GRASS documentation

Simon Le Bail-Collet, Lukas Gelbmann, Björn Gudmundsson

May 2019

In the following, we're going to focus mainly the particular points that may be different in our implementation than in others.

1 Protocol

To let the client know when the server response is done, we end every server transmission with an end-of-transmission byte (0x04). We do this even if the response is empty. This would cause incompatibilities if we were to use a different client with our server or vice versa.

2 Architecture

We used C sockets api in order to make the client and server communicate together. The client first spawns a socket that will try to establish communication with the server. If the server accepts the incoming connection it itself spawns a new thread which will the main communication link with this client.In particular, in the get command the server spawns a new socket that will send the file's bytes to the client.

For this project we used the classic client/server architecture. The server will listen for incoming connections on a socket and when a client wants to start a session on our platform the server will start a new thread that will open up a port and will listen for incoming commands from the client. The commands are then parsed and passed to a commands API. The methods stated in the API are the same as given in the specification of the GRASS protocol. The structure of the program is as follows: All of the headers of the program are listed in a directory called include and those headers are split into server and client headers and headers that are shared between both implementations of the server and the client.

A client session in our implementation is identified by a connection object. The connection object is unique per user and session and keeps track of the state of an active session. An example of how the connection object is used is in how we implemented the traversal from one directory to another. One of the instance variables variables in the current relative path of the session. It keeps track of where the user is located in the shared directory, relative to the base directory in the program without having to perform cd commands on a per-process basis. Every command the user issues is then calculated relative to the users current directory.

An important issue that we had to tackle was implementing such that multiple users can be using the service concurrently and that the behaviour of the system should remain consistent while a user is logged in. An example of such a problem would be when a user would like to issue an rm command and remove a directory or file from the shared repository. A user may have traversed into the directory or a child directory that another user would like to issue an rm command on. To solve that problem there are two independent global data structures, UserReadTable and FileDeleteTable. These data structures are shared between all active sessions and keep track of which directories currently have users that are either residing in one of its child directories or reading a file in one of its child directories. If an rm command also takes a significant amount of time then a user should not be able to either read the contents of the file or traverse into one of its child directories. For that reason we also have the global data structure of entries that are being deleted. When a user logs out or exits from the system the connection object will make sure that the user activities and locations within the system have been cleaned to further prevent it from preventing future actions from clients that are currently using the system.

3 Development

For the development of this protocol we used a test-driven development style. A lot of test cases that tested out different functionalities of the protocol and various edge cases and worked on the project with the tests in mind. The tests were written in python and use regular expression to determine if the outputs from the server match the expected values that were determined by the tests.