CITS3002 Computer Networks Project Report

22966841 - Kristijan Korunoski
22904833 - Amandeep Singh
22723279 - Callum Paterson

For this project we implemented a stop and wait protocol. Once a connection has been made between server and client, the client sends the first data (Usually a cost query for servers) and waits for an acknowledgement. In our implementation of this protocol the acknowledgements usually carry extra data that is used by the client/server. If any actionset fails then the client and server closes indicating failure.

For an example rakefile:

```
# A typical Rakefile

PORT  = 6285
HOSTS = 192.168.0.95

actionset1:
    echo starting actionset1
    remote-cc -c square.c
        requires square.c allfunctions.h
```

First the client will interpret the lines in this files, storing the default port number, and an array of hosts. For empty lines or comments (beginning with #) these lines are discarded and all other lines are stored. The client recognises a line with no tab character at the start as an actionset and will send a cost query to all servers simultaneously, it will then wait for all servers to send back an acknowledgement carrying an integer representing the cost.

Once the server, in this case only one server is available, receives the cost query it sends back the acknowledgement, carrying the cost of operation.

The client then chooses the server with the minimum cost to complete the actionset. In the actionset lines beginning with remote- are sent to the remote server for completion. Lines with two tab characters represent files required for the above action. In this example the first line “echo starting actionset1” is not a remote action and so the client completes this process on the localhost machine.

