Cluster Shell

IS F462: Network Programming

Assignment 1

Bir Anmol Singh
Guntaas Singh
K Vignesh

1 Objective

In the given problem, the goal is to extend the shell implemented in problem 1 to a cluster of machines, so that each node in the cluster can remotely issue commands on the other nodes, and observe their output. Additionally, commands can be broadcast to all nodes, and the output of commands can be piped across nodes.

2 Design

2.1 Client Server Architecture

The design of the network application follows client-server architecture. Each of the N nodes in the cluster execute the client program, while the server program executes on a separate host. This simplifies the client design. Also, the name-to-IP mapping can be restricted to the server, making it easier to update the mapping and scale the cluster size.

The client application parses commands and sends a **request** to the server. Each request represents a response expected by a client node. In order to service this request, the server needs to contact other nodes in the cluster (clients) and request them to execute commands. Each request made by the server is a **job**. Since, the application supports broadcast messages and pipelined commands, a request may contain multiple jobs. The server iteratively forwards these jobs, and returns the response for the last job in the request to the client which originally issued it. In order to do so, the server must maintain state for each request it receives.
In order to maintain shell state such as working directory, Each client forks and spawns separate shell instances for servicing requests originating from each node. This ensures that commands issued by node A and node B to node C, do not interfere with each other, or lead to unexpected changes in shell state.

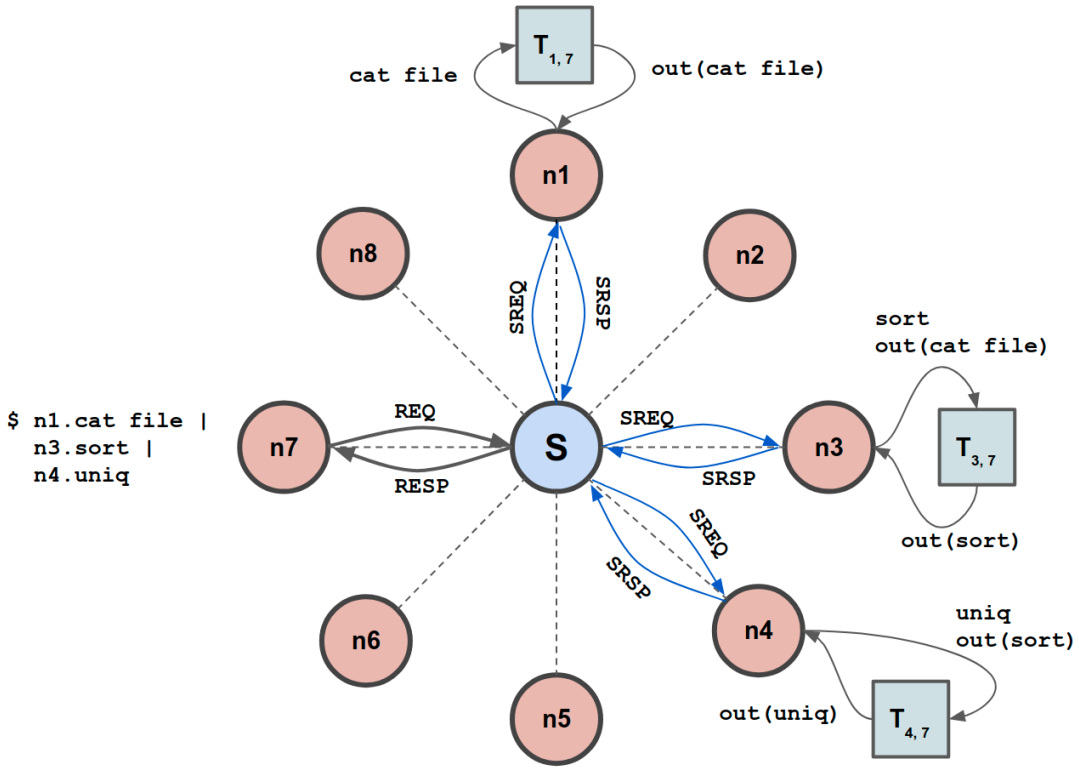

Fig. 1. Execution of a pipelined command issues by node n7, and associated exchange of client-server messages. The blue node represents the server, red nodes represent clients (cluster nodes) and green nodes represent instances of the shell (implemented in P1) spawned by clients.
The client at n7 parses the command and sends a request as a REQ message to server. The server forwards the three jobs in the request iteratively to the nodes as SREQ messages. Each node executes the job using an attached shell instance, and returns the output as a SRSP message. The server returns the output in the SRSP message corresponding to the last job to n7 as a RESP message.

2.2 Message Protocol

A protocol must be established for the clients and server to effectively communicate. In total, the application makes use of six types of messages. The formats for these message types are specified in Figure 2. REQ, SRSP and TCLS are client-to-server messages, while RESP, SREQ and CLOS are server-to-client messages. The semantics of these messages are described in Figure 1 and Section 3 (Shell/Client Termination).
To handle broadcast messages, the server expands the single job in the request to multiple jobs for conveying the command to all active nodes. For handling the *nodes* command, the server directly generates and returns a RESP message by observing all nodes having an open connected socket.

