Independent Submission M. Alaniz

Request for Comments: XXXX November 2019

Internet Relay Chat Protocol (Ala-chat)

Abstract

This document describes Ala-chat, an application layer text-based communication protocol. Ala-chat’s creation has been an intellectual exercise meant to explore the fundamental principles of internet protocol design and as such Ala-chat is not meant for real world deployment. Ala-chat draws no direct inspiration from the IRC described in RFC 1459 but, as Ala-chat has been created to provide a subset of the services RFC 1459 provides, it can be evaluated against RFC 1459. Ala-chat is built on the Twisted python library. Ala-chat uses Twisted’s transport layer interfaces and relies on Twisted’s backend for multithreading handling. The actual application protocol is an original design.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <http://www.rfc-editor.org/info/rfc> but also may not be available.

1. INTRODUCTION

Ala-chat is a pared-down text-based conferencing application. The following describes its first inception. Ala-chat may be subject to alteration or redesign at developer’s discretion. In fact, since the protocol is still in its infancy, alterations, including wide scale alterations, should be expected. Current deficiencies and strength will be explored below.

Ala-chat is built with Python’s Twisted networking library. Ala-chat relies on Twisted TCP implementation for network layer communication, multithread management, and standard input/output management. As a result, Twisted is essential to Ala-chat and Twisted (unfortunately) cannot be replaced by another library without wide scape, full-stack, redevelopment.

Ala-chat uses the client-server model. As of this documentation Ala-chat server is not capable of distribution. That is, there must be one server process on one host. However, the client is capable of distribution. That is, multiple client processes can exist on multiple hosts across the internet. Each client forms a single persistent TCP connection with the server and the server routes messages between clients.

As of now, clients can create, list, join, and leave chatrooms. Desirable features for future addition include the automatic removal of an empty list, the ability for a chatroom creator to mange the room, the ability for user to privately chat with one another, and the ability to reserve a password across Ala-chat sessions.

* 1. Server

As of this writing, Ala-chat consists of a single server process running on a single host. As such Ala-chat is subject to a single point of failure. Enabling Ala-chat to run over several hosts across the internet in a distributed fashion would require wide scale reengineering of the server. Additionally, any updates to the server will require the application shut down for at least the duration of the updates.

Although unable to run in a distributed fashion, the server can handle multiple client connects across the network. As of this writing the server can handle at least 4 different client connections. Future versions of Ala-chat will require the server be put through more rigorous testing to establish an upper limit on the number of client connections the server can maintain at once. As of now the exact number of client connections the server can maintain simultaneously is unknown.

All messages between clients are routed to one another by the server. That is, the server manages which user is part of which group (or chatroom) and routes messages between clients of a chatroom on behalf of the clients.

* 1. Clients

Each client is uniquely identified via a character username that is established at connection with the server. As of this writing clients do not send messages directly to one another. That is, clients signal to the server when they want to be associated with a room. After being associated with a chatroom, clients send messages to the server. The server then broadcasts those messages to every other client associated with the given chatroom. As of this writing the clients can be a part of one, and only one, chatroom at a time.

* 1. Chat rooms

A chatroom is merely an association of clients maintained and managed at the server. Clients may associate with multiple chatrooms across an Ala-chat session but can maintain only one association at a time. Clients cannot send message until they join a chatroom. After joining a chatroom, clients send text messages to the serve. Once the message arrives at the serve, the server then broadcasts the message to each other client associated with the same chatroom. Chatrooms are created, destroyed, and managed by the server. Although the server also maintains room membership, each client maintains a state variable with its current room affiliation. This simplifies implementation by allowing the client to specify the chatroom in its communications with the server. This dramatically reduces the complexity of message routing although does require slightly more over head as both the client and server sides of Ala-chat maintain this state information.

1. ALA-CHAT CONTROL FLOW

The following details the sequence of programmatic state (or phase) the server and client are during communication.

* 1. Overview

The Ala-chat server exists in a single phase. The clients exist in one of 3 phases. All packets exchanged between the server and a client begin with a 5-byte header. If a packet contains more bytes than the 5 header bytes, and how those bytes are structured, depend on the phase the client is in. The header bytes are either a signal or a command depending on the phase of the client.

* 1. Server

The server exists in one packet receiving state. In that state it receives packets from clients, decomposes the packets, and performs actions. Upon receiving a packet, the server removes the 5-byte header, determines if it is a signal or a command (acts on the command if it is one) and then responds. The response has the same 5-byte header as the message the response results from, followed by an acknowledgment or a negative acknowledgment, and then possibly followed by information depending on the state of the client and the command. The code uses Twisted asynchronous masking of multithreading and so although written in away that does not appear to handle multiple simultaneous action request from clients, it can and does.