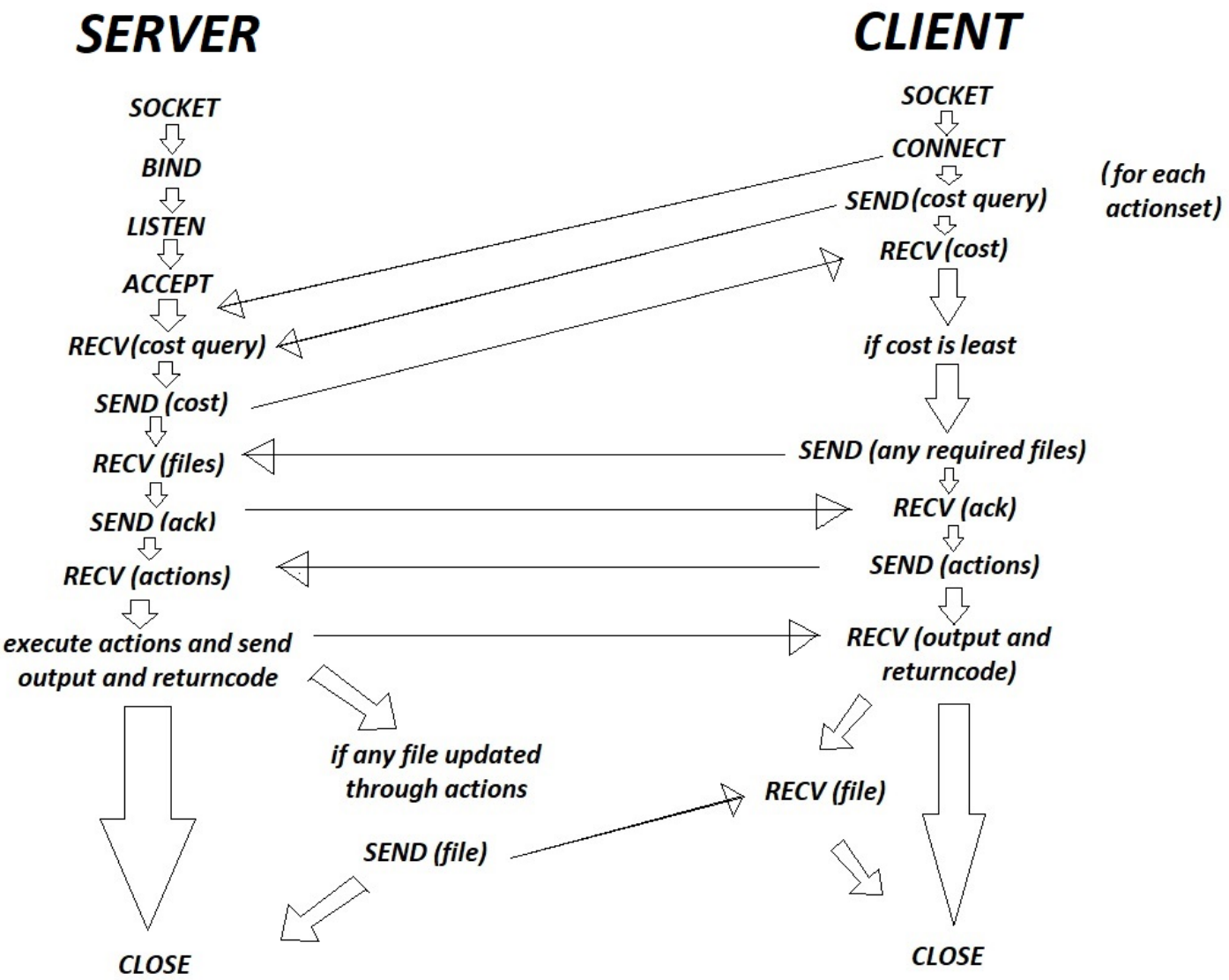
The second line, “remote-cc -c square.c” Is a remote action. The client checks if the next line begins with “\t\trequires”, as this means files must be sent to the server. In this example the next line does begin with “\t\trequires” so the client sends a file transfer packet indicating that the client is requesting a file transfer. The server then sends back an acknowledgement telling the client it is ready. The client then checks how many files are required to be sent, in our example there are two. The client sends a packet with the number of files to the server. The server then sends back an acknowledgement indicating it has received the number of files to be sent and begins a while loop, staying open until the number of files received is equal to the number of files required. The client then begins a loop itself to send each file individually. The filename is read from the rakefile and we locate the file on the client machine and check it’s size and store it as an integer. The client sends the filename and waits for an acknowledgement from the server, on the server side the sever opens a new file in write binary mode with the name of the filename sent in a temporary directory. The client then calculates the number of times the server will need to receive data as each packet is a maximum of 1024 bytes long. Once the client has calculated how the file is fragmented it sends to the server the number of times it will need to receive without sending back an acknowledgement. The server receives this number and sends an acknowledgement back, starting a loop until it has received data the amount of times specified. The client opens the file in read binary mode and sends the content to the server and waits for the acknowledgment that the file transfer completed. On the server each time a 1024 byte message is received it is written to the created binary file and this loops until the number of sends specified earlier has

been reached. Once the number of sends is reached the server sends an acknowledgement back to the client indicating the entire file has ben received. This process loops until all files have been sent to the server.

Once all files are received by the server the client sends the action line “remote-cc -c square.c” and awaits an acknowledgement that the line was received. The client then waits for another acknowledgement either the process completed stout and return code or a request for file transfer back. On the server side, the server receives the line, and sends the acknowledgement. It then completes the subprocess, here the file square.o will be compiled. The server checks if there has been a new file created, in this case there is a new file and so it begins the file transfer back to client process. The file transfer back is the same process as transfer to server. The final acknowledgement from the server will return the stdout and return code.

Once all of the actions in the rakefile have been completed a “\EoR” packet is sent to all the servers indicating that all actions have been completed.

Importantly, completing actions on a server increases it’s cost, hence a server that was used for the first actionset in a larger rakefile may not be used for the second.



^— A diagram representing communication between client and server visually.

Remote compilation and linking appears to perform better than using the local machine when there is a strong connection between client and server and there is a fast transfer rate. The remote compilation may also perform better when the local machine has other tasks executing which are using a large portion of memory. Remote compilation would also be extremely useful for an operating system that does not have all the dependencies required to compile and link files, although this seems unlikely.