CS594
Internet Draft, Portland State University
Intended status: IRC style ChatApp Project Specification
Expires: July, 2020

ChatApp IRC Style Class Project

Abstract:

This document specifies the format and use of messages sent between Client and Servers functioning as a part of ChatApp an IRC style client/server system for the CS594 Internetworking Protocols class at Portland State University.

Table of Contents

1. Introduction

This specification describes a simple Chat App protocol similar to IRC that can be used by clients to communicate via the console of their computers. Only text based communication is This is a client/server based system where the Server routes messages to and from clients.

Chat rooms are used as the message organizing paradigm. A room represents a topic of conversation. Users can create rooms and join rooms. Messages sent to rooms are viewable by all logged in users who have joined the room. Users can elect to message all rooms they have joined to or specific joined rooms. Upon joining a room, users will receive the most recent 10 messages sent to that group. When users leave a room they will no receive the messages associated with that group.

Users can list all created groups and view all users who have joined each room.

2. Key words

In this document: "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document have the same connotation as they do in RFC 2119 [RFC2119]. Capitalization signifies key word status, lower case variations should not be seen as equivalent. Key words have been bolded.

3. Basic Information

All communication in the ChatApp protocol uses TCP/IP, the server listens for connections on port 1234. The client(s) connect to the server over this port. The client sends messages and system requests to the server over this socket. The server replies in the same way. Messages are sent as byte streams. The client sends messages to the server at any time while the connection is open, and the server sends messages to the client as a result of requests initiatied by the client, messaes sent to a room the client has joined, or other system events the server determines as required for the client. The server and client may close the connection at any time. The server and client MAY elect to send a connection message informing their counter part of the connection closing. The server does not limit the number of users or rooms, and can not guarantee performance due to the specific limitations of hosts

4. Message Structure

a. Generic Message Structure:

```
class Message(object):
  def __init__(self, timestamp=None, sender=None, dest=None, data=None):
     if timestamp is None:
        timestamp = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
     self.timestamp = timestamp
     self.sender = sender
     self.dest = dest
     self.data = data
```

i. Attribute Definitions and Requiremens:

1. timestamp - string representing data time that message was created, is created automatically if the field is not specified when initialized, SHOULD not be empty string
2. sender - string representing entity sending message, MAY be ‘Server’ for messages originating at the Server or MAY be the username of the client where the message originated MUST verify that data in this filed does not exceed the MAX_NAME size prior to sending messages, SHOULD not be length 0
3. dest - string representing entity as the intended destination value is dependent on the type of message being sent and where it originated server and client MUST verify that data in this filed does not exceed the MAX_NAME size prior to sending messages, SHOULD not be length 0

4. data - string representing message text, server and client MUST verify that this field does not exceed the MAX_DATA value, SHOULD not be length 0
5. len(msg_out) is the length of the entire message being sent the value MUST be validated by client and server for all incoming messages and is used to determine how many bytes to read in
6. Base.header_len is the globally constant length of the header is stored as header_len

ii. Message Types:

1. ConnectionMessage
2. CreateRoomMessage
3. JoinMessage
4. LeaveMessage
5. ChatMessage
6. Message

iii. Validation constant definitions:

Constant	Definition	Value
MAX_MSG	Max length of Message	2000 bytes
MAX_NAME	Max length of room or user name	50 bytes
DATA_MAX	Max length of data field	1000 bytes
header_len	Size of header buffer	20 bytes

b. Error Message Format:

i. Generic Error Structure:

```
class Error:
  def __init__(self, code=99, data='ERROR UNSPECIFIED'):
    self.code = code
    self.data = data
```

ii. Error Codes:

