– DESCRIZIONE DEL PROGETTO

Il progetto implementa una stanza di chat, secondo il modello applicativo CLIENT-SERVER.
Sono previsti due diversi applicativi, quello lato client, e quello lato server.

CLIENT

Lato client, una volta lanciato l'applicativo, viene presentato un menu' di scelta che permette di scegliere di:

1. Impostare i parametri per la connessione al server secondo il **protocollo TCP/IP**
    - **Impostazione porta di uscita, ip col quale si e' visibili dal server, porta del server e ip del server**

2. Scegliere il **nickname** col quale connettersi alla chat
    - Gli altri client connessi alla chat, leggono il nostro messaggio preceduto dal nostro nickname.
    - Ogni volta che partecipiamo ad una sessione di chat/ci disconnettiamo, gli altri client leggono una notifica a riguardo, che indica che il nostro nickname si e 'disconnesso
    - Il nickname risulta anche nei file delle cronologie di chat, e nel logfile del server
3. **Connettersi alla chat, scegliere la stanza e iniziare a chattare.**
    - Per uscire, l'utente puo' digitare il comando **quit**: la sessione di chat verra' interrotta, ed i messaggi inviati/ricevuti verranno automaticamente salvati sul file apposito con le relative date di inizio sessione della chat.
    - Le stanze vanno dalla numero 1 alla numero 9
    - Un utente puo' leggere ed inviare messaggi soltanto ad utenti connessi alla sua stessa stanza. Lo stesso vale per i messaggi di servizio del server, che avviseranno un utente della connessione/disconnessione alla chat solo degli utenti che sono presenti nella stessa stanza.

E' inoltre prevista una funzione di salvataggio automatico dei messaggi inviati e ricevuti, sotto forma di **cronologia** in un file che viene creato nella stessa cartella dell'exe del programma, file che avra' come nome il nick usato per connettersi alla sessione di chat. Se in diverse esecuzioni del programma, viene usato piu' volte uno stesso nick, le diverse cronologie di chat vengono aggiunte allo stesso file.
Ogni nuova sessione e' indicata all'interno del file, con un apposito messaggio comprensivo anche della data dell'inizio della sessione. All'interno del file della cronologia e' possibile leggere chi ha scritto cosa.
I messaggi inviati/ricevuti vengono periodicamente salvati in una stessa esecuzione, e automaticamente salvati all'interruzione della sessione di chat (sia essa interrotta dall'utente, sia essa provocata da un errore)

ERRORI:
Nel caso in cui si verifichino errori, dovuti ad un cattivo utilizzo dell'applicazione (ad esempio inserimento di un ip o di una porta non valida, o di una porta gia' impegnata) viene stampato un messaggio di errore relativo, per aiutare l'utente a capire cosa e' successo, e nel caso sia possibile, non viene interrotto il processo, ma viene ripetuto il passaggio critico.
Altri casi, come ad esempio una system call() interrotta da un SIGNAL, sono gestiti in maniera trasparente per l'utente, direttamente a livello applicativo.

SERVER

Lato server, una volta lanciato l'applicativo, viene chiesto su quale porta mettersi in ascolto.
Il server imposta automaticamente, come IP sul quale ricevere, tutti quelli delle interfacce dell'host (INADDR_ANY)
Il server inoltre e' concorrente, ovvero attraverso un multiplexing dell'I/O (attraverso una select()), permette di stare in ascolto per le richieste di connessione di nuovi client, e contemporaneamente gestire tutti i client connessi.

4. Il server implementa il comportamento di piu' stanze di chat, in un primo momento accettando le richieste di connessione dei client che vogliono chattare. Quando un client si connette/disconnette, viene stampato sulla console dell'applicativo della chat un messaggio che avvisa che un nuovo utente e' entrato/uscito.
    - Il server tiene traccia di tutti i movimenti (ingressi/uscite) dei vari client, scrivendo su un file all'interno della cartella dell'exe, il nick, il numero di porta e l'ip dei client che si connettono (ed il solo nick di quelli che si disconnettono) e in aggiunta data e ora di sistema
    - Sulla console dell'applicazione lato server, viene di volta in volta stampato l'elenco dei socket attivi relativamente alla chat (quindi l'unico listening socket sul quale il server riceve le richieste di connessione, e tutti i connection socket dei relativi client connessi) dei rispettivi nickname e delle rispettive stanze

    Se NON sono presenti client connessi, la console riporta un messaggio che indica che la stanza e' vuota e sta aspettando che qualcuno si connetta.
    Successivamente, ogni volta che un client dice qualcosa, il server gira quel messaggio a tutti gli altri client connessi che risultano avere lo stesso identificativo di stanza.

