Protocol

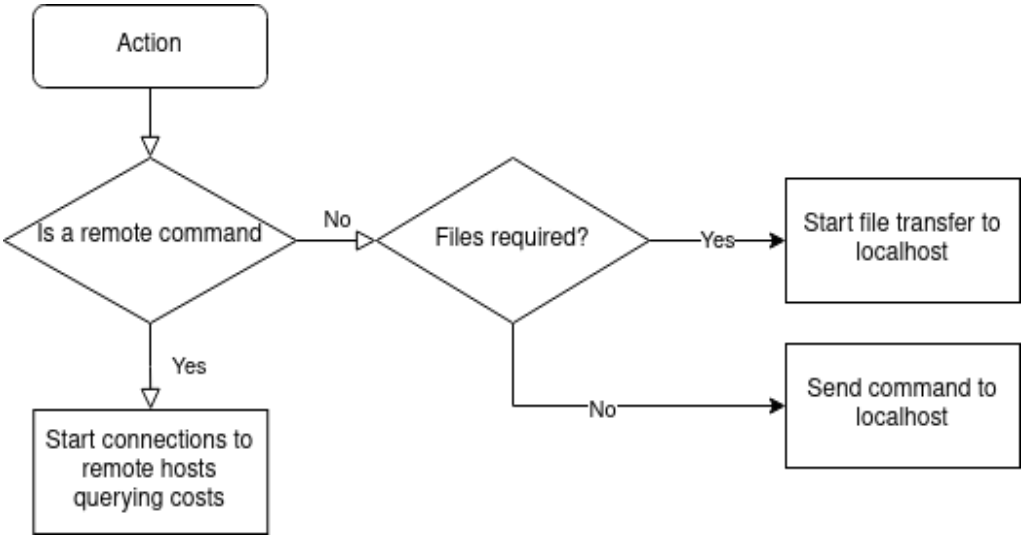
The protocol uses variable length messages with the sender specifying how many bytes are to be sent in the payload and how many bytes are in the header as the header contains optional fields. The protocol is an extension from the python sockets tutorial at https://realpython.com/python-sockets/. It uses the Message class to contain all the methods and structure of encapsulating the payloads between the client and server. The message is composed of three parts. The first two bytes, the protoheader, are a 16-bit unsigned short integer in network (big-endian) format, containing the number of bytes in the second part, the jsonheader. The jsonheader contains the following required fields:

1. Byteorder: little or big endian of the system
2. content length: the number of bytes in the payload
3. content type: the type of request, one of three
4. content encoding: the encoding of the payload

The client or server can add additional information in the jsonheader depending on the type of request. These include the filename for a file transfer, the output and exit status of a subprocess from the server, and a keep_connection_alive field to keep a socket open to be reused.

The client does most of the heavy lifting in the protocol and after parsing a rakefile will store the hosts and port numbers in one variable and the actions in another.

There are only three types of requests that the client can send, and that the server can handle.



1. Query: to ask a server a cost of performing an action
2. Binary: to send a file
3. Command: to spawn a subprocess on a server to do an action, in this case remote-compilation but other commands would work too

The client maintains 5 buffers and an integer variable to keep track of which action it is at.

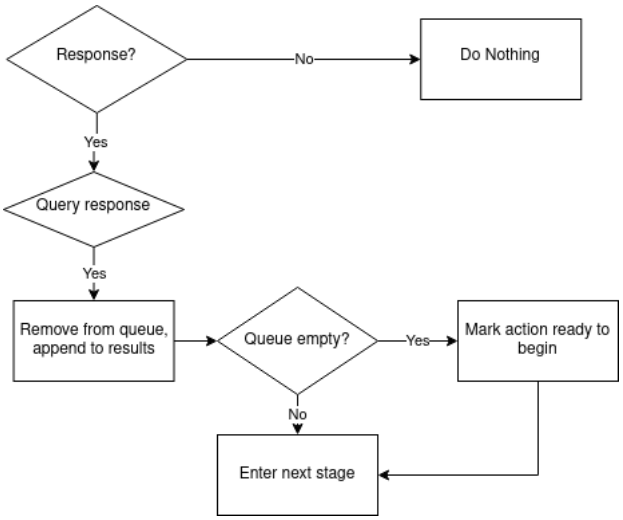
1. Queue: a buffer to monitor sockets for which the client has sent a query request to
2. Queries: a buffer to store the results of the queries for the current action
3. Requires: a buffer to store the files to send
4. ready_to_begin: a Boolean buffer to monitor which action/s are ready to begin, if an index is marked true, the action at that index is ready to begin
5. alive_connections: a buffer to maintain which sockets the client is waiting on a response for a server to remember what action a returning socket may be for. A socket 5 at index 3 means that action 3 has an existing connection on socket 5.

Each action set in the rakefile is a separate event loop and a failure of any action in an actionset will terminate the entire program. This is by design as later action sets may depend on files one previous ones which would result in failure.

The client will loop through each action set and check if it contains “remote”. If not, it will send the action straight to the localhost. If it contains remote, the client will send a query to all hosts and store the socket numbers for each of the connections the queue. It

will then monitor those sockets using select and process these events when they return. When the client receives a response from a server, it will first enter its first conditional to check if it is a server response for a query request. If it is, it will append those results into the queries buffer and remove that socket number from being monitored in the queue buffer. There is some error handling if the client is refused a connection, it will remove that socket from the buffer and continue with the remaining hosts.

The client will continue in this way and wait for all queries for an action to return at which point the queue will be empty and then the client will be ready to start the action. It does this by marking which action is ready at the index of the action in the ready_to_begin buffer to true and increments the queried_n. At this point, it will also start queueing the next action in the _remote_start() method as the current socket being processed will enter the next control loop. The client will close the connection for any returning query requests.