Code	**Error Message**
99	Generic Error
2	CONNECTION ERROR: lost connection from {ip or

	username}
3	CONNECTION ERROR: {username} username already in use
4	MSG ERROR: msg in message exceeds {MSG_MAX}
20	ERROR: empty string submitted
21	ERROR: name exceeds MAX length {NAME_MAX}
22	ROOM ERROR: {room} does not exist
24	ROOM ERROR: {name} username already joined to {room}
25	ROOM ERROR: {room} name already in use
26	MSG ERROR: data in message exceeds {DATA_MAX}
27	ROOM ERROR: {name} username not joined to {room}
28	WARNING: could not join all selected rooms
23	USER ERROR: username does not match username associated with connection

iii. Use

1. General Use:
    a. Provide notification that illegal data has been provided or a connection error has occurred
    b. MAY be sent by server to the client
    c. MAY be used by client or server for internal input validation prior to sending messages to recipient
2. CONNECTION ERROR Use:
    a. MAY be sent by either the client or server to signify that unplanned termination of the connect
    b. SHOULD be consider the connection closed upon receiving such an error

c. Keep Alive Management

MUST be managed on the client via functions in the socket library. MUST test for a connection to the server on an interval specified by CHECK_TIME. If keep alive check fails the FAIL_LIMIT number of times, the connection is closed and the application terminates.

5. Transmitted Message Context:

Transmitted messages must be byte encoded UTF-8 strings. On the client and server side, messages are treated as byte encoded JSON objects and are encoded and decoded as such. Messages MUST not exceed the MSG_MAX.

a. Each message MUST be preceded by a header when sent over the connection. msg_out_header = f"{len(msg_out):<{base.header_len}}".encode('utf-8')
b. Each header MUST be byte encoded prior to transmission of the format string encoded integer, header must be of size header_len:
   b'int '
   i. int represents the size of the preceding message and MUST not exceed the MSG_MAX
c. Message Structure when sent and received:
   b'{"__type__": " ", "timestamp": " ", "dest": " ", "sender": " ", "data": " "}'
   i. The class name of the entity sent represents the type of Message. There is a class for each Message type.
d. Error Message Structure when sent and received: b'{"__type__": "Error ", "code": " ", "data": " "}

6. Client Messages:

a. General Message Info:

i. Messages sent to Server from Client Chat App:
   1. timestamp attribute is set to current date and time as str formatted month/day/year hour:minute:seconds
ii. Client Chat App SHOULD perform validation on outgoing messages and incoming messages from Server and MAY present Errors to the console that are not included in the Error Codes outlined above

b. ConnectionMessage ‘CONNECT’ sent to Server:

i. Format:

```
class ConnectionMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(ConnectionMessage, self).__init__(timestamp,
        sender, dest, data)
```

ii. Use:
Before any further messages can be sent and upon launching the Client version of ChatApp, an initial ConnectionMessage MUST be sent by the Client to provide a username.

The server MUST associate the username to the client’s socket connection. This message SHOULD only be sent once. If the client provides a username that is already in user by another client connection, the server MUST send an error message.

Attributes:

1. timestamp - SHOULD set as the current date time in the format: month/day/year hour:minute:seconds
2. sender - the username the client wishes to connect as
    a. MUST not already be in use by another connected socket will result in Error 3 and connection will be terminated if message reaches server
    b. MUST not be empty will result in Error 20 and connection will be terminated if message reaches Server, will be validated and user can retry in Client ChatApp
    c. MUST not exceed the NAME_MAX length, will result in Error 21 and connection will be terminated if message reaches server, will be validated and user can retry in Client ChatApp
3. dest - SHOULD be 'Server'
4. data - MUST be set to 'CONNECT'
    a. If not set to connect connection attempt will not occur in Server and connection will terminate
    b. Client ChatApp sets this value

c. CreateRoomMessages sent to Server

i. Format:

```
class CreateRoomMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(CreateRoomMessage, self).__init__(timestamp,
    sender, dest, data)
```

ii. Use:

Sent by a client to create a new chat room. A chat room MUST exist prior to being able join or send chat messages to other users. A client MAY send a CreateMessage at any time.

