ChatScript Client/Server Manual

Revision 9/25/2016 cs6.84

While the system defaults to running as a stand-alone chatbot under Windows, when run under LINUX it defaults to being a server.

Nominally (meaning depending on hardware and what your bot does) ChatScript can process a volley on a single core in 10 milliseconds on a slow machine, thus handling 100 volleys every second from different users using one core. A human-human volley is often around 15 seconds, so handling 1000 simultaneous users with a single core slow server is not unreasonable.

The fastest server OS for ChatScript is Linux. The Mac tends to misfire in the OS itself with heavy client loads. Windows is a much slower server in general. And Linux version of CS has support for forking chatscript (no such support under Windows), so you can run a fork of the engine on every core, saturating cpu processing to the max while still serving a single port. Speedup is nearly linear per core added.

Running the Server

When you run the Mac/LINUX program, it defaults to server mode, port `1024`. To run the server under Windows you must give it a command line parameter specifying a port. There are various command line parameters to affect behavior, described at the end.

Unique User Names

ChatScript maintains an independent history with each user-bot combination in a single file in the USER's directory. It is nominally up to you to define some unique name for each user. There is no login validation service provided by ChatScript; that is your responsibility. Some simple things which ChatScript supports directly are:

1. If the login name is "." , the system will assign a user name of the IP address it receives. This doesn't work from localhost nor will it work if you have a server of your own relaying between the client and the ChatScript server (since the IP address will always be of your relay server).
2. If the login name is *guest*, the system will assign a user name of guest concatenated with the IP address. You could also use the user's email address as a login id. ChatScript will automatically convert periods and @ in a user's name to _ , so a login like *gowilcox@gmail.com* will become *gowilcox_gmail_com*. Likewise logging in as guest will result in something like *guest_123_124_155_12*. Of course, if the user eventually comes back some other day on a different IP address, they lose their history. Can't be helped. And you end up with dead files.

Communicating with the Server

The client webpage/program connects on a socket to the IP and port of the server. If you are coming from a webpage, the webpage must establish the socket. The system does not use HTTP. HTTP is an agreement on what port to use (a standard http port) and what message protocols look like to the HTTP server. Similarly ChatScript uses an agreement on what port to use (but you get to specify the port) and what the message protocols look like.

And, each communication is a one-shot deal. The socket is made, the client sends a message to the server, the server sends data back, AND CLOSES THE CONNECTION. Given that potentially thousands of users may be using the server at the same time, it is undesirable for it to try to maintain that many open sockets. If for some reason you need to maintain a permanent connection to a client, you can write an intermediary server program that has a permanent socket to the client, and relays messages back and forth between the client and the ChatScript server.

This is also what you would do if you wanted ChatScript to "push" unsolicited messages to the client. Your intermediary server can use timeouts to decide to send a specific message to the ChatScript server, and then relay the result back to the client as an unsolicited output. Similarly, your intermediate server might receive various asynchronous events and signals, and can pass requests to

ChatScript at that time and pass the result back to a permanently connected client.

Each chat volley (incoming and outgoing message) is an entirely different connection. This means the Chatbot server is not devoting resources waiting for a user to reply and doesn't care how long it takes the user to come back again.

ChatScript Protocol

The message a client sends is a concatenation of three null-terminated text strings. The first string is the user login name. The second is the name of the chatbot to talk to. If this is a null string, the system will connect to the default bot.

The third string is the message. If the message is null, this is a start of new conversation. This MUST be the first thing you do with a new user. Ideally you do it whenever a new conversation is starting with that user which is how the system knows the old conversation ended. Usually script will detect that this is the start of a new conversation and say something like *hello* or *welcome back* to indicate the two parties are starting up a new conversation, though through the history file the system may have a lot of information about what has gone on in prior conversations. As long as the user is connected to the webpage, for example, you wouldn't send a startup message again.

Due to the requirement of a unique user name, you NORMALLY require the user to enter a login name once on the client, after which you pass that on each transmission to the server. You can bypass asking for a user name if you always just use the "guest" or "." user names.

The message sent to the server during a conversation should never be null (since that looks like a conversation start). Either always prepend a blank on every line from the user, or add a blank if the user presses ENTER without anything else or pass along the newline/cr character.

