The WS_2017 protocol

Max Hackinger

April 16, 2017

Contents

Abstract

This document will Introduce and define the WS_2017 protocol. The WS_2017 protocol uses the POP3 protocol as a reference.

1 Session States

The session states in the current draft are directly lifted from the POP3 protocol.

The session goes through several states during it's life time. After the TCP connection is established and the server sends a greeting and the session enters the AUTHORIZATION state. By transmitting a user name to the server with the **uname** command the client authorizes it's self and the session can advance to the TRANSACTION state. In the TRANSACTION state the client can execute commands till the client sends the **cquit** command, that then moves the session in to the UPDATE state. In this state the server releases all resources from the TRANSACTION state and says goodbye and the TCP connection is closed.

2 Basic Overview

The WS_2017 service is started by the server listening to port 1030. To start a session the client establishes a TCP connection with the server and the server sends a greeting to the client. Commands are exchanged between the client and server till the connection is closed or aborted.

3 Commands

All commands are case insensitive and made up entirely of ASCII characters. Commands always have exactly one **keyword** followed by none or more **arguments**. The keyword is never longer then 5 characters. Commands are always terminated with a CRCF (Carriage Return: \r, New Line: \n). Commands will either be answered with a positive response confirming that the command has been understood and processed or negative response pointing out what is wrong.
A positive response has a '+OK' followed by the command that was successful and if required one or more argumments. A negative response has a '-ERR' followed by the command that failed and an argument that either is a message with what went wrong or a suggestion for change that is relevant to the keyword.

If the entered command is badly formatted the Server should return:

s: -ERR '<command> is not a properly formatted command'

If the entered command does not match a valid command, the server should return:

s: -ERR 'entered command does not exist'

4 AUTHORIZATION State Commands

Once the session has gone in to the AUTHORIZATION state, the server will be expecting a user name to identify the client by. This is done by sending a command with the desired user name to the server. If the name is already present in the server (i.e there is already a client connected with that name), the server returns a negative response. If the name given to the server is unique the server confirms it with a positive response. As soon as the client has entered the user name, it is broadcasted to all other connected clients. When a user disconnects from the server the user name is removed.

registering a name:

> c: uname <name>
> s: +OK uname 'you are' <name>

own username entered:

> c: uname <name>
> s: -ERR uname 'same username entered'

username already taken:

> c: uname <name>
> s: -ERR uname suggested <name_suggestion>

broadcast new user name to other clients:

> s: nuser <name>

5 TRANSACTION State Commands

5.1 Ping/Pong

Every 5 seconds server and client should exchange a ping and pong which is initialized by the server, to make sure that they are still connected. If there is no response within 15 seconds (after 3 pings) the connection should be disconnected. The server does this by starting a thread that sends out a ping to the client, which if it is not interrupted soon enough by the client with a pong, it removes the client form the user list and closes the socket. On the client side, once the ping has been received from the server, the client starts a thread that if not interrupted by a server ping within 20 seconds, will shutdown the client

ping/pong:

s: cping
c: cpong

5.2 Changing user name

To change user name the same command is used as to enter the original user name in the AUTHORIZATION State.

change name:

c: uname <new_name>
s: +OK uname 'you are' <new_name>

When the a name is changed this has to be sent to all other clients:

s: +OK nuser <old_name> <new_name>

(actully part of the AUTHORIZATION State)When a user joins the server the name is announced to all users

s: nuser <newusername>

5.3 Get all user names

To be able to send a message or find to an other user one needs to know the name of the other users.

c: cgetu
s: +OK cgetu <user0> <user2> <user2> ...

5.4 Chat

5.4.1 Main Chat

When chatting, the server acts as a relay between two clients. When the message arrives at the server, the server sends the message to the recipient. To ensure that the message was sent properly, the client also sends the message to itself. The client sends the messages to every client as a separate message.

sender:

c: chatm <sender_name> <recipient_name> '<message>'

server to recipient:

s: chatm <sender_name> <recipient_name> '<message>'

5.4.2 Whisper Chat

When sending a whisper message the server sends the message both to the recipient and the sender. The second for the sender to know the message has been delivered.

sender:

c: chatw <sender_name> <recipient_name> '<message>'

server to recipient and sender:

s: chatw <sender_name> <recipient_name> '<message>'

5.5 Games State

5.5.1 Game Creation and Destruction

The Client tells the server that they created a new game.
Client to Server:

newgm <points> <gamename> <mapname>

If name is Taken, Server writes back to client. Server informs the Client that the chosen game name is already taken.
Server to Client:

-ERR game name taken

Else the Server tells then all the Clients about the new game (including creator). The Server informs the Clients about the new game which can now be joined.

