocol messages received from each client, provides the intended functionality, and responds to individual requests as need be.

1.2 Clients

A client, in this context, may be viewed as synonymous with a “user.” Any node in the network that is not the server is a client. Each user must identify themselves to the server (and thus to a chat room) by a String username, chosen by the user upon program initialization.

1.3 Rooms

A “room” is the virtual space in which users are allowed to communicate with each other. When a user begins a session, they are given the option of joining an existing room or creating a new one. A new room is named by the user who creates it. The room will persist with its given name even when no users currently occupy it, as long as the server remains running; there is currently no means for persistence in the event of a server crash. The room’s name serves as its unique ID, and cannot be changed. Within a room, users may post messages that are viewable by all other users within that room. Users may join or leave an existing room as they please.

2. THE CRIMP SPECIFICATION

2.1 Overview

The protocol detailed in the following pages deals specifically with client-to-server communications and server-to-client responses. As of this writing, security measures beyond simple error handling are not addressed, though they may be in future releases.

2.2 Character Codes

Commands are passed between the client and server as 4-letter character codes in the fashion described in Section 2.3, below.

2.3 Commands

Commands are passed as Strings and generally contain three parts (fields), though they may conclude with an optional message from the user (see Section 3.6). The first field contains the username associated with a particular client, which is chosen by the user each time they run the program. The second field of the String contains the 4-letter command code. The third field, if present, contains the room name the command is to be applied to. While this field is generally present, there may be situations in which a room name is irrelevant and should be omitted (see Section 3.2) Finally, an optional fourth field contains any messages the user wishes posted to a chatroom. All fields in the command string are delimited by a single space (“ “). Command codes and messages are covered in further detail in Section 3.

2.4 Responses

The server responds to all commands with one of three Strings: 1) the information requested (such a list of available chat rooms); 2) an “OK” message; or 3) an appropriate “ERR” message. How the user-side application deals with these messages is at the

discretion of the developer. “ERR” messages are specific to the command being executed; examples can be found in Section 3, below.

3. COMMAND CODES AND MESSAGE DETAILS

3.1 Create Chat Room

Command format: String(username, “CTRM”, roomName)
Server replies: OK_CTRM, ERR_DUPLICATEROOM

When the user opts to create a new chat room, a command is sent from the client to the server using the “CTRM” command code, along with the new room name, as input by the user. Since rooms must carry unique names, server implementations should determine whether or not a room by the chosen name already exists. If not, the server responds with a “OK_CTRM” message upon creating the new room. If a room by that name does already exist, the server-side function should not create the new room and should respond with an “ERR_DUPLICATEROOM” message.

3.2 List Chat Rooms

Command format: String(username, “LSRM”)
Server replies: list of rooms, ERR_NOROOMS

When the user wishes to list available chat rooms, the client sends a string containing the “LSRM” command code, and an empty message body (i.e. the empty string). The server either responds with a list of rooms in String form, or an “ERR_NOROOMS” message in the event no rooms currently exist. (For all but the very first user, said error message should never be sent.)

3.3 Join Chat Room

Command format: String(username, “JOIN”, roomName)
Server replies: OK_JOIN, ERR_NONEXISTENTROOM, ERR_ALREADYJOINED

If a user would like to join a chat room, the “JOIN” command code is sent to the server along with the name of the chat room they would like to join. Server responses include “OK_JOIN” in the event the user is successfully added to the room, or “ERR_NONEXISTENTROOM” if the user has provided a room name that does not exist on the server’s room list. In order to ensure the same user does not join the same room multiple times, which can lead to debugging issues depending upon the implementation, the server also provides the “ERR_ALREADYJOINED” message.

3.4 Leave Chat Room

Command format: String(username, “LEAV”, roomname)
Server replies: OK_LEAV, ERR_NOTINROOM, ERR_NONEXISTENTROOM

Should the user decide to leave a particular chat room, a command is sent to the server that contains the “OK_LEAV” command code, along with the name of the room they would like to leave. If a room of that name exists and the user is currently on its active users list, the server removes the user from that room and responds with an “OK” message. In the event the server cannot find a room of that name associated with the user, the server responds with “ERR_NOTINROOM”; in the event that room does not exist at all, the server responds with “ERR_NONEXISTENTROOM”.

3.5 List Members in a Chat Room

Command format: String(username, “LSMB”, roomName)
Server replies: list of members, ERR_ROOMEMPTY, ERR_NONEXISTENTROOM

Should the user wish to see a list of members who are currently in a particular chat room (thus, exist within the server’s list of users for that chat room), a command is issued to the server containing the “LSMB” command code, along with the room name for which they would like members listed. The server must then check its list of active rooms. If a room by the provided name exists and there are users currently accessing the room, the server responds with a String object containing the list of members in that room. If the room exists but has no current members, the server responds with “Room Empty.” Lastly, if a room by that name does not exist on the server’s list, the server returns “ERR_NONEXISTENTROOM” to the client.

3.6 Post Message

Command format: String(username, “POST”, roomname, message)
Server replies: OK_POST, ERR_NOTINROOM, ERR_NONEXISTENTROOM

Should a user wish to post a message to a chatroom they currently occupy, the client sends a “POST” command to the server, along with the roomname, as entered by the user. The message composed by the user is included as the fourth argument to this command. While this protocol does not limit the size of user messages, it is recommended that some limit be implemented by either the client- or server-side software. Presuming the user is present on the server’s list of active users for the given room, an “OK_POST” message is returned to the client. If the room exists but the user is not a current member, the server returns “ERR_NOTINROOM”. Finally, if a room by

the given name does not exist on the server's list of rooms, the server returns "ERR_NONEXISTENTROOM".