The chatbot can wait forever for each input (the connection is terminated for each volley) and the only way to know that the human "left" is when the human "comes back" with a start of a new conversation.

Testing the Server

You can test everything on your own machine in Windows using batch files in SERVER BATCH FILES. Launch the server by double clicking on server.bat and then launch the client by double clicking on localclient.bat. Since this is on your own machine, firewall opening a port is unnecessary. Remember that to be

a remote server you need to make your port available for inbound TCP if you have a firewall - if you don't have a firewall you must be insane).

If you run the Windows ChatScript engine with client=ip:port as an argument, it will act as a client to talk to and test a remote server. The ip address must be numeric and the `:port` is optional.

The client will start a new conversation and then loop with you conversing to it (assuming your server is running). If you issue a :restart command to the server, the client will prompt you for a new login. If you prefix your login name with `*`, then you get to speak first to the chatbot continuing any prior conversation (normally the chatbot speaks first starting a new conversation).

For LINUX, just perform the equivalent commands of the batch files (except that since you can't readily run multiple apps, you'll have to background the server presumably by doing a nohup command on it.

Revising a Live Server

When you want to change the contents of the server, obviously you could just stop and start it again. Of course that blocks people from using it in the interim and you might even cut off a user from getting his response. You don't have to do that.

You can restart a live server without interferring with people. In actuality, the server starts up by loading all of its data into memory (except for specific users). So the dictionary, topic data, livedata, etc are all memory resident. This means you can, while the server is running, revise those files on disk. You can, for example run a stand-alone copy and rebuilt topics from raw data. Or you can edit livedata files or dictionary files. None of this impacts a running server.

Then, as an authorized user, you can issue a :restart command. Because the chatbot server handles one user one volley at a time, when you are talking to the server no one else is (briefly). So you can tell the server to reload all of its data and that will complete your turn and it will then merrily handle the next user with the new data.

Revising a topic

Each user's record tracks where they have been in a topic. If you have changed that topic, the record becomes invalid and any memory it has of where they have been in that topic, or if they are sitting at a rejoinder there waiting for the next user input, will be forgotten. It's now a fresh topic. All other unchanged topics will not affect the user in any way.

The user's topic data does not have the actual rules of the topic (it is not a copy of the topic) because that would make the files huge. Instead, the checksum of the topic is stored with the user's topic data in USERS when they start using up rules from a topic. And each top level rule is identified by a corresponding bit in the user's topic data, which can be compressed if a block of rules is all used or all unused. So effectively we turn on a bit whenever a rule is used up.

When you bring a new topic into the system, it has a different checksum. Therefore the system detects that the user's file does not match the new topic data and should be considered invalid. IF the checksum of the new topic is 0 (a special value which you can request with the flag "safe" on the topic) then the system considered the topic automatically compatible. So if you just change spelllings or add rules at end or add rejoinders (which are not top level), such a topic can continue to be safely used with a user's saved data.

In reality, the only change where you have to worry about compatibility is altering gambits, since you can always safely add responders at the end or rejoinders anywhere. And you can even add gambits using a trick. You could insert a block of n gambits to replace an existing gambit via making the block be a new topic and replacing the existing gambit with a call to that new topic. This is a "safe change" in that someone who has not reached that gambit will get the new experience. Of course some one who has already been to that gambit will not see the revision, but can continue on the rest of the topic as it used to be.

And technically I speak a mistruth in saying that you can just dump responders at the end. IF you use a segemented responder to access the topic `^respond(~yourtopic.secondhalf)` then you would be impacting that. You can get around that by instead replacing a responder with `u: () ^respond(~auxresponses)`.

Preparing for compiling on the Server

I develop the source on a Windows machine and transfer it to a LINUX box. To insure the source does not have carriage returns, I use :clean to read and write all src directory files without carriage returns.

Testing for server presence

If you send the message: null 1 null (that's the null string user id, the string of the character "1" as bot id, and the null string message, the server will send back the string of [you have no user id]with no logging done and minimal load on the server. This constitutes an echo-test to prove the server is running.

Server Crashes & cron

If the server crashes, it may automatically recover, generating the message

Sorry. I forgot what I was thinking about.

If your bot personality assigns a message to the user variable $crashmsg, it will use that instead. One cannot guarantee the server doesn't go down completely, and I recommend it be on a cron job trying to start it maybe every minute or every 5. The system will detect if its port is already busy and not start a new copy if the old one is still running.