A room name must be unique and not already in use by another user. A client SHOULD not assume that a room has been created just because a CreateRoomMessage has been sent.

iii. Attributes:

1. timestamp- SHOULD set as the current date time in the format: month/day/year hour:minute:seconds
2. sender- SHOULD be username of client
3. dest- SHOULD be 'Server'
4. data- name of room user wishes to create
    a. MUST not already be associated room doing so SHOULD result in Error 25 and room will not be created
    b. MUST not exceed NAME_MAX will result in Error 21

c. MUST not be an empty string will result in Error 20

d. CreateRoomMessages received from Server

i. Format:

```
class CreateRoomMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(CreateRoomMessage, self).__init__(timestamp,
        sender, dest, data)
```

ii. Use:

SHOULD be received by a client when a new room has been created. In ChatApp, this adds a room to the ALL_ROOMS dictionary.

This allows the Client ChatApp to support listing all of the rooms that currently exist and can be joined, the $view_all$ option.

iii. Attributes:

1. timestamp- SHOULD be current date time server sent message
2. sender- SHOULD be 'Server'
3. dest- SHOULD be 'ALL' since all connected clients will receive this message
4. data- name of room that has been created
    a. In ChatApp if name is already in ALL_ROOMs will result in Client Server mismatch error printing to console
    b. If name not in ALL_ROOMS, room will be added to the dictionary

e. JoinMessage sent to Server

i. Format:

```
class JoinMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(JoinMessage, self).__init__(timestamp, sender,
        dest, data)
```

ii. Use:

Client MAY send JoinMessage to Server when the user wishes to join a room and send ChatMessages to a room. User can enter a comma seperated list of rooms to join, but a single join request will be sent for each room listed. Failure to join all listed rooms will result in a WARNING Error.

In Chat App Client a user MAY elect to join one or more rooms. A room MUST exist in order to be joined. A user SHOULD NOT join a room that they have already joined.

iii. Attributes:

1. timestamp- SHOULD be current date time client sent message

2. sender- MUST be username associated with client in server
    a. Not validated in Client ChatApp
    b. Vailure to match username associated with connection SHOULD result in a failed join request and a subsequent Error 23 from Server
3. dest- SHOULD be ‘Server’
4. data -SHOULD be room name to join
    a. MUST not exceed NAME_MAX will result in Error 21
    b. MUST not be an empty string will result in Error 20
    c. MUST not be the name of a room the user has already joined will result in Error 24
    d. MUST be room name of existing room will will result in Error 22

f. JoinMessage received from Server

i. Format:

```
class JoinMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(JoinMessage, self).__init__(timestamp, sender,
        dest, data)
```

ii. Use:

Join messages are sent to the client when any logged in user joins a room. Upon reciept in the Client Chat App, the ALL_ROOMS entry associated with the room in the data attribute of the message will be updated and the user associated with the sender in the messages will be append to the corresponding user list. The MY_ROOMS dictionary will also be updated with name of the room.

These messages support the ability to list any user that has joined a room. In Client ChatApp this supports the $list$ option. Also supports validation on Client side that user is only sending messages to their joined rooms.

iii. Attributes:

1. timestamp- SHOULD be current date time server sent message
2. sender- SHOULD be username associated with client that joined the room
    a. MUST not be empty string, will result in Error 20
    b. MUST not already be associated with room, will result in Error 24
3. dest- SHOULD be ‘Server’
4. data -SHOULD be room name to join

a. MUST not be an empty string will result in Error 20
b. MUST not be the name of a room the user has already joined will result in Error 24
c. MUST be room name of existing room will will result in Error 22

g. ChatMessage sent to Server
   i. Format

```
class ChatMessage(msg.Message):
 def __init__(self, timestamp, sender, dest, data):
        super(ChatMessage, self).__init__(timestamp, sender,
dest, data)
```

   ii. Use:
   User MAY send a chat message to any joined room. Message will be seen by any logged in user that has joined the room. From Client Chat App a user can send the same message to multiple joined rooms. Will be processes as a ChatMessage request for each room selected. So if 4 rooms were selected as recipients, 4 ChatMessages will be sent.

