

Distributed C Compiler

Zarana Parekh*, Saumya Bhadani[†] and Shreyansh Prajapati[‡]

Information and Communication Technology

Dhirubhai Ambani Institute of Information and Communication Technology

Gandhinagar, India

Email: *201301177@daiict.ac.in, [†]201301100@daiict.ac.in, [‡]201301062@daiict.ac.in

Abstract—Distributed computing over a network can be used to speed up the compilation of source code. We present a distributed compiler application designed to work with C programming language and use gcc at its back-end. By making use of the unused processing power, a machine running the application can send the code to be compiled across the network to a computer which has the applications daemon and a compatible compiler.

I. INTRODUCTION

Distributed Computing is an efficient method to obtain the desired level of performance by using a cluster of several low-end computers in comparison with a single high-level computer. Such a system is also more cost-efficient and reliable since there is no single point of failure. Such systems are used to solve problems that are computationally intensive by dividing the problem into many tasks, each of which is solved by one or more computers.

The distributed compiler application is designed to work with the C programming language and use gcc at its back-end. A machine running the application can send the code to be compiled across the network to a computer which has the application's daemon and a compatible compiler.

II. BACKGROUND

A. TCP Client Server Communication

Client-server model relationship is one in which one program (i.e. client) requests service or resource from another program (i.e. server). The client establishes a connection with the server over a wide-area network or local-area network. The client then has a request to be fulfilled by the server after which the connection is terminated.

Depending on the client-server model being used, TCP or UDP protocol may be used. Since the compiler application involves file sharing between the two programs (and requires data of infinite length being shared), TCP protocol has been used because it is a connection-oriented protocol. This means that before communicating between the pair of sockets, a connection must be established, after which both sockets act as streams - all available data is read immediately in the order in which it has been sent.

B. Socket Programming

In Unix systems, each process has a set of I/O descriptors that can be read from and written to. These descriptors may refer to files, devices or communication channels (sockets).

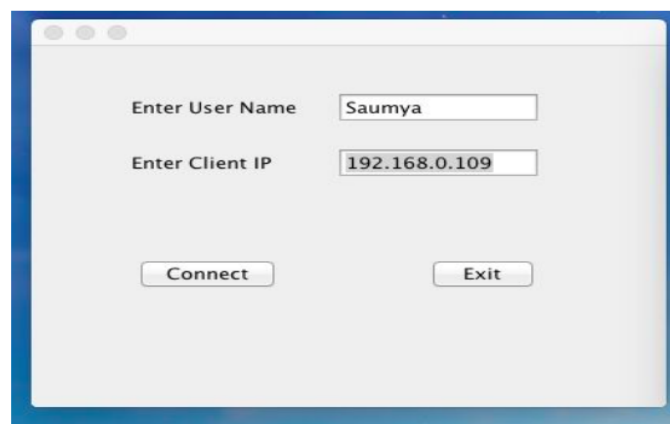
Sockets are an interface for inter-process communication (IPC) similar to that of file I/O. The IPC interface is implemented as a layer over the network TCP and UDP protocols.

The lifetime of a descriptor is made up of three phases: creation (open socket), reading and writing (receive and send to socket), and destruction (close socket). Message destinations are specified as socket addresses; each socket address is a communication identifier that consists of a port number and an Internet address.

IPC is done by exchanging some data through transmitting that data in a message between a socket in one process and another socket in another process. When messages are sent, the messages are queued at the sending socket until the underlying network protocol has transmitted them. When they arrive, the messages are queued at the receiving socket until the receiving process makes the necessary calls to receive them.

III. USING THE APPLICATION

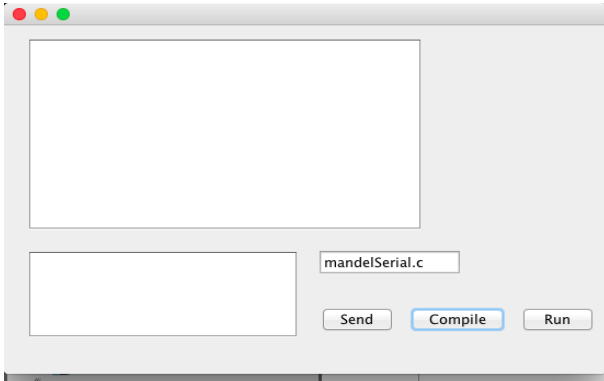
When the client starts the application, a JFrame 'ConnectWindow' appears with fields for entering the IP and user name of the server. After entering the required information, the client connects to the server and another JFrame 'CodeWindowClient' appears for the client and 'CodeWindowServer' appears for the server.