Be advised that ChatScript assumes the current directory is the one the executable is in and accesses its data relative to that. For cron this means you want an entry like:

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * * cd /home/bruce/ChatScript;
./ChatScript/LinuxChatscript32 2>/home/bruce/cronserver.log
```

CPU vs IO bound

The ChatScript server consists of a main thread to handle computing a response, a thread to accept incoming connections, and threads for each connection spawned. The connection threads handle reading the message from the user, getting the attention of the main thread to get a response, and then passing that response back to the user. Any logging is normally done from the connection threads, so the main server is free to spend all its effort handling volleys.

Under a full load of users, the main thread will nominally be always busy and the system CPU bound as a consequence. However, the main thread must also read and write user data for each volley. That may slow things down and make the system IO bound if there are lots and lots of user files around and the system is using cloud-based files instead of local ones. This will lower thruput significantly. It can be compensated for by locally caching active user data. There is a parameter to tell the server to track some number of users in memory. It will write out those memory copies periodically, but obviously if the server crashes, some users may be 50 volleys out of date.

Memory Issues with multiple servers on a machine

If you run multiple servers on a machine, you may find one of them fails to start because it runs out of memory. ChatScript allocates all the memory it thinks it needs at startup, so if it succeeds, it won't fail at runtime on a memory

request but will run forever. But if multiple servers allocate too much memory at startup, then a new server trying to startup may fail. You need to configure the memory used on the command line. Go look at the advanced doc on command line parameters (non-server).

Commands affecting the server

Various `:xxxx` commands primarily control/affect the server.

`:show serverlog`

Toggles whether the server is logging data into the server logfile.

`:show echoserver`

If server logging is enabled, this will print the entries sent to the log on the print console as well.

`:quit`

This stops a running server, causing the program to exit. It will first flush any cached user files.

`:restart`

This will force the system to reload all its data files from disk (dictionary, topic data, live data) and then ask for your login. It's like starting the system from scratch, but it never stops execution. Good for revising a live server.

`:crash`

Force a crash to test system behavior during a crash.

`:flush`

If the server is caching user topic data in memory, this flushes all the cache items to disk.

Command Authorization

ChatScript has various :xxxx commands that can be given instead of normal chat input. These remain valid in server mode, and are a security liability if you have potentially hostile users. Therefore all `:xxxx` commands require authorization (even in stand-alone mode). This is the file `authorizedIP.txt` existing in the top level directory. It will be read to validate any `:xxx` command. If the file is absent AND no `authorize=` command line parameter was given, authorization is granted.

The authorization file ships with a default `all`, WHICH IS DANGEROUS TO A SERVER. Normally the file will exist but have its content erased. That will lock out commands.

When you want to issue a command yourself to a running server, you go edit the file to insert on a line your own IP numeric address (since you have access to the file system itself, you are obviously an authorized person). Then you can log in to the ChatScript server and issue a `:xxxx` command.

The file can have any number of lines of IP addresses to be considered legal. You can also enter `L_login` lines, which will match a login name against the user's login. If that matches, that is also authorization. This works best when user names are assigned by a member authorization system, but it good enough in short durations if you use a weird enough login, e.g., `L_b1r8u2c9e0` (bruce interspersed with digits). Example file:

```
129.124.155.1
125.155.156.1
L_master
```

(a user named master case insensitive)

An authorization of `all` allows everyone. See also `authorize=` command line parameter and nodebug parameter.

Command Line Parameters

Either Mac/LINUX or Windows versions accept the following command line args:

Server or Not

`port=xxx`

This tells the system to be a server and to use the given numeric port. You must do this to tell Windows to run as a server. The standard port is 1024 but you can use any port.

`local`

The opposite of the port command, this says run the program as a stand-alone system, not as a server.

`interface=127.0.0.1`

By default the value is `0.0.0.0` and the system directly uses a port that may be open to the internet. You can set the interface to a different value and it will set the local port of the TCP connection to what you designate.

User Facts

Scripts can direct the system to store individualized data for a user in the user's topic file in USERS. It can store user variables (`$xxx`) or facts. Since variables hold only a single piece of information a script already controls how many of those there are. But facts can be arbitrarily created by a script and there is no natural limit. As these all take up room in the user's file, affecting how long it takes to process a volley (due to the time it takes to load and write back a topic file), you may want to limit how many facts each user can have written. This is unrelated to universal facts the system has at its permanent disposal as part of the base system.

`fact=xxxx`

This limits the user to only the xxxx most recent facts created by his interactions. The default is `800000` which is HUGE. It's also meaningless if you don't have scripts that write facts.

User Caching

Each user is tracked via their topic file in USERS. The system must load it and write it back for each volley and in some cases will become I/O bound as a result (particularly if the filesystem is not local).

You can direct the system to keep a cache in memory of recent users, to reduce the I/O volume. It will still write out data periodically, but not every volley. Of course if you do this and the server crashes, writebacks may not have happened and some system rememberance of user interaction will be lost.

Of course if the system crashes, user's may not think it unusually that the chatbot forgot some of what happened. By default, the system automatically

writes to disk every volley, If you use a different value, a user file will never be more out of date than that.

`cache=20`
`cache=20x1`

This specifies how many users can be cached in memory and how big the cache block in kb should be for a user. The default block size is `50` (50,000 bytes). User files typically are under 20,000 bytes.

If a file is too big for the block, it will just have to write directly to and from the filesystem. The default cache count is 1, telling how many users to cache at once, but you can explicitly set how many users get cached with the number after the "x". If the second number is 0, then no caching is done and users have no data saved. They remember nothing from volley to volley.

Do not use caching with fork. The forks will be hiding user data from each other.

`save=n`

This specifies how many volleys should elapse before a cached user is saved to disk. Default is 1. A value of 0 not only causes a user's data to be written out every volley, but also causes the user record to be dropped from the cache, so it is read back in every time it is needed (handy when running multi-core copies of chatscript off the same port).

Note, if you change the default to a number higher than 1, you should always use `:quit` to end a server. Merely killing the process may result in loss of the most recent user activity.

Access to server machine itself

`sandbox`

If the engine is not allowed to alter the server machine other than through the standard ChatScript directories, you can start it with the parameter `sandbox` which disables Export and System calls.

`nodebug`

Users may not issue debug commands (regardless of authorizations). Scripts can still do so.

`authorize="" bunch of authorizations ""`

The contents of the string are just like the contents of the authorizations file for the server. Each entry separated from the other by a space. This list is checked first. If it fails to authorize AND there is a file, then the file will be checked also. Otherwise authorization is denied.

Logging or Not

In stand-alone mode the system logs what a user says with a bot in the USERS folder. It can also do this in server mode. It can also log what the server itself does. But logging slows down the system. Particularly if you have an intervening server running and it is logging things, you may have no use whatsoever for ChatScript's logging.

`Userlog`

Store a user-bot log in USERS directory. Stand-alone default if unspecified.

`Nouserlog`

Don't store a user-bot log. Server default if unspecified.

`Serverlog`

Write a server log. Server default if unspecified. The server log will be put into the LOGS directory under serverlogxxx.txt where xxx is the port.

`Noserverprelog`

Normally CS writes of a copy of input before server begins work on it to server log. Helps see what crashed the server (since if it crashes you get no log entry). This turns it off to improve performance.

`Serverctrlz`

Have server terminate its output with 0x00 0xfe 0xff as a verification the client received the entire message, since without sending to server, client cannot be positive the connection wasn't broken somewhere and await more input forever.

`Noserverlog`

Don't write a server log.

`Fork=n`

If using LINUX EVSERVER, you can request extra copies of ChatScript (to run on each core for example). n specifies how many additional copies of ChatScript to launch.

`Serverretry`

Allows `:retry` to work from a server - don't use this except for testing a single-person on a server as it slows down the server.

No such bot-specific - nosuchbotrestart=true

If the system does not recognize the bot name requested, it can automatically restart a server (on the presumption that something got damaged). If