   User MUST be joined to room to send message to room.
   iii. Attributes:
      1. timestamp- SHOULD be current date time server sent message
      2. sender- SHOULD be username associated with client
         a. MUST be associated with room, will result in Error 27
         b. MUST be username associated with socket's username, will result in Error 23
      3. dest- MUST be name of room to send message to
         a. MUST not be empty string, will result in Error 20
         b. MUST be an existing room, will result in Error 25
         c. MUST not exceed NAME_MAX, will result in error 21
      4. data -SHOULD text based message intended for room
         a. MUST not be an empty string will result in Error 20
         b. MUST not exceed the DATA_MAX, will result in Error 26

h. ChatMessage received from Server
   i. Format

```
class ChatMessage(msg.Message):
 def __init__(self, timestamp, sender, dest, data):
        super(ChatMessage, self).__init__(timestamp, sender,
dest, data)
```

   ii. Use:
   Upon reciept in Client Chat App, message SHOULD be displayed to user in format:

```
print(f"[ROOM: {room}] >> {self.sender} >> {self.timestamp} >>
{self.data}")
```

And message will be appended to MY_ROOM entry associated with room name.

iii. Attributes:
1. timestamp- timestampe from message that originate with sender
2. send- username of client that sent the message
3. dest- name of room
    a. Must not be empty string
    b. Must be in MY_ROOMs list in Client ChatApp
4. data- text of message sent

i. LeaveMessage sent to Server:

i. Format:

```
class LeaveMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(LeaveMessage, self).__init__(timestamp, sender,
        dest, data)
```

ii. Use:

Client sends LeaveMessage to no longer be joined to a room and to no longer send and receive ChatMessages associated with a room.

In Client ChatApp a user can enter multiple rooms they wish to leave, a single message will be sent for each room. The room will be deleted from MY_ROOMS dictionary.

A user MUST be joined to a room in order to leave the room.

iii. Attributes:
1. timestamp- SHOULD be current date time client sent message
2. sender- MUST be username associated with client in server
    a. Not validated in Client ChatApp
    b. Vailure to match username associated with connection SHOULD result in a failed join request and a subsequent Error 23 from Server
3. dest- SHOULD be ‘Server’
4. data -SHOULD be room name to leave
    a. MUST not exceed NAME_MAX will result in Error 21
    b. MUST not be an empty string will result in Error 20
    c. MUST be the name of a room the user has already joined will result in Error 27
    d. MUST be room name of existing room will will result in Error 22

j. Leave Message received from Server

i. Format

```
class LeaveMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(LeaveMessage, self).__init__(timestamp, sender,
        dest, data)
```

ii. Use

Leave Message from server signifies that a user has left a room. Upon receipt Client App SHOULD remove the sender from the corresponding room in ALL_ROOMS that is specified in the data field.

Room in data field SHOULD exist and sender SHOULD be in the associated ALL_ROOMS[data] list.

iii. Attributes

1. timestamp- SHOULD be current date time server sent message
2. sender- SHOULD be username associated with client that joined the room
    a. MUST not be empty string
    b. MUST already be associated with room
3. dest- SHOULD be 'Server'
4. data -SHOULD be room name to leave
    a. MUST be the name of a room the user has already joined

k. ConnectionMessage QUIT sent to Server

i. Format:

```
class ConnectionMessage(msg.Message):
        def __init__(self, timestamp, sender, dest, data):
            super(ConnectionMessage, self).__init__(timestamp,
        sender, dest, data)
```

ii. Use:

SHOULD be sent prior to a graceful quit from client. MAY be sent prior to an unplanned exit.

Used to signify that client session is ending. Client ChatApp will close after issuing this request.

iii. Attributes:

1. timestamp - SHOULD set as the current date time in the format: month/day/year hour:minute:seconds
2. sender - the username of the client
3. dest - SHOULD be 'Server'
4. data - MUST be set to 'QUIT'
    a. If not set to quit connection will not be treated as connection close request

7. Server Messages

a. ConnectionMessages ‘Connect’ received from Client

i. Format and Attributes: See 6.b.i, 6.b.iii

ii. Use:

SHOULD be received after detecting intial socket connection from client.