NICKNAME:

OGNI MESSAGGIO INVIATO/RICEVUTO DA UN CLIENT, PREVEDE CHE I PRIMI 14 CHAR SIANO RISERVATI AL NICKNAME. QUINDI QUANDO UN CLIENT RICEVE UN MESSAGGIO (CHE E' STATO INIZIALMENTE INVIATO DA UN CLIENT, E GIRATO POI DAL SERVER) LEGGENDO IL CAMPO DEL NICK, PUO' STAMPARE CORRETTAMENTE CHI E' IL MITTENTE DI QUEL MESSAGGIO.

NUMERO STANZA:

IL NUMERO DELLA STANZA DI UN CLIENT (MEMORIZZATO NEL CORRISPONDENTE CHAR) OCCUPA IL 15CHAR DI OGNI MESSAGGIO RICEVUTO INVIATO.
L'IMPORTANZA DI QUESTO IDENTIFICATIVO, RIGUARDA IL MOMENTO DELL'INVIO, IN QUANTO DICE AL SERVER A QUALI CLIENT (QUINDI A QUALE STANZA) E' DESTINATO QUEL MESSAGGIO.
(NON E' ALLO STESSO MODO IMPORTANTE IN RICEZIONE, IN QUANTO IL CLIENT RICEVE SOLO I MESSAGGI DI UTENTI NELLA SUA STESSA STANZA, QUINDI NON DEVE ESEGUIRE CONTROLLI SUL 15ESIMO CHAR IN RICEZIONE)

TIPI DI MESSAGGI:

QUANDO UN CLIENT RICEVE UN MESSAGGIO, DEVE POTER DISTINGUERE TRA UN MESSAGGIO CHE E' STATO GIRATO DAL SERVER MA ERA INIZIALMENTE STATO INVIATO AL SERVER DA UN ALTRO CLIENT (MESSAGGIO NORMALE DI CHAT) E UN MESSAGGIO CHE E' ORIGINARIO DEL SERVER STESSO, E SERVE AD ESEMPIO PER AVVISARE IL CLIENT CHE UN UTENTE SI E' CONNESSO/DISCONNESSO.
AFFINCHE' QUESTI 2 CASI, POSSANO ESSERE GESTITI UTILIZZANDO LA STESSA PORTA, LO STESSO ALGORITMO DI INVIO/RICEZIONE E LO STESSO PROTOCOLLO, HO PREVISTO QUINDI 2 DIVERSI UTILIZZI DEL CAMPO RISERVATO AL NICK, IN MODO TALE CHE IL CLIENT RICEVENTE POSSA DISTINGUERE TRA I 2 CASI SOPRA ELENCATI:

5. IL CLIENT RICEVE UN MESSAGGIO COL CAMPO RISERVATO AL NICK (PRIMI 14 CHAR) RIEMPITO DA 14 *. IN QUESTO MODO CAPISCE CHE QUELLO E' UN MESSAGGIO ORIGINATO DAL SERVER, E QUINDI SI LIMITA A STAMPARE SOLTANTO IL MESSAGGIO PRESENTE NEI SUCCESSIVI CHAR (DAL 15ESIMO IN POI)
6. IL CLIENT RICEVE UN MESSAGGIO CON CHAR PRESENTI NEL CAMPO RISERVATO AL NICK. IN QUESTO MODO SA CHE QUELLO E' UN MESSAGGIO GIRATO DAL SERVER, MA INVIATO ORIGINARIAMENTE DA UN CLIENT, QUINDI STAMPA IL MESSAGGIO PRESENTE DAL 16ESIMO CHAR IN POI, MA FACENDOLO PRECEDERE DAL NICK DEL MITTENTE, SECONDO LA FORMA:
    - %NICKMITTENTE "SCRIVE" %MESSAGGIO.

IN QUESTO MODO CON UN SEMPLICE CONTROLLO SU 14 CHAR DEL MESSAGGIO RICEVUTO, IL CLIENT PUO' STAMPARE A VIDEO COMUNICAZIONI DI SERVIZIO E/O MESSAGGI SCRITTI DA ALTRI UTENTI.
AD ESEMPIO QUANDO UN CLIENT IN STANZA 2 LASCIA LA CHAT, IL SERVER PRENDE IL NICK DI QUEL CLIENT, E LO INSERISCE IN UNA STRINGA DI SERVIZIO ("NICKNAME LASCIA LA CHAT) CHE FA PRECEDERE DA **********, E INVIA QUESTO MESSAGGIO A TUTTI I CLIENT DI STANZA 2 TRANNE QUELLO CHE APPUNTO STA USCENDO.
OGNI CLIENT, RICEVE IL MESSAGGIO "**********2NICKNAME LASCIA LA CHAT", QUINDI SI LIMITA A STAMPARE "NICKNAME LASCIA LA CHAT"
SE INVECE UN CLIENT DI NOME PINCO DICE: "CIAO", TALE CLIENT INVIERA' AL SERVER IL MESSAGGIO "PINCO 2CIAO". IL SERVER GIRERA' IL MESSAGGIO A TUTTI GLI ALTRI CLIENT DI QUELLA STANZA (TRANNE A QUELLO CON NICK PINCO) E OGNI CLIENT ANDANDO A GUARDARE I PRIMI 14 CHAR, SA CHE DOVRA' STAMPARE SECONDO LA FORMA:
"PINCO SCRIVE: CIAO"