3.7 Retrieve Messages

Command format: String(username, "RETV", roomname)

Server replies: message text, ERR_NOTINROOM, ERR_NOMESSAGES, ERR_NONEXISTENTROOM

Should a user wish to display messages for a room they currently occupy, a Retrieve command is sent to the server containing the "RETV" command code and the roomname, as input by the user. If a room by the provided name exists and the user is a current member of that room, the server responds with a String object containing all messages attributed to that room. If the room exists but the user is not a current member, the server returns "ERR_NOTINROOM". If the room exists but no messages have been posted to that room, the server returns "ERR_NOMESSAGES". Finally, if a room by the given name does not exist on the server's list of rooms, the server returns "ERR_NONEXISTENTROOM".

This command is provided as a utility for implementations that do not include a user interface, such as those run from the command line.

3.8 Disconnect from Server

Command format: String(username, "DSCT")

If the user opts to exit the program, a Disconnect command is issued to the server containing the "DSCT" command code. If the server receives a "DSCT" command, it should close the socket associated with that user. No response is issued by the server in the event of a "DSCT", and client implementations should close all open sockets and I/O resources prior to exiting the program.

4. ERROR MESSAGES

A robust set of error messages has been made available for the purpose of communicating useful, user-friendly instructions. As described above, "ERR" messages are sent by the server when the requested operation cannot be performed as per the user's intent. Most of the time, error messages are sent as the result of a no-op. For instance, if the server is unable to create a new chat room under the name provided because one already exists under that name, the server responds with a message conveying as much. It is the responsibility of the client-side developer to see that the return-message semantics are conveyed to the user upon receipt. A comprehensive listing of available error-messages is conveyed on a per-command basis in Section 3, above.

5. AUTHENTICATION

At this time, credentialing and authentication is unnecessary for a client to connect with a server. This may be implemented at a later date. In order to connect with the server, a user must simply provide a username. As of this writing, there is no requirement that usernames be unique within the context of a single server "life-cycle;" multiple users may have the same username, and/or the same user may have multiple simultaneous sessions. There are obvious downsides to this design, and it is recommended that this be addressed in server-side implementations. Usernames do not persist in the event of a server power-down or crash, though this may also be remedied on a per-implementation basis.

6. IMPLEMENTATION DETAILS

There is currently a single server implementation and a single client implementation of this protocol which utilizes each of the command codes and responses described in Section 3. This section is provided to those seeking guidance on their own implementations, and utilizes knowledge gained throughout the process of said implementation.

6.1 Network protocol

Implementations thus far have made no assumptions regarding the use of TCP, UDP or multicast IP. TCP is recommended due to the measures it takes for ensuring complete and accurate delivery of packet data; however, because these are simply a "toy" protocol and implementation as of this writing, UDP does not introduce any particular risks.

6.2 Asynchronous operation

In order to service multiple clients independently, the current server implementation is subdivided into a server class and a "client handler" class. (In the current Java implementation, ClientHandler extends Thread.) Upon instantiating a new connection with a client, the server creates a new client handler object (thread) through which all further client communications are handled. Thus, the server provides relatively few services; in the current implementation, the server is responsible for maintaining Room and User lists, establishing new client connections, instantiating datastreams with said clients, then handing each client off to its newly instantiated handler. It would not be incorrect to assume that the server will be far simpler to implement than the client handler.

6.3 Command parsing

As suggested above, it is best for any/all messages to be passed to the client handler for parsing; this enables the server to handle operations solely associated with creating new client connections. Handling all parsing asynchronously via independent threads is a performance win for the server.

6.4 Message Delivery

There is no limit to buffer-size in the aforementioned implementation, however it is recommended that messages be limited to some predetermined size in subsequent implementations as a security measure. Limiting incoming message size (messages, in this case, referring to those posted by a user to a chat room) also addresses buffering/performance issues that might arise in the event the server is flooded with numerous Post or Retrieve commands at once.

6.5 Connection Liveness

Each client handler should check for input-stream liveness regularly in order to ensure connectivity. (The current implementation does so on each iteration.) Client handlers should gracefully handle client crashes, meaning the server and other open client handlers continue to provide services to their associated clients with no disruption. In the event of a client-side crash, the client handler should remove that user from all applicable chat rooms and close the assigned I/O data streams.

Similarly, in the event of a server crash, client implementations should handle any resulting Exceptions with a user-friendly message, followed by closure of any open resource (i.e. sockets and data streams).

6.6 Connecting Server to Client

The server must keep two data structures: a list of rooms and a list of users. The server listens for new connections on Port 6789; all clients wishing to make a connection must do so on this port. Once connected, the new username must be added to the server's list of active users. In the current implementation, this task is performed by the client handler each time a new user thread is instantiated. It is via these lists of active users and rooms that the appropriate responses and error messages are issued. (To reiterate, these data structures MUST be held by the server, NOT by the client handler; while each client handler may add and remove users and rooms from these lists, each client handler has awareness of only one user -- or thread -- thus it cannot be held responsible for the master lists.)

6.7 Disconnecting Client from Server

When a client disconnects from the server (whether via a user-initiated “DSCT” command or on account of a client-side crash), the server must remove the username from its master list of active users and remove the username from any applicable rooms. As recommended above, the server must also close any open resources associated with that client, such as sockets and data streams. No messages are sent from the server to the client in the event of a disconnection, though the event should be logged.