CITS3002 - Networks Project

*Jamie McCullough **22238418** and Cormac Sharkey **22983427***

For our client-server protocol, communications are undertaken through used of a fixed size, 64 byte, UTF-8 header. The header is comprised of up to three components: communication code; option flags; and length of the message to be sent (payload). Payload is then sent to the recipient, with it's data expected to match the header-described attributes.

Communication Codes

`DISCONN_MSG` (!D)

Client: Sends to server once it has finished with it.
Server: Cleans up data structures for client.

`COMMAND_MSG` (!C)

Client: Sends this with a payload of command to execute.
Server: Executes payload, captures exit state, stdout, stderr, and generated files, passing them to client.

`REQUEST_MSG` (!R)

Client: Sends this to indicate to the server that it will need to receive files.
Server: Prepares to receive files from the client.

`SUCCEED_RSP` (!S)

Client: Indicates success, keeps processing packet and Rakefile.
Server: Sent to indicate execution was successful (exit code == 0).

`FAILURE_RSP` (!F)

Client: Indicates failure, receive two packets, one containing the stderr and one containing the exit code.
Server: Sends this to indicate execution was unsuccessful. Detect based on exit code.

`EXECUTE_GET` (!E)

Client: Sends this to get current amount of requests server is servicing
Server: Sends active thread count to client, effectively the load of the server.

Option Flags

`STDOUTP` (S): Set if the packet contains standard output
`INCFILE` (I): Set if files to be sent back to client for it to prepare to receive a filestream.
`FILETRN` (F): Set only on filestream packets

Message Length

Last bytes indicate the length of the payload. This value is padded out to be the length of the remaining bytes in the header. i.e. Message sent with code of `!E` would be padded by 62 bytes of white space, bringing the length to 64 bytes total.

Filestream Protocol

Filestreams are multiple files to sent and received between the client and server. Each consists of a metadata packet, packets for each filename, filesize, and for transfer confirmation. Filestreams use these codes to replace the first two bytes of "options", with last byte of options, set to `F` to indicate it is a filestream. Meta packets are have all three options set. i.e `!S` + `SIF` + `length` + `padding`

```
FILENAME    = "!N"  FILETRAN    = "!T" FILESIZE    = "!Z"
```

Header: `!S!NF7[57 spaces]` **Payload**: `Dog.txt`

Example: `dog.txt` and `feline.txt`.

```
|--------!R--------->| Indicate requirements
|------!SSIF2------->| Client sends metadata packet showing 2 files to be
sent
|<-------!N----------| Server asks for name of first file
|------!S!N7-------->| Client sends header indicating 7 byte filename
|------dog.txt------>| Client sends payload of filename
|<-------!Z----------| Server asks for filesize of first file
|------!S!Z2-------->| Client sends header indicating 2 byte file size
|--------34--------->| Client sends payload of file size
|<-------!T----------| Server requests transmission to start
|-The dog or domes..>| Client begins transfer of file.
Server receives based on filesize and saves to file based on received
filename
[Repeat steps 3 onward for feline.txt]
```

Execution Walkthrough

```
HOSTS = host1 host2

actionset1:
  remote-cc -c func1.c
    requires func1.c
[...]
```

1. Client parses the rake file then looks at the actionset.
2. `remote-cc -c func1.c requires func1.c`
    - Polls host1 and host2 by sending `!E` (plus payload length).
    - Receives from host1: `!E1[padding]` and `0`
    - Receives from host2 `!E1[padding]` and `0`
    - Neither busy, picks host1, sees `func1.c` is required for command
    - Sends `!R` + payload length, then sends filestream of `func1.c`
    - Sends `!C` + payload length to server, with payload `remote-cc -c func1.c`
    - Server executes command, detects output to send
    - Client doesn't wait, instead adds host1 socket to watchlist, incrementing command sent count.
3. Until commands waiting = `0`, `select()` watchlist, see host 1 is ready:
    - Get header and payload `!SSI 0[padding]` and nothing (no stdout)
    - `I` set, so pass socket to filestream handler and receive `func1.o`
4. Repeat this process for other actionsets (unless a command failed, then terminate).

Remote Compilation Performance

Remote compilation performs better under these conditions:
(1) The server has a faster compilation time than the client. Perhaps the client is a small, embedded device, and the server is a proper workstation.
(2) Performance difference is larger than performance penalty incurred by sending over a network. If the network is slow, it can take more time to transfer requirements than that saved through a better CPU.
(3) Large complex compilations can be spread across an arbitrary amount of servers, a huge task which can be executed in parallel would perform better remotely than locally.