NB: ANCHE SE UN CLIENT RISULTA CONNESSO (DAL PUNTO DI VISTA DEL SERVER) NEL MOMENTO IN CUI RITORNA L'ACCEPT() LATO SERVER, DAL PUNTO DI VISTA DEI CLIENT, SI E' VERAMENTE CONNESSI NEL MOMENTO IN CUI SI INVIA IL PRIMO MESSAGGIO VERO E PROPRIO IN CHAT. QUESTO POICHE' NICKNAME DEL CLIENT E NUMERO DI STANZA, SONO PRESENTI ALL'INTERNO DI OGNI MESSAGGIO INVIATO DA UN CLIENT, QUINDI E' IN QUESTO MOMENTO CHE IL SERVER PUO' SALVARE IL NICK E LA STANZA ASSOCIATI AL FD DEL SOCKET DI QUEL DATO CLIENT.
ECCO PERCHE' SOLO QUANDO UN CLIENT, APPENA CONNESSO, INVIA IL PRIMO MESSAGGIO, VENGONO INVIATI A TUTTI GLI ALTRI CLIENT IN QUELLA STANZA GLI AVVISI CHE IL CLIENT SI E' CONNESSO ALLA STANZA (ANCHE SE APPUNTO IL CLIENT PER IL SERVER ERA GIA' CONNESSO) E VIENE SALVATO L'EVENTO DI CONNESSIONE NEL LOGFILE (PERCHE' E' IN QUEL MOMENTO CHE ABBIAMO OLTRE ALL'IP E ALLA PORTA DEL CLIENT, IL SUO NICK E LA STANZA DA LUI SCELTA)
A QUESTO PUNTO SORGE UN PROBLEMA:
COSA SUCCEDE SE UN CLIENT SI CONNETTE, MA PRIMA DI INVIARE IL PRIMO MESSAGGIO NELLA STANZA DI CHAT, DECIDE DI DISCONNETTERSI ?
QUESTO CASO INFLUENZA 2 COMPORTAMENTI DEL SERVER:

1)QUELLO DELL'AVVISO A TUTTI GLI ALTRI CLIENT DELLA DISCONNESSIONE DI UN UTENTE PER IL QUALE NON ERA ANCORA APPARSO IL MESSAGGIO DI CONNESSIONE

2)QUELLO DEL SALVATAGGIO SUL LOGFILE DEL SERVER DELLA DISCONNESSIONE DI UN UTENTE PER IL QUALE NON AVEVAMO ANCORA REGISTRATO UN EVENTO DI CONNESSIONE.

LA SITUAZIONE E' GESTITA IN QUESTO MODO:

1)IL CLIENT SI CONNETTE (CONNESSO DAL PUNTO DI VISTA DEL SERVER, ABBIAMO IL FD DEL SUO SOCKET, IL SUO IP E LA SUA PORTA, MA NON SAPPIAMO IL SUO NICK E LA STANZA DA LUI SCELTA) SE ESCE PRIMA DI INVIARE IL PRIMO MESSAGGIO (E QUINDI DI FARCI CONOSCERE NICK E STANZA) VISTO CHE NON AVEVAMO ANCORA AVVISATO NESSUN CLIENT DELLA SUA CONNESSIONE, EVITIAMO DI AVVISARE I CLIENT DELLA SUA DISCONNESSIONE (QUINDI IL CLIENT, DAL PUNTO DI VISTA DEGLI ALTRI UTENTI CONNESSI IN CHAT, NON E' MAI ENTRATO IN REALTA')

