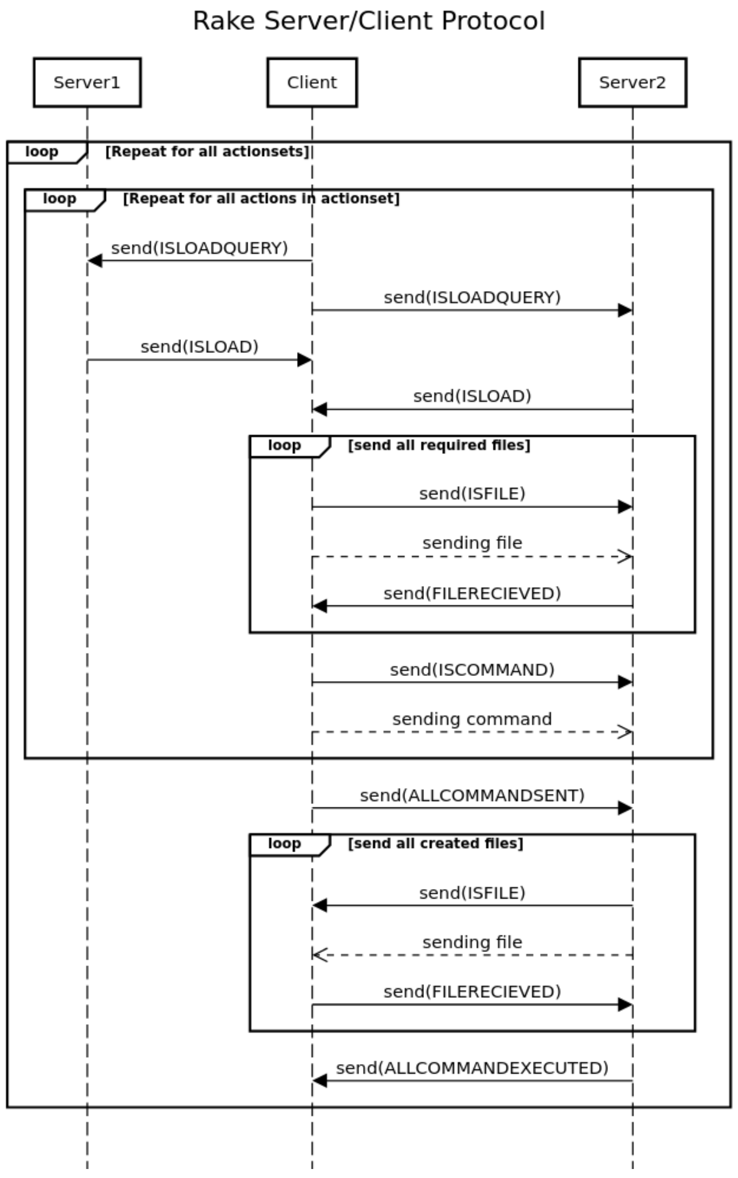
CITS3002, 2021 Project Report



This diagram depicts a typical interaction between a rake client and two rake servers. The following section will provide a more detailed walkthrough of the execution sequence to compile and link a multi-file program.

Software walkthrough

Example rakefile:

```
PORT  = 4321
HOSTS = 127.0.0.1 192.168.0.99:6543

actionset1:
    remote-cc -c func1.c
            requires func1.c

actionset2:
    remote-cc -o program2 func1.o func2.o
           requires func1.o func2.o
```

A note on instructions:

All instructions are integers predefined between the server and the clients. They range from 1 to 8, all representing a unique instruction

Step by step walkthrough:

I am omitting the steps to parse the rake file and connect to the hosts.
All integers to be sent and received by the client and server are 32-bit integers (4 bytes) in network byte order (big-endian).

First, the client will send an ISLOADQUERY instruction to all servers connected. In the context of this project, the server's load is simply the number of commands a server has already received.

Before both servers have replied with their respective loads, the client will not attempt to send anything else to the server.