It must be noted that as of this writing the server maintains no username database. As a result, users need to create and register a username at the beginning of every Ala-chat session and cannot reserve a specific username for use in future sessions.

* 1. Client

The client exists in one of 3 distinct phases (GREET, CONVO, and MESSG) and the client side is where most of Ala-chats complexity is located.

2.3.1 Connection establishment and GREET phase

Ala-chat relies on Twisted’s framework for TCP connection establishment. Once the TCP connection is established the client makes its presence known to the server at the application level via a 5-byte greeting packet that reads: HELLO. To which the server responds with the 5-byte WHAT! packet. Upon receiving the WHAT! the client and server are aware of one another at the application level and the client enters the GREET phase.

In the GREET phase the client is locked into a username validation loop. It gets a potential username from the user, attaches a 5-byte header that reads MAKE! to the front of the message, sends the message to the server, and waits for a response. Depending on uniqueness of the username the server responds with a 5-byte NACK! or ACK!! NACK! signals to the client that the username is taken that the client needs to continue to loop. ACK!! signals to the client that the username the user has chosen is unique and that the TCP connection between the server and client has been associated with the given username. The client then informs the user of the success and enters the CONVO phase.

2.3.2 The CONVO phase

In the CONVO phase the client is in a loop in which the client gets commands from the user, translates and packetizes them, sends the commands to the server to be acted upon, and waits for results or acknowledgments from the server before informing the user of success or failure of the command. Commands include creating, listing, or entering chatrooms and closing the session. Each action has a unique header associated with it. Entering a room cause the client to enter the MESSG phase and from the MESSG phase the client leaves the MESSG phase; that is, leave a chatroom.

2.3.3 The MESSG phase

After the user requests to join a valid chatroom, the client enters a receive and send message phase in which the user cannot request any actions. That is, all the user can do is send messages to other users in the room while simultaneously receiving messages from those same users. The user will need to leave the room in order to see the actions menu or to make or join another room. In the message phase all messages sent to the server and received from the server have the MESS! header. The packets additionally contain the room name, the username of the user who sent the message, and the message itself. To leave a chatroom, while in the room the user must type: “IM LEAVING THE ROOM” and click enter to leave the room.

1. ALA-CHAT PACKET SPECIFICATION
   1. GREET phase packets

* HELLO
  + Sent by client on TCP connection creation
* WHAT!
  + Response to client HELLO, acknowledges client’s existence.
* MAKE[username]
  + Client to server message.
  + Asks server to associate this TCP session with the username: [username].
* ACK!!
  + Server signals to client that username is unique and so valid.
* NACK!
  + Server signals to client that username is not unique and so is invalid.
  1. CONVO phase packets
* CREAT[chat room name]
  + Client to server message.
  + Requests the create of a chatroom called [chat room name].
* CREATACK
  + Server signal that the room name is unique and valid and so has been created.
* CREATNACK
  + Server signal that room name is not unique and so invalid and no created.
* LIST!
  + Client request for list of room names.
* LIST![list of room names]
  + Server response to client.
  + [list of room names] is a newline (‘\n’) character delaminated list of chatroom names.
  + Server converts the list to bytes before sending.
  + Client converts the received bytes to a python string before processing.
* JOIN![username length][username][room name]
  + Client to server message.
  + [username length] is the length of the username.
  + [username] is the username.
  + [room name] is the name of the room.
* JOIN!ACK
  + Server to client message.
  + Signals that the user is now part of the room and so client should phase transition from CONVO to MESSG.
* JOIN!NACK
  + Server to client message.
  + Signals that the room doesn’t exist and so client needs to stay in CONVO phase.
* LEAVE[room name]
  + Client to server message.
  + Signals to server sending username wishes to leave the chat room named [room name].
* LEAVE
  + Server to client message.
  + Acknowledges that client request has been processed.
  + Client should phase transition from MESSG to CONVO if it is in MESSG phase.
* TERM!
  + Client to server message.
  + Signals to server that client will close connection.
* TERM!ACK
  + Server to client message.
  + Acknowledges client’s connection close request.
  + Server release saved username and disconnects.
  + Client disconnects.
  + Client shuts down.
  1. MESSG phase packets
* MESS![room name length][room name][message]
  + Client to server message. Sent only while in MESSG phase.
  + [room name length] is the length of the chat room name.
  + [room name] is the name of the chat room the sender is affiliated with.
  + [message] is the message the user is sending to their peers.
* MESS![message]
  + Server to client message.
  + [message] is a message from a user in a chatroom. [message] is broadcasted to all room members and is printed to screen when it reaches client side.