2)PER QUANTO RIGUARDA IL LOG FILE, NON AVREBBE SENSO CREARE UN SERVER CHE NON TIENE TRACCIA DEGLI UTENTI CHE SI CONNETTONO, MA ESCONO PRIMA DI SCRIVERE QUALCOSA. QUINDI IN QUESTO CASO, SE L'UTENTE ESCE PRIMA DI AVER INVIATO IL PRIMO MESSAGGIO (QUINDI NON C'E' ANCORA UN EVENTO SUL LOGFILE CHE DICE CHE QUEL CLIENT E' ENTRATO SUL SERVER) INSERIAMO DIRETTAMENTE UN EVENTO SUL LOG CHE SPIEGA CHE UN UTENTE E' ENTRATO (IN UN MOMENTO PRECEDENTE) MA E' USCITO SUBITO PRIMA DI SCRIVERE QUALCOSA (E LASCIAMO SU UNKNOWN IL NICK E LA STANZA SCELTI DA QUEL CLIENT, PERCHE' NON LI ABBIAMO RICEVUTI) IN QUESTO MODO, GLI AMMINISTRATORI CHE LEGGONO IL LOGFILE DEL SERVER, CAPISCONO COSA E' AVVENUTO, E NON SI TROVANO NELLA SITUAZIONE DI DOVER LEGGERE EVENTI RIGUARDANTI UTENTI DISCONNESSI PRIMA CHE RISULTASSERO CONNESSI NEI LOG.

- ARCHITETTURA DELL'APPLICAZIONE

SERVER:

Il server e' concorrente, quindi riesce a gestire "contemporaneamente" tutti i client connessi.
Questa concorrenza e' ottenuta tramite un multiplex dell'I/O, attraverso una funzione select() ,che si occupa di controllare lo stato di tutti i socket attivi relativamente al processo di chat.

Il processo mantiene un socket di ascolto sempre attivo, che si occupa di rimanere in attesa delle richieste di connessione dei client. Quando un client e' pronto a connettersi, il listening socket risulta ready in lettura e quindi lanciamo l'accept() che aprira' la connessione su un'altra socket dedicato al client, mentre il listening socket continua a restare in attesa di richieste di connessione.

E' tenuta traccia di tutti i socket attivi attraverso un'array di struct, che si occupa di registrare oltre a i nickname e alle stanze corrispondenti, i file descriptor di tutti i socket della chat().
Il listening socket non viene mai chiuso, mentre i connection socket vengono chiusi al termine della connessione del client.

Allo start del server, l'array di struct (multiplexedfds) contiene solo il fd del listening socket (in quanto non ci sono client connessi) ed il relativo set di controllo (da passare alla select) conterra' solo il fd del listening socket. Man mano che i client si connettono, vengono aggiunti nuovi fd sia all'array di struct sia sul set da passare alla select , e questi nuovi fd aggiunti, saranno ricontrollati assieme a tutti gli altri al giro di select() successivo.
Nello specifico:

- Si prende una copia del set di controllo (in lettura) di fd e la si passa alla select()
- Il programma si blocca, finche' uno dei fd del set di controllo non diventa ready in lettura
- Appena uno o piu' fd diventano ready, la select azzera tutti gli altri fd del set passato
    - che non sono ready, e lascia a 1 quelli che lo sono
- Controlliamo uno ad uno tutti i fd attivi (di cui teniamo traccia nell'array di struct) per sapere quali hanno fatto scattare la select():
    - Se il listening socket e' tra questi, allora un nuovo client e' pronto a connettersi. Quindi stabiliamo la connessione (lanciando accept() per il client che sta sulla coda del listening socket dei 3wh completati) su un nuovo socket apposito e salviamo il fd di questo nuovo socket nella prima posizione valida dell'array di struct.
      Da adesso in poi, per "dialogare" col client connesso, lavoreremo su quest'ultimo socket (il listening socket torna ad aspettare richieste di connessione di altri client.
        - Aggiungiamo il fd del nuovo socket di connessione col client, all'originale del set di fd
        - NB: che anche se adesso il nuovo client e' dal punto di vista del server connesso a tutti gli effetti (in quanto si e' concluso il 3wh) non abbiamo ancora ricevuto un vero e proprio messaggio testuale dal client, quindi non sappiamo il suo nick e la sua stanza. Quindi non possiamo ne' completare la corrispondenza sull'array di struct, ne' inviare a tutti gli altri client connessi il messaggio di servizio che li avvisa dell'accesso, ne' inserire l'evento sul logfile