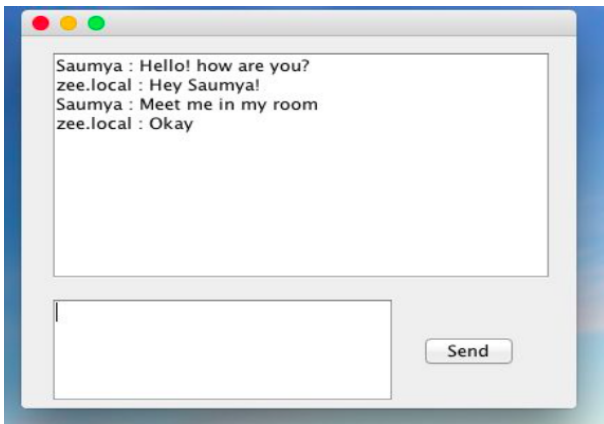
ConnectWindow

'CodeWindowClient' has three text fields, chatpane, messageArea and CFileArea and three buttons 'Send', 'Compile' and 'Run'. 'CodeWindowServer' has

the three textfields same as the client but has one button 'Send'



CodeWindowClient



CodeWindowServer

The client and server can exchange messages by writing the message in the chatPane and pressing the send button. Their messages will appear in the messageArea.

But the main function of the app is to enable the client to compile and run C programs in the server. To do this, the client can write the name of the C file in CFileArea and hit send. His/Her C file will be compiled by the application's daemon running in the server and any errors will be displayed in the messageArea. If there are no errors the client can hit 'Run' button and the server application will run the object code generated and show the output of the program in the messageArea.

A. Communication Between the Sockets

Java is the programming language used for the application. The `java.net.Socket` class represents a socket and `java.net.ServerSocket` class is used by the server side application to listen for clients and establish connection with them.

The following are the steps involved in setting up a TCP connection between the application programs using sockets:

- The server sets up a socket (i.e. `serverSocket`) object and specifies a port number associated with it. This is

the port designated for this communication.

- It then enters the listening mode, invokes the `accept()` method and waits for a client to connect on the specified port.
- While the server is waiting, the client creates a socket object and specifies the IP address (or name) of the server and its port number to connect to.
- A connection can now be established between the client and the server. The client and server sockets can now communicate using sockets.
- The `accept()` method at the server returns a reference to the socket on the server that is connected to the socket of the client.

IV. APPLICATION MODULES

A. Message Exchange Module

This module is meant for setting up the connection and passing messages between the two users using sockets.

- Both the applications first create a socket and then the server waits for the client application program to connect to it.
- For connecting to the other application program, a connection window is used. It is a `JFrame` that lets you enter your user name and IP address you want to connect to.
- Using the entered IP addresses and a default port number for the application, a connection request is sent using the above mentioned parameters.
- Once both applications confirm the connection, a chat window opens up on each end. This is another `JFrame`.
- Each socket involved in the communication process has a reader and writer associated with the input and output stream. Using the reader it can read the messages received and send messages to the other application using the writer. A `BufferedReader` object is associated with the read end of the socket and a `PrintWriter` is associated with the write end of the socket.
- The messages are written down in a text field. Also, the chat log is displayed on a chat pane in the chat window.
- When a message is received from the other application, it is read using the `BufferedReader` and displayed onto the chat pane using an event handler.
- Another event handler is associated with the send button on the screen which is used to send messages written in the text field using the `PrintWriter` object.

B. File Transfer Module

- The main function of the application is to compile and run C programs on the server and send the results back to the remote computer. Thus, the remote host sends a C file to the server and the server receives it.

- When the client presses the 'compile' button in the JFrame, the application sends a message '&COMPILE&', to the server. The server then constructs an object of the class 'FileReceive', and calls its receive() function.
- The receive() function then creates another passive socket using another port. This socket is used to receive the C file.
- While the client also constructs an object of class 'FileReceive', and calls its send() function. The send() function in turn constructs an active socket using the server IP address and port number.
- Once a different connection is established between the client and server, the send() function in client creates a FileInputStream reader, reads the C file and stores the contents of the C file in an array. It then writes the array to the outputStream of the socket. After the file is sent, server closes the file transfer socket.
- The receive() function creates a new file using the FileOutputStream, and reads the array sent by client using the socket and writes the contents of the array to the new file. After the file is received, client closes the file transfer socket.