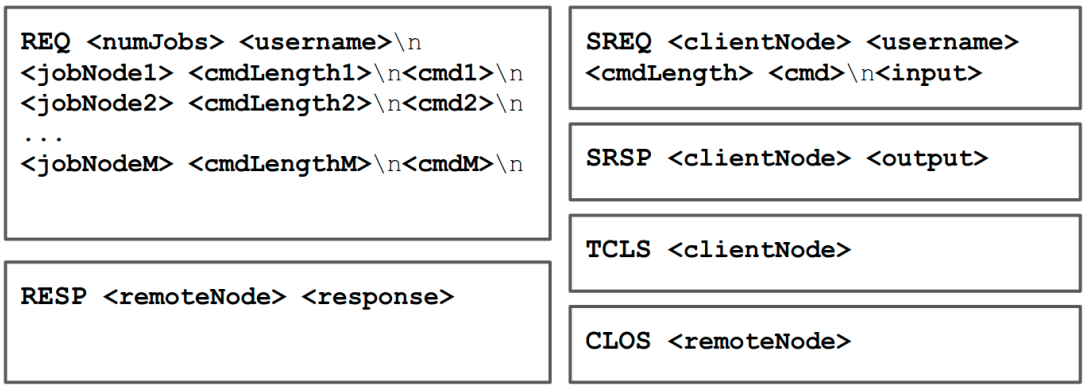

Fig. 2. Text encoded message formats. Each message is prefixed by its length in bytes for the purpose of framing.

2.3 I/O Model

Both - the client and the server program need to monitor multiple file descriptors simultaneously to see if I/O is possible on any of them. Each client monitors a socket for messages from the server, and descriptors corresponding to each shell instance for output. The server monitors sockets for each client, along with a listening socket for new connections. Therefore, **I/O Multiplexing** is a natural choice. It allows us to manage multiple descriptors efficiently, without the buffer management overhead of non-blocking I/O, or the inter-process communication that would be required for using multiple processes or threads.

3 Implementation

Transport Layer Protocol: We use the TCP sockets for building the application, instead of UDP. This choice is motivated by the reliable data transfer service provided by TCP, simpler state management and message exchanges on the server-side.

Framing and Encoding Messages: TCP sockets provide a reliable byte-stream for exchanging data. Consequently, message boundaries are not preserved and must be identified explicitly. Towards this end, for each message type, each message is prefixed with its length as a 32 bit integer in network byte order, and is interpreted by the receiver to identify message boundaries. The messages are expicitly text encoded by the sender and decoded by the receiver through tokenization. This arrangement necessitates that messages be completely buffered before being sent out - imposing limits on the size of commands and outputs. Using sufficiently large buffers allows us to handle most commands correctly.

I/O Multiplexing: The *select()* call was used to implement an I/O multiplexing based solution on account of portability and ease of use. *epoll()* may present a more scalable solution. However, the gains may be negligible when the number of nodes is small.

Parsing Commands: Commands made by the user at a node have to be parsed into a list of jobs for the server to process. This is done at the client side itself and the request is encoded as an REQ message sent to the server.

Shell Communication: The client process communicates with shell instances using pipes and FIFOs. Pipes are used for reading output produced by the shell, and are monitored using select(). Pipes, however, do not suffice for passing commands and input to shell instances. Commands like *wc* and *echo* expect variable length input, and only stop reading on receiving EOF. In order to generate EOF while still maintaining control of the shell instance, FIFOs are used. The write end of the FIFO can be closed to generate EOF, and later reopened to convey more commands to the shell instance, when required. The shell instance pauses and waits for the *SIGUSR1* signal, sent by the client process on reopening the write end. This helps avoid busy waiting by the shell process.
The shell program reads all input provided to it at once. It also outputs prompts indicating the current user and working directory to *stdout*. To facilitate simpler interaction between the client and shell processes, the shell is executed in a special mode in which prompts are not printed and the shell accepts the length of the command in an encoded form before the command itself. This ensures that the shell process only reads the command, while the input attached to it is read only by the child process spawned by the shell to execute the command.

Execution of local commands: Commands having no node-identifier prefix have to executed on the node at which they are issued. A separate shell instance is maintained for locally executing these commands, without exchanging any messages with the server.

Shell/Client Termination: If the shell instance for node x or the parent client process (node n) terminates, a TCLS message is sent by the client to the server. If node n was servicing a pending request made by node x, the server sends a RESP message to the client for node x, indicating the same.
Moreover, other nodes may have spawned new shell instances for servicing commands from (now closed) node n. These instances should be closed to avoid unnecessary overhead, and to present any future connections made by node n, with fresh shell instances. Hence, when the server receives FIN from a client (indicated by reading 0 bytes on the connected socket), it sends a CLOS message to all active nodes, which then terminate any shell process created for node n. No TCLS messages are sent to the server in this case, since node n is inactive.