    - Se un socket di connessione risulta ready in lettura, vuol dire che il client ci sta dicendo qualcosa:
        - Se leggiamo un numero di char >=0 e questo e' il primo messaggio che il client ci invia (sull'array di struct la corrispondenza del nick non c'e') allora tiriamo fuori il nick e la stanza del client dal messaggio ricevuto, e li salviamo sull'array di struct. Giriamo poi a tutti gli altri client connessi (e che si trovano nella stessa stanza) un messaggio di servizi o che li avvisa dell'accesso di quell'utente, e salviamo l'evento sul file di log che si trova sul disco.
          Se invece non era il primo messaggio, allora ci limitiamo a girarlo a tutti gli altri client di quella stanza.
        - Se leggiamo zero char, vuol dire che quel client ci ha inviato un FIN, e quindi si sta disconnettendo.
          Se aveva gia' inviato il suo primo messaggio dopo essersi connesso, allora sappiamo il suo nick e la sua stanza, quindi possiamo inviare un messaggio di servizio agli utenti della sua stessa stanza che li avvisa che quel nickname sta uscendo. Se invece non aveva ancora inviato il suo primo messaggio, per gli

altri utenti e' come se non si fosse mai connesso, quindi non gli inviamo neanche il messaggio che li avvisa della disconnessione di tale utente, e sul log file salviamo un evento apposito (*si legga sopra*). In ogni caso proveddiamo a cancellare la sua corrispondenza sull'array di struct, a chiudere quel socket e ad eliminare il relativo file descriptor dall'originale del set di controllo che si passa alla select al giro successivo.

- Si ricomincia tutto da capo

CLIENT:

Lato client, il processo inizia col menu' (main()) nel quale e' possibile scegliere l'operazione da eseguire. Si possono settare i vari parametri per la connessione TCP/IP (in tal caso viene lanciata una funzione per il settaggio). Se si prova a connettersi al server, senza aver precedentemente lanciato la funzione di settaggio dei valori, quest'ultima viene lanciata automaticamente. Una volta settati tutti i parametri TCP/IP (e nickname) la funzione puo' ritornare correttamente, altrimenti se uno o piu' parametri non sono corretti, viene stampato un messaggio adeguato, e la funzione ritorna senza che siano state salvate le modifiche (e quindi in questo caso non e' ancora possibile connettersi alla chat).
Nel caso in cui il settaggio vada a buon fine, si ritorna al menu', ed e' possibile lanciare la sessione di chat (che parte attraverso la funzione chat()). Quando la sessione di chat e' terminata, la funzione chat() ritorna un valore in base al quale a livello del main e' possibile stampare un messaggio che indichi il motivo per il quale la sessione e' stata interrotta. (se ritorna 1 siamo stati noi client ad interromperla, digitando quit, se ritorna 0 allora e' stato il server a chiudere la sessione, altrimenti se ritorna -1 si e' verificato un errore durante la sessione di chat).

Appena parte la funzione di chat viene chiesto di inserire un numero che indichi la stanza alla quale si vuole accedere. Viene creato il socket per la connessione al server, viene bindato (*anche se il bind() lato client non e' strettamente necessario*) e viene lanciata la system call connect() che invia il SYN al server, e ritorna nel momento in cui il client riceve sul socket il SYN-ACK dall'altro end point.Se le system call danno errore, la funzione di chat ritorna con valore -1 (il caso di una system call interrotta da un signal e' gestito direttamente ripetendo la chiamata alla funzione).

Inizia quindi il ciclo del client, che verra' interrotto nel momento in cui il client/il server decide di chiudere la sessione di chat, o nel caso in cui avvenga un errore. Per gestire il ciclo, si utilizzano 2 variabili logiche, keep4us e keep4srv, che indicano (quando sono vere) che la sessione e' ancora attiva per volere sia del client che del server. Il ciclo continua finche' sono entrambe vere. Se il client decide di chiudere la sessione di chat, viene posta a zero keep4us (e quindi si esce dal ciclo, si chiude il socket, e viene ritornato 1). Se e' il

server a chiudere invece, keep4srv viene posta a zero (si chiude il socket, si esce dal ciclo e viene ritornato 0).
Se si verifica, durante la sessione, un qualunque tipo di errore, il ciclo si interrompe ugualmente, vengono effettuate le azioni opportune (stampa di messaggio di errore, chiusura di buffer aperti e/o socket) e la funzione chat() ritorna -1.

All'interno del ciclo esterno, viene operato un multiplex dell'I/O, affinche' il processo possa "contemporaneamente" prendere i caratteri da tastiera (che rappresentano il messaggio che si sta scrivendo in chat) e inviarli (stdin ready in lettura), e accorgersi che il server ci stia inviando dati (in questo caso il socket diventa ready in lettura).
Viene quindi utilizzata una select(), alla quale viene ad ogni ciclo passato un set contenente il fd del socket e quello dello stdin (==fileno(stdin)).