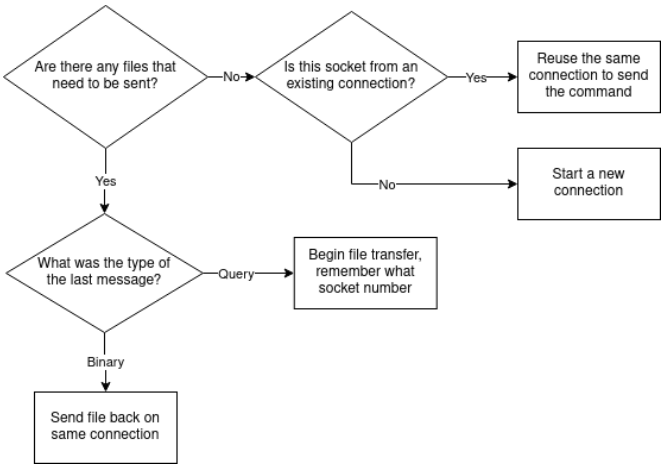


In the second stage, the client checks if the last socket to be returned is for a previous request or if it is for an action which is ready to begin. It stores this in the action_n variable to remember which action the current socket is for.

If the socket is the last one to return from a query, (the queue is empty and the action has been marked in the ready_to_begin buffer), it is the first request of the action, and the client will pick the lowest bid from the queries buffer. It will first check the requires buffer to see whether the current action has any files that need to be send. If so, start a connection to the server to send the first file.

The client will pass the keep_connection_alive in the header as true by default which will tell the server not to close the socket. This is the default for sending a file as one of the assumptions of this protocol is that a command will always be sent to the server after it has received all the files and this command will tell the server to close the socket then.

The client will remove each filename from the requires buffer and send it in the header to the server until the buffer is empty. The server will create a temporary directory for each connection and store all the sent files there to isolate the connections from each other. As such, each action has it's own temporary directory on the server and will not be affected by similar files for other actions.



When the requires buffer is empty for an action, the client will reuse the socket for the last time to send the command with the keep_connection_alive header set to false. It will process other events until the server responds, which will return a compiled program (or the output of any other command).

On the server, a subprocess is spawned and if a compile request is received, will proceed with the compilation. The newest file is searched for in the temporary directory for that socket and sent back to the client. When the client receives the response, it will check that the subprocess exited successfully (the exit_status, any error_messages and filename are added to the header) and write the new file. That concludes one successful action and the client loops through the process until all the actions in an action set are complete before starting a new event loop for any subsequent action sets.

Walk-through

Take an example rakefile as follows:

```
HOSTS = 5.27.234.136 87.98.3.198:5000 29.229.171.57

PORT = 8000

Actionset1:
        remote-cc –c func1.c
                requires func1.c
        remote-cc –c func2.c
                requires func2.c
Actionset2:
        remote-cc –c program1.c
                requires program1.c
        cc -c program2.c
                requires program2.c
Actionset3:
        remote-cc –o program3 program2.c func1.c func2.c
                requires program2.c func1.o func2.o
```

The client will assume port 8000 for the hosts without ports specified and for localhost as well. The client will then send queries to all three remote hosts requesting a cost. Since this is the first action, the client must wait for all queries to return. When all the queries return, the client will send out queries for action 2 immediately and then continue in the same iteration to pick the host with the lowest bid, say host 2 at `87.98.3.198:5000`, to continue with action 1.

The client will first send func1.c to host 2 before the select loop is entered again. The select loop will reset with three new queries and a file transfer response, ready for the client to process. It will process the queries as in action 1 and once completed with that, send the file transfer for action 2 to the lowest bid, say host 1 at `5.27.234.136:8000`. It will then process the response for the previous action, which was the file transfer to host 2. After the client processes the file response from host 2, it will reuse that socket to send the cc command to compile file1.c to host 2.

Now, the client is awaiting a file transfer response from host 1 for action 2 and a command response for action 1. When it receives from host 1, a response that the file was transferred successfully, it will reuse that connection to send the cc command to compile file2. It will then process the command response for action 1, if the compilation took a reasonable time, and have successfully received func1.o for action 1 and be waiting for the command response for action 2. Similarly, upon a success response from host 1 for the second and last action of action set 1, the client has completed actionset1 and will terminate the event loop.

The client will then begin action set 2 with a new selector and continue as it did for the previous action set. When it reaches action 2 without the remote command prefix, the client will not send out any queries and instead just sent program2.c to the local server and reuse the same connection after to send the compile command. On successful completion of both actions, the client will have completed action set 2 and can begin the final action set.

The client now has func1.o, func2.o, program1.o, program2.o on its filesystem and will start the last event loop just like before. Querying all the hosts, waiting for the responses and starting a new connection to the lowest bidder. It will send all the required files, followed by the compile command and if no errors were found, will receive program3 back on its filesystem.

Conditions

The conditions in which compilation and linking would occur faster using remote compilation compared to local are when there are many actions that must be completed first. For example, if ten actions were required in an action set, on a local machine they would have to be done sequentially but for the remote application, if action two took a long time, the client could have sent actions three and four to other hosts and wait for the responses. Remote compilation would be faster if large number of files required to be linked together. Generally, the greater the number of actions and the computationally longer actions there are, the more benefit would be observed as compared to local compilation in which a long action would block the computer until it was finished. There are some situations where remote compilation might even perform worse, such as if the files sent were large such that the time to send them was longer than the time to compile them at which point it would be quicker to compile on the local machine.