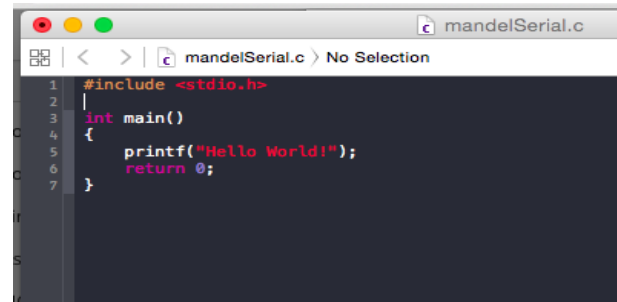
C. Back-end Compiler Module

- Once the server application receives the C file, it needs to compile it and run it. For this it needs to invoke the gcc compiler. To implement this we have used ProcessBuilder class in java. ProcessBuilder class is used to create operating system processes. Each ProcessBuilder instance manages a collection of process attributes. The start() method creates a new Process instance with those attributes. The start() method can be invoked repeatedly from the same instance to create new sub-processes with identical or related attributes.
- After closing the file transfer socket, the receive() function in the server creates an object of CompileCProg class and calls its compile() function. The compile() function creates a new process using the ProcessBuilder class. Each process represents an operating system process. So, here the created process will invoke gcc with the new created C file as its arguments. Any errors can be read using the InputStream of the process and will be sent to the client. After compiling an error free code, an object file a.out will be created.
- Now, if the client presses the '&Run&' button in the JFrame, a message 'RUN' will be sent to the server. When server receives the '&RUN&' message, calls the run() method of the class RunCFile. The method creates a process './a.out' using ProcessBuilder class. The output of the process, which is the output of the C File is then read and sent to the client.

V. RESULTS

We tested our Application with the client as a computer having windows 8.1 OS and server as a computer having

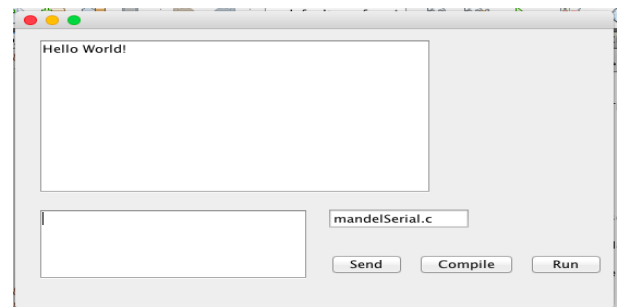
MAC OS. The client did not have any C compiler installed. The user at the client creates a C file and the application sends the file to the server. The application's daemon is running in the server and it compiles the received C program. When the user hits the 'Run' button, the server sends the output to the client.



```
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

Sample C program



Sample output in chat window

VI. FURTHER SCOPE

- The system can be improved in precompilation step in two ways. The originating machine can invoke a preprocessor to handle header files, preprocessing directives and source files to send the preprocessed source to the other machines on the network via TCP either encrypted or using SSH. The application daemon has to run on each of the participating machines. The remote machines can compile the source files without any local dependencies such as libraries, header files or macro definitions to object files and send to the originator for further compilation. The other modification to reduce compilation time is by caching the output from the same input source files.
- The application developed thus far is limited to 2 users. Another improvement can be to make the application scalable for multiple users. This can help to make better use of the system resources of the server.
- Also, if multiple users are available on the network, the client can send the code to be compiled and run to the server which then performs the precompilation step. Now, based on the system resources that are available on the network, it can send the pre-compiled output for further processing by dividing the load equally among these shared system resources. It can then collect the output from each of the systems and send the final output file to the client. This can improve the performance of the system drastically.

VII. CONCLUSION

The system always generates the same results as a local build, is simple and easy to use and usually much faster than a local compile. It does not require all machines to share a file system, have synchronized clocks, or to have the same libraries or header files installed. They can even have different processors or operating systems, if cross-compilers are installed.

ACKNOWLEDGMENT

The authors would like to gratefully acknowledge Professor Laxminarayana Pillutla for his guidance and support throughout the course of the project.

REFERENCES

- [1] Larry L. Peterson and Bruce S.Davie, *Computer Networks a systems approach*, 5th ed. Morgan Kaufman.
- [2] Elliotte Rusty Harold, *Java Network Programming*, 3rd ed. O'Reilly Media, Inc.