The servers will each send their respective loads. The client will compare the received loads and store the socket descriptor of the server with the lowest load in the variable lowest_load_socket.

Only after all servers reply with their respective loads will the client now move on to send the first command in the first action set of the rakefile to the lowest_load_socket. In this case, the command to send is "cc -c func1.c", the "remote-" prefix is cut off during the rakefile parsing phase, and a boolean is set true to inform the program this command is intended to be executed remotely by a server.

The client will notice that this command requires files. Thus, the client will first enter a loop and send all required files to the server.

The client will attempt to send "func1.c" to the server, assuming this file exists in the current working directory. The process of sending a file is:

1. Send ISFILE to the server to inform it that a file is on its way.
2. Send the size of the file's name (7 for "func1.c").
3. Send the file's name ("func1.c") as raw bytes.
4. Send the size of the file.

Notice that steps 1 to 4 are the client sending the header for file transmission to the server. It contains all the information necessary for the server to receive the file.

5. Send the contents of the file as byte chunks.

The client will now wait indefinitely for the server to respond with FILERECIEVED.

The server receives the file in the same order:

1. The server receives 4 bytes of data (4 bytes because instructions are 32 bits integers), recognizes the instruction is ISFILE and prepares itself to receive a file.
2. The server receives 4 bytes of data, which will be the size of the file's name, and stores it in a variable file_name_len.
3. The server now receives file_name_len bytes of data, which should be the file's name.
4. The server receives 4 bytes of data, which will be the size of the file being sent, and stores it in a variable, file_size.

Notice that steps 1 to 4 are the server receiving the header necessary for file transmission.

5. The server receives file_size bytes of data and writes it into a file with the name received from step 3.

Once the server receives all bytes sent by the client, the server will send FILERECIEVED to the client to inform it that the server received the file successfully.

Once the client receives FILERECIEVED from the server to acknowledge that the file has been successfully transmitted, the client will now transmit the actual command. The process of sending a command is:

1. Send ISCOMMAND to the server to inform it that command is on its way.
2. Send the incoming command size (13 for “cc -c func1.c”).

Notice steps 1 to 2 are the client sending the header for command transmission to the server

3. Send the command.

The server receives the command in the same order:

1. The server gets ISCOMMAND and prepares itself to receive a command.
2. The server receives 4 bytes of data, which will be the size of the command, and stores it in a variable, command_size.

Notice that steps 1 to 2 are the server receiving the header necessary for command transmission.

3. The server receives command_size bytes of data into a variable, decodes it according to UTF-8 standard, and stores the decoded command in a list (ALL_COMMANDS)

The client, having sent all commands in actionset1, will inform the servers that it has sent a command to execute all commands received in parallel by sending the instruction ALLCOMMANDSENT.

Upon receiving this instruction, the server will execute all commands received and stored in ALL_COMMANDS in parallel. If any files are created, the server will first send all newly created files to the client.

The ways servers send the created files, and the client’s steps to receive them are the same as when they send files to the server. Some details, such as how data is packed and unpacked, are changed. However, the steps remain the same and thus will not be elaborated on in this report.

Finally, after sending all created files, the server will send either ALLCOMMANDEXECUTED if no error was encountered when executing commands received, or, if an error occurred, the server would send FAILEDCOMMANDEXECUTION instead.

The client, if received ALLCOMMANDEXECUTED, will repeat all the above steps for the next actionset (actionset2). If FAILEDCOMMANDEXECUTION is received instead, the client will close all sockets gracefully and terminate itself.

Conditions when remote compiling and linking are faster

Remote compiling and linking never appeared to be faster than compiling and linking using just the local machine.

The theory is that because the files are being compiled in parallel remotely, they should be able to compile and link faster. However, with the steps in between, especially time taken to transfer the files, acts slow down the time saved from compiling and linking by multiple computers in parallel.

Perhaps it will only be noticably faster if the local computer is,

1. Has slow computation speed.
2. Has fast read and write speed.
3. There are a large amount of files to compile and link.
4. Remote machines have fast computation speed, and fast read and write speed.