- Il processo si blocca sulla select, in attesa che vengano digitati caratteri (e venga premuto invio=>questo manda i char dal buffer della tastiera al buffer stdin, che risulta quindi ready in lettura) e/o che compaiano dati da leggere sul socket della connessione (al di sopra della soglia minima).
    - Se passano 150os senza che nessuno dei 2 I/O risulti ready, la select () ritorna ugualmente, ma siccome siamo stati troppo tempo senza far nulla, viene stampato un messaggio di avviso, e sostanzialmente questo equivale all'aver digitato quit (keep4us diventa o e la funzione di chat ritorna 1)
    - Se invece diventa pronto in lettura il socket di connessione allora proviamo a leggere da quest'ultimo:
        - Se la full_read() non va a buon fine, nel caso in cui siano presenti messaggi nel buffer della cronologia temporanea, li scarichiamo sul file della cronologia, e poi ritorniamo -1.
        - Se la full_read() ritorna zero, allora il server ci ha mandato un FIN e sta chiudendo la connessione: impostiamo su zero keep4srv.
        - Altrimenti abbiamo letto un messaggio valido:
            - Se i primi 14 char del messaggio sono la stringa riservata **********, allora stampiamo direttamente il messaggio di servizi o che parte dal 15esimo char in poi.
            - Altrimenti il messaggio ricevuto e' un messaggio girato dal server ma che era stato inviato in primo luogo da un client, quindi tiriamo fuori il nick del mittente dai primi 14 char, e stampiamo secondo la forma Nickmittente "scrive" restodelmessaggio
            - In entrambi i casi, che sia una comunicazione di servizio, o il messaggio di un altro client, salviamo questo messaggio nel buffer della cronologia temporanea (se il buffer aveva raggiunto gia' un certo limite, allora lo scarichiamo prima sul file della cronologia sul disco e poi spostiamo l'ultimo messaggio nel buffer)

    - Se nel frattempo, sono stati digitati char su tastiera, ed e' stato premuto invio, il buffer stdin risulta pronto. In tal caso costruiamo il messaggio da mandare mettendo il nostro nick nei primi 14 char, la stanza (convertita in char) nel 15esimo, ed il vero contenuto del messaggio dal 16esimo in poi. Prima di mandare, ci assicuriamo che non sia stato digitato quit da tastiera (in tal caso keep4us diventa zero). Altrimenti inviamo il messaggio cosi'

costruito, con una write() sul socket. Se la write() non va a buon termine, scarichiamo il contenuto del buffer della cronologia temporanea sul file del disco, e la funzione chat() ritorna -1. Altrimenti se va a buon fine, salviamo il messaggio inviato al server nel buffer della cronologia temporanea (controllando sempre che se questo e' gia' pieno oltre un certo livello, va prima scaricato sul file del disco).

- Prima di controllare le variabili del ciclo, scarica sul file del disco il contenuto del buffer della cronologia . In generale(con questo, e con gli accorgimenti di sopra) quindi la cronologia sara' salvata:
  - Sia se digitiamo quit sia se il server chiude la sessione
  - Sia se si presentano errori
  - E ogni volta che il buffer temporaneo raggiunge un certo limite massimo
- Il ciclo ricomincia (se le variabili keep4us && keep4srv sono vere, altrimenti si esce)

- PROTOCOLLI DI RETE

L'applicazione, da entrambi i lati, utilizza un protocollo TCP a livello transport, e IP4 a livello network. (TCP/IP).
Quindi mentre a livello network il servizio e' datagram (il protocollo IP e' sempre datagram) a livello transport il servizio e' con connessione e con riscontro.
A lato client, viene creato un socket (che viene bindato, anche se non strettamente necessario, su una combinazione di IP e PORTA di uscita che rappresentera' l'end point client) che e' utilizzato per lanciare una richiesta di connessione al server (il cui indirizzo IP e porta di destinazione, sono passati attraverso un'opportuna struct sockaddr_in, alla funzione connect()) Quando il client lancia la system call connect() sta performando una connessione attiva.
Dall'altra parte il server, avra' aperto un socket (che binda su INADDR_ANY e su una data porta e trasforma in un socket di tipo listen, che e' una apertura passiva) che mantiene una doppia coda (vedi avanti).
Nel momento in cui il client lancia la connect(), a livello transport parte il segmento SYN diretto al server. Quando il server lo riceve, mette quel client nella coda che delle due rappresenta i client che sono al primo stadio del 3wh, e risponde sempre a livello transport con un segmento SYN-ACK (ack relativo al SYN ricevuto prima). Il client adesso (per il quale il flusso dell'applicazione si era fermato, essendo la connect() una funzione bloccante) riceve quindi il SYN-ACK, e:
-a livello transport: risponde con un ACK al SYN originato dal server (inizio terzo step del 3 way handshake)
-a livello applicativo: ritorna la connect(), quindi il flusso del client riprende a 2/3 del 3 way handshake.
Il client, nel momento in cui ritorna la connect() pensa di essere connesso, anche se per il server, la fase di inizializzazione non e' ancora terminata completamente.