If username is valid, see 6.a.iii for validation:

- MUST add socket to CLIENTS dictionary and associates username
- MUST add socket to SOCKETS_LIST
- MUST send client a created message for each room in the ROOMS list: see 6.d.i and 6.d.iii for format and attributes
- MUST send client a joined message for each user that is joined to a room in the JOINED list see 6.e.i and 6.e.iii for format and attributes

If username is invalid, an ErrorMessage will be sent to user. See 6.a.ii for associated error codes.

- If username is already is connection MUST not set connection

b. ConnectionMessages ‘Quit’ received from Client

i. Format and Attributes: See 6.k.i and 6.k.iii validation

ii. Use:

SHOULD be received after client planned connection is terminated.

Upon receipt:

- MUST send LeaveMessage to for each room that client has joined, message is sent to all logged in users in the CLIENTS list
- MUST remove username from JOINED list associated with room
- MUST remove socket from SOCKETS_LIST
- MUST remove socket username pair from CLIENTS list
- Closes socket
- Uses the username associated with the socket for determining user to perform termination tasks agains

c. CreateRoomMessages received by and sent from Server

i. Format and Attributes: See 6.c.i, 6.c.iii, 6.d.i, 6.d.iii

ii. Use:

Upon receipt of a CreateRoomMessage from a client and if valid, see validation and error codes in 6.c.iii:

- Room MUST be added to ROOMS list
- CreateRoomMessage must be sent to all logged in users in the CLIENTS list
- Room added to JOINED dictionary and list is initialized to empty
- If request is invalid sends ErrorMessage associated with error

d. JoinMessages received by and sent from Server

i. Format and Attributes: See 6.e.i, 6.e.iii, 6.f.i, 6.f.iii

ii. Use:
Upon receipt of a JoinMessage from client and if valid, see validation and error codes in 6.e.iii
- MUST add Sender to list in JOINED dictionary associated with room
- MUST send JoinMessage to all logged in user see 6.f
- If invalid sends ErrorMessages associated with error encountered
- MUST send most recent 10 ChatMessages associated with room to joining user

e. LeaveMessages received by and sent from Server
i. Format and Attributes: See 6.i.i, 6.i.iii, 6.j.i, 6.j.iii
ii. Use:
Upon receipt of a JoinMessage from client and if valid, see validation and error codes in 6.d.iii
- MUST remove sender from JOINED list associated with room
- MUST send LeaveMessage ito all logged in users
- If invalid sends ErrorMessages associated with error encountered

f. ChatMessages received by and forwarded from Server
i. Format and Attributes: See 6.g.i, 6.g.iii, 6.h.i, 6.h.iii
ii. Use:
Upon receipt of a JoinMessage from client and if valid, see validation and error codes in 6.g.iii
- MUST append to ROOMS list
- MUST send contents of message to all users associated with the Room in the JOINED list

8. Error Handling

Both Client and Server MUST detect when socket connection linking them is closed. This is done on the client side via keep alive management and checking that a header is received on each incoming message being received. If the client detects that server connection is closed, the client application will exit. This done on server side, via detecting headers, and processing messages received from client. If closure is detected, then the process outlined in 7.b.ii is completed and the socket is closed.

Errors that are not fatal and do not result in terminating a connection MAY be sent from the server to client and the client MAY be prompted of these errors when interacting through Client ChatApp.

9. Conclusion & Future Work:

This specification is for a very basic console based ChatApp protocol. Without modifications it is possible for a client that is on the same network (unless port forwarding is configurable on the client router) as the server to send text based

messages to other users who are logged into the system. Data is sent as byte streams. Future work could include supporting file transfer, or encrypting data for secure transfer. Additionally private or user specific messaging could easily be implemented.

10. Security

Messages are not encrypted and do not have protection against being inspected or violated. The server coverts all byte streams to strings, and can therefore easily see all messages that come through it. If security is a concern, encryption protocols should be added prior to use.

11. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC sample document provided for class] IRC Class Project Specification , Dec2015-June2016