Informs all the clients that this game was deleted (because every player left it). The game could either be already playing or still be in the starting phase.
Server to all Clients:

rmgam <gamename>

The Client informs the Server that they want to join the game.
Client to Server:

joing <gamename> <username>

The Server informs all the Clients, that the client has joined this game.

The Server informs the user that there is no place left in the game.
Server to Client:

-ERR joing <gamename> already full

The Client tells the server that it is ready and has chosen the specified characters.
Client to Server:

ready <username> <gamename> [<characterstring>]

The Server tells the Clients (including sender) that the Client is now ready.
Server to Client:

ready <username> <gamename> [<characterstring>]

[<characterstring>] is formated as following (with any number
of characters > 0):
[<character1name> '<weapon1name>' <character2name> '<weapon2name>']

Client asks about all open games (similar to cgetu).
Client to Server:

cgetg

Client asks when getting the answer for the cgetu command, after registering all usernames (to make sure they are registered). Server responds with a response for each game (waiting = not yet started; running = already playing):

+OK cgetg waiting <gameName> <maxPoints> <mapName>
<username1> (ready [<characterstring>]|choosing) <username2>
(ready [<characterstring>]|choosing)
+OK cgetg running <gameName> <maxPoints> <mapName>
<username1> <username2>

The Client informs the server that they left the game.
Client to Server:

leavg <gamename> <username>

The Server informs all the Clients, that the Client has left the game.
Server to Client:

leavg <gamename> <username>

The Client tells the server that they want to start the game. Client to Server:

stgam <gamename> <username>

The server then forwards the command to all the Clients to inform them about the start of the game and which user triggered the game start. Server to Client:

stgam <gamename> <username>

5.5.2 Turn Based System

When the Client which's turn it currently is ends their turn it sends the endtn command to the Server which then informs all the Clients. Client to Server:

endtn <game_name> <user_name>

Server to Clients:

> +OK endtn <game_name> <user_name>

If the user doesn't actually hold the turn (e.g. if the endtn command was sent twice before the server could answer) this message is sent back. Server to Client:

> -ERR endtn <game_name> its not your turn

5.5.3 Player Position

Client sends position of children by using the chpos command. The server validates the
Client to Server:

> chpos <game_name> <user_name> <old_child_position: x,y> <new_child_position: x,y> <distance>

Server to Clients:

> +OK chpos <game_name> <user_name> <old_child_position: x,y> <new_child_position: x,y> <distance>

5.5.4 Damage

When the client attacks an other user the attack intensity is transmitted to the server, which then calculates the new level of wetness of the attacked and transmittes it to all clients.
If the child has reached complete wetness the server, changes the high score of the attacking user and also sends all clients the old position and returns instead of the new position a "null".
Client to Server:

> attch <game_name> <targeted_child_position: x,y> <attacker_child_position: x,y> <attack_intensity>

Server to Client:

> +OK attch <game_name> <targeted_child_position: x,y> <attacker_child_position: x,y> <wetness_value>

5.5.5 Weapons Change during Game

During the game a player can change their weapon, this is done by transmitting the chwea command to the server. Currently unused.
Client to Server:

> chwea <game_name> <user_name> <child_position: x,y> <new_weapon>

Server to sending Client:

> +OK chwea <game_name> <user_name> <child_position: x,y> <new_weapon>

5.5.6 Power-Ups

When a client finds a power-up the server sends the client a chpow command, with the powered up child position and the power-up. Currently unused.
Server to Client:

> chpow <game_name> <user_name> <child_position: x,y> <power_up>

Client to Server:

> +OK chpow

5.5.7 High Score

As soon as there is a winner, the server calculates the high score of all players and returns the high score back to all clients. The Server also saves the high score in an xml file to be able to recall the previous high score. Server to Clients:

> uhigh <game_name> <team_name> <user_name1> <score> <user_name2> <score> .....

6 UPDATE State Commands

6.1 Quit

When the client wants to terminate the connection to the server, the client uses the quit command which leads the session from the TRANSACTION state to the UPDATE state. In this state the server ends tasks related to the client in a safe manner.

c: cquit
s: +OK cquit 'terminating tasks and disconnecting'