Il server che riceve l'ACK finale inviato dal client, mette quel client nella lista (del listening socket) per i quali e' completato il 3way handshake. Il listening socket quindi risulta ready, e (essendo multiplexato con la select()) lanceremo l'accept() che girera' su un nuovo socket la connessione per quel client (o per il primo di quelli che si trovano ad aver completato il 3 way handshake).
La MAN page avverte che se un client si trova ad inviare dati, nel lasso di tempo che passa tra la ricezione dell'ACK finale da parte del server, e l'uso dell'accept() da parte di quest'ultimo, questi vengono mantenuti nel buffer del socket.
I socket utilizzati sono ovviamente di tipo TCP (AF_INET, SOCK_STREAM) per ip a 32 bit (ip4).

All'interno dell'applicativo, sono previste funzioni di conversione tra l'host order (spesso dotted decimal) in network order.
A livello applicazione, utilizziamo le API Berkley per la network programming, ovvero attraverso i socket, interfacciamo il livello applicativo con il livello transport (porte TCP in questo caso) e quello network (non direttamente pero', in quanto non usiamo raw socket).
Quando uno dei due end point (un end point e' la combinazione di ip e porta) decide di disconnettersi (chi invia per primo la richiesta di disconnessione sta eseguendo una active close) invia un segmento a livello transport di FIN. Il peer dall'altra parte risponde con un ACK (performando una passive close).
Da questo momento in poi, l'end point che ha effettuato l'active close, ricevuto l'ACK chiude la connessione, ma non chiude definitivamente il socket: lo lascia in uno stato di attesa, in modo tale che quando anche l'altro end point mandera' il suo FIN (performando una close) lui potra' riceverlo e rispondere con un ACK (se lo chiudesse, risponderebbe al FIN con un RST).

- SPIEGAZIONE CODICE

• SERVER

✓ MAIN:

Dichiariamo nel main un array di tipo struct fdmultiplexati. Esso conterra' le corrispondenze tra filedescriptor (del socket aperto) stanza in cui si trova il client di quel socket di connessione e nick. Quando un client si collega (cioe' viene completato il 3 way handshake e l'accept() lato server ritorna il connection socket, viene inserito il fd nell'arra)y. Nickname e stanza del client (essendo presenti nei messaggi che invia un client) vengono salvati nel momento in cui quel client connesso (per il quale quindi e' stato precedentemente inserito il fd nell'array) manda il primo messaggio.

```
struct fdmultiplexati {
      char *nickclient;
      char   stanza;
      int    fd;
                  };

int main() {…
…
struct fdmultiplexati multiplexedfd[FD_SETSIZE];
…}
```

Il corpo centrale del codice del server, e' un ciclo for infinito che parte fin da subito con una select(), che blocca il flusso del programma, in attesa che uno o piu' dei socket passatigli in input diventino

pronti in lettura. Alla select e' passata una copia del set di fd da controllare. Ogni modifica al set (in un ciclo) verra' salvata al set originale (che sara' quindi riutilizzato al ciclo successivo). Durante il ciclo in corso invece, il controllo prosegue sulla copia del set passata all'inizio alla select(). Al primo ciclo, quando il server e' vuoto, c'e' un solo socket aperto, quello di listening, che occupa la posizione 0 dell'array (senza avere corrispondenze di stanza e nick) e non viene mai chiuso. Il massimo fd (inizializzato sul listening socket) assume di volta in volta il valore del massimo fd creato, e viene passato di conseguenza (sommandogli uno) alla select.

```
for(;;)
{

   if(iultimofd==0)    printf("\n----CHAT APPENA AVVIATA, IN ATTESA DI CLIENT...\n");
   rset=set1;
   nfdready=select(maxfd+1,&rset,NULL,NULL,NULL);
```