you don't expect no such bot to happen, you can enable this restart using `nosuchbotrestart=true`. Default is false.

Testing a server

There are various configurations for having an instance be a client to test a server.

`client=xxxx:yyyy`

This says be a client to test a remote server at IP xxxx and port yyyy. You will be able to "login" to this client and then send and receive messages with a server.

`client=localhost:yyyy`

This says be a client to test a local server on port yyyy. Similar to above.

`Load=1`

This creates a localhost client that constantly sends messages to a server. Works its way through `REGRESS/bigregress.txt` as its input (over 100K messages). Can assign different numbers to create different loading clients (e.g., load=10 creates 10 clients).

`Dual`

Yet another client. But this one feeds the output of the server back as input for the next round. There are also command line parameters for controlling memory usage which are not specific to being a server.

CS as an embedded Client

Building a web-based interface to ChatScript is different from building an app involving ChatScript. In the web-based version, the ChatScript engine will always be a server on a reasonably powerful machine and have available the full resources of the web. In building an app-based NLP application with ChatScript, there are two general architectures, embeed-based and server-based. Embedded based is described in the External communcations manual.

Server-based

The other app configuration always requires an internet connection and communicates with a ChatScript server over it. w addition, this means it can easily to voice-to-text over it and text-to-voice back. All log files are already on the

server so complete access to all conversations is conveniently available and all ChatScript data files are also easy to update as desired.

Of course the requirement of having an internet connection available is a severe one on the client and may also mean his app is generating data charges he may not like.

You still have a client app which sends and receives the communications (voices and/or texts) and must still decode return text from ChatScript to retrieve app commands for avatar behavior and other functions.

If you are using voice-to-text on the server, then you would create a server app that receives inputs from the client, communicates with the voice-to-text server, then hands appropriate text over to the ChatScript server.
The server app, on receiving the ChatScript output, may need to send it over to a text-to-voice conversion, and then ship all results back to the client. If you are not using voice in any capacity, the client app could communicate directly with the ChatScript server.

RESTful Server

ChatScript normally is not RESTful. It saves user state between volleys. But if you are scaling using multiple CS servers, you either need to arrange for routing specific users to specific CS servers where their files are, or have a central fileserver or database where user files are kept. And for proper full-scale service, you need a database that has multiple copies of data (like DynamoDB) that does not depend on a single machine. Or, your use of CS must be RESTful, meaning no user state is actually kept on the server side. This is partly possible. The data kept in the user topic file is:

1. prior things recently said by user and chatbot
2. user facts
3. state of every topic – which rules have been used up, changed topic flags
4. current topic & current interesting topic stack
5. input rejoinder tag
6. user variables & turn number and random seed
7. user context data – recently executed rule tags and on what volley

You can direct CS not to save ANY user data by turning on user caching, but setting the cache count to 0. Now you have a system that is RESTful, but can't keep any state. You can, however, send out state via out-of-band messages and read back in state the same way.

Currently there is a limit of 20k for an output message and user topic files vary in size including up to 40K (and you can allow them to be bigger via command line option). So it is not really practical to transmit all user state back and forth as oob data. Here is what is reasonable to accomplish:

1. Dont save most recent messages. You cannot detect the user repeating themselves automatically, though if you wrote out the most recent message you could manually check for immediate repeat.
2. Maybe write out user facts, depending on how many.
3. Don't change topic flags on the fly (so don't write them out). Don't write out the state of all rules (so you can't detect repeat or used up responders) but do write out the deepest gambit of a topic used. From that on start of a volley you can manually turn off all preceeding gambits.
4. Write out the current topic (for sure) and the interesting topic stack if you care.
5. Write out the rejoinder tag
6. Write out any user variables you care about. You may not care about the random seed or turn number, or you can write them out as well.
7. Currently there is no interface to context data so you can't write that out. Generally all this will be written out by a post-process phase.

Encryption

ChatScript normally reads and writes everything in plain text. If you need greater security then you need to do several things.

1. turn off server logs and user logs
2. Use encrypt= and decrypt= command line parameters.
3. Use default system or roll your own encryption in privatecode.

Encryption applies to user topic files and long term memory files (^export and ^import). The built-in encryption method calls an external api server using json to perform encryption and decryption, passing the server urls provided by encrypt= and decrypt= command line parameters. The routines will add in the user's login id if you use %s in the command line parameters. Eg.,

encrypt=http://someapi/someotherdata/%s decrypt=http://someapi/%s/something

Where %s goes depends on the api you call.

To roll your own, you have to define routines in private code that encrypt and decrypt (see examples in os.cpp) and on private initialization override the default variables controlling userFileSystem.encrypt and userFileSystem.decrypt.