EE122: Communications Systems and Networks
Spring 2004 Programming Project
Due: April 30th 2004
Version 1.0 (Aprill 7th)

Logistics:
This project can be done either individually or in groups of 2. For the project description included in this handout, the programming language of choice is C. There will be a separate project handout, but along the same lines, with harder requirements for Java.

Objective:
The objective is to create a very simple chat program that supports single file transfers.

General Outline:
For a chat session exactly one client and one server must participate. There should be only 1 program that can be started in either mode. The server will listen on a user specified port for the client to connect. When a client connects, the server has the option of accepting or declining the chat session depending on the user response. During a chat session, text messages are exchanged between the 2 parties. Additionally, either of the users can initiate a file transfer to the remote host. Depending on user input on the remote host, the file transfer may be declined or accepted. If it is accepted, the file will be transferred in the background (chatting will still take place in the foreground) over UDP. At any given time, only 1 file transfer may be in progress.

Specifics:

Usage:
There will be exactly 1 program, named chitchat, that can work in either client or server mode. The command line usage will be:
chitchat [<port> | <host port>]

If no arguments are specified, a usage message should be displayed.
If 1 argument is specified, the program should start up in server mode.
If 2 arguments are specified, the program should start in client mode.

Server:
The server should use the specified argument as a port, and listen on it for incoming client connections. A single server should support at most 1 client. When a client connects, the user should be prompted whether to accept the chat session. Depending on user response, either an ACCEPT or a DECLINE response will be sent to the client. If a DECLINE is sent, the server should continue listening for another client. If an ACCEPT is sent the server should close the listening socket and begin the chat session.

Client:
The client should use the specified arguments as the hostname and port to connect to. The hostname may be either an IP address or a domain name. If the host is unreachable, the

client should display an appropriate message and exit. If the host is reachable, the client should wait for the ACCEPT/DECLINE response. If it receives a DECLINE response, it should inform the user and exit. If it receives an ACCEPT response, it should being the chat session.

Chat Session:
Once a chat session has started, both client and server are equivalent, and will henceforth be referred to as hosts. For simplicity, assume that all chat input will be in lowercase. Chat packets should be no larger than 100 bytes in size each. However, the user can enter lines longer than 100 bytes, so these should be split over multiple packets if need be.

For the chat session, there are 2 input streams – the keyboard input from the user, and the socket input from the remote host. The keyboard input from the user will take precedence. The keyboard input should be polled first for any text. If there is none waiting, then the receive buffer for the socket should be polled next and any pending messages will be processed. The users can also enter 2 commands which should be handled differently from normal text – "/quit" and "/send <filepath>".

Quitting:
If the user enters "/quit" on the terminal, a QUIT message should be sent to the remote host and the program should immediately exit thereafter (remember to close all sockets and free up any allocated memory). When the remote host receives the QUIT message, it should exit immediately (after the proper close and free operations).

File transfer:
The user can initiate a file transfer by typing the "/send <filepath>" command. Only 1 transfer can be in progress at any given time, so if the program is already sending or receiving a file, it should inform the user and ignore the command. If a file transfer is not in progress, the program should verify that the file exists, and lookup the file size. If the file does not exist, an error message should be displayed to the user. If it does, the program should generate and transmit a SEND message that includes filename (not entire path – if any directory or path information is specified, it should be cut) and file size.

The remote host upon receiving the SEND request should prompt the user whether to accept or not, and return a RECEIVE or NOTRECEIVE message accordingly.

If the host receives a NOTRECEIVE response, it should inform the user and ignore the send command. If it receives a RECEIVE response, it should begin the file transfer over UDP. The UDP packet sizes should be no bigger than 1024 bytes.

The file transfer should go on in the background, whenever the program gets a chance i.e. every time it polls the keyboard and socket stream, it should send a UDP packet. On the receiving end, the program should alternate between reading keyboard input, reading socket input and reading file transfer packets.

The file should be transferred without errors, so some kind of retransmission scheme will be needed. Also, once a file transfer is completed, the user should be informed and the program should again allow the user to enter /send commands.

Keep Alive:
If there is no response (no incoming TCP or UDP packets) from the remote host for 30 secs or more, the program should assume that the remote host crashed. It should inform the user and exit. Thus, every 10 seconds the program should automatically send a KEEPALIVE packet, which the remote host should ignore (except for purposes of tracking that the connection is alive).

NOTE: The program should work on both SPARC and x86 architectures (Big-Endian vs Little-Endian).

Design Report:
A design report (2-4 pages) should also be submitted that includes a description of your application protocol for TCP and file transfer protocol for UDP. List the merits and demerits of your design choices, and highlight any special features that you included. Also, include a short summary of your program flow (this could be a flow chart or state diagram or simple textual description). Specifically describe the main program loop that handles the multiplexing of user keyboard input, socket messages, and UDP file transfer messages.

Submission:
To submit your assignment, first create a readme with compilation instructions. Take the source files and this readme, and copy them into a directory with your SID as its name (for partners use either of the SIDs). Then archive the directory using the tar command and email the archive to ee122@cory.eecs.berkeley.edu. Turn in the design report in the HW box by 5 pm the Monday following the assignment due date.

Grading:
The grading will be done face-to-face with the readers. Timesheets will be posted for people to sign up. The grading will be broken down on a per-checkpoint basis. The major checkpoints are:

1. Getting the chat working
2. Being able to transmit the SEND and QUIT messages and getting appropriate responses
3. Being able to transfer the file:
    a. Using TCP – 33% credit
    b. Using UDP without any retransmission scheme - 50%
    c. Using UDP with retransmission scheme – 100%
4. Satisfying all the miscellaneous specifications e.g. keep-alive, packet size constraints etc.

Extra Credit:
If you can get working implementations of both C and Java programs that can communicate with each other, you can get extra credit. However, this will be awarded iff you complete the first 3 project specifications.

Lastly – be sure to check the newsgroup and webpage for announcements regarding changes to the specifications!!!