Da qui in poi, se il flusso continua, vuol dire che qualcosa ha fatto scattare la select(). Iniziamo quindi i controlli per vedere quali dei fd in utilizzo, puntano a socket che sono in quel momento risultati pronti alla lettura. Il codice si divide in 2 parti: una che si occupa di eseguire un controllo fisso sul socket di ascolto, in posizione 0, e l'altra che si occupa di scorrere i rimanenti fd sull'array (che saranno tutti fd riguardanti socket di connessione con i client).
Nel caso in cui FD_ISSET(listensfd,&rset) restituisca 1, allora il socket di listening e' risultato ready. Questo vuol dire che c'e' un client dall'altra parte per il quale possiamo compiere l'ultimo passo per il completamento del 3 way handshake, stabilendo una connessione. In tal caso lanciamo un' accept() sul listening socket (riavviandola nel caso in cui la system call() sia interrotta da un signal, e riavviando il ciclo nel caso in cui si verifichi un errore diverso. *(Questa implementazione, potrebbe risultare problematica nel caso in cui il server sia floddato con un numero elevato di richieste di connessione che poi risultano tutte in errori sull'accept(). Ad esempio se ci sono gia' n client connessi, e per qualche motivo tutte le successive accept() non andranno a buon fine, se ogni volta che uno o piu' client inviano un messaggio, e contemporaneamente arriva la richiesta di connessione di un client, che dara' luogo ad un valore errato della accept(), il ciclo for verra' riavviato, e verranno ignorati i socket sui quali sono presenti i messaggi inviati dagli altri client. Per evitare una situazione del genere si potrebbe pensare sostituire la forma usata, con un if che aggiunge il nuovo client se e solo se la accept() ritorna valore >=0, ed in caso contrario, lascia proseguire il flusso verso il resto del codice, in modo tale che vengano controllati gli altri fd anche nel caso in cui la accept() non vada a buon fine. -Per evitare di aumentare la complessita' del codice, comunque, ho lasciato il codice nella forma come sopra-).*
Se la accept() va quindi a buon fine, il nuovo socket di connessione viene salvato in connsfd, e viene lanciato un ciclo for che salva il fd sulla struct (elemento) dell'array multiplexedfds non appena trova una posizione vuota (che per implementazione ha valore -1) ed esce dal for. All'uscita dal for viene effettuato un controllo per essere sicuri che l'uscita non sia avvenuta per completamento di tutti gli step, che in tal caso vuol dire che non c'e' una posizione vuota, quindi l'indice assume valore FD_SETSIZE, ed in tal caso come nel caso di una accept() errata, il cilco viene riavviato.
Altrimenti il flusso prosegue, e il nuovo fd (che punta al socket di connessione per il nuovo client) viene aggiunto al set originale di fd che dovranno essere ricontrollati al ciclo successivo.
Viene valutato il nuovo maxfd, aggiornato il valore del numero di fd totali da controllare, e nel caso in cui decrementando la variabile che conteneva il totale dei fd risultati ready nella select(), si ottenga 0, vuol dire che non ci sono piu' socket da controllare, ed il ciclo si riavvia.

```
   if(FD_ISSET(listensfd,&rset)) {

            do connsfd=accept(listensfd,NULL,NULL);
               while(connsfd<0 && errno==EINTR);
            if(connsfd<0) {perror("\nCONNECT():"); continue;}
            printf("\n****UN NUOVO CLIENT SI E' CONNESSO ALLA CHAT !");
fflush(stdout);

            for(i=0;i<FD_SETSIZE;i++)
                  if(multiplexedfd[i].fd<0) {
                     multiplexedfd[i].fd=connsfd;
                     break;
                  }

            if(i==FD_SETSIZE) {printf("\nTROPPI UTENTI CONNESSI\n"); continue;}

            FD_SET(connsfd,&set1);


            if(connsfd>maxfd) maxfd=connsfd;
            if(i>iultimofd) iultimofd=i;

            stampa_arraygenerale(multiplexedfd,i,iultimofd);

            if(--nfdready<=0) continue;
   }
```

Altrimenti inizia un ciclo for che scorre sull'array di struct, dall'elemento in posizione 1 all'ultimo inserito in uno dei precedenti cicli: se l'elemento in analisi e' <0, allora non e' inizializzato, ed il for va avanti. Altrimenti controlla se e' risultato ready nella select()

```
(i=1;i<=iultimofd;i++)   {
            if( multiplexedfd[i].fd <0)
                        continue;
```

Se e' risultato ready, lancia una full_read() sul socket. Se la full_read() ritorna 0, vuol dire che il socket di connessione con un client, era risultato pronto in lettura, ma non leggiamo dati. Quindi il client ci ha inviato un FIN e ha chiuso la connessione.
In tal caso si presentano 2 scenari:
Il client si sta disconnettendo, ma non aveva ancora inviato il suo primo messaggio vero e proprio. Oppure aveva gia' inviato un messaggio.
Nel primo caso non abbiamo salvato il suo nick e la sua stanza, quindi non avevamo inviato a tutti gli altri client connessi un messaggio di servizio che li avvisava dell'ingresso del client, ne' avevamo salvato l'ingresso dell'utente nel log file. Quindi in tal caso non lanciamo la funzione che avvisa tutti gli altri client della disconnessione di questo utente (toall_mexSRV_uscitautente()) e lanciamo

comunque la funzione che si occupa del logfile (la quale registrera' un evento particolare sul log, gestendo direttamente questo tipo di caso – clientdisconnesso_salvalog()).
Altrimenti, se il client aveva gia' inviato il primo messaggio, avvisiamo tutti gli altri client della sua disconnessione, e lanciamo comunque la funzione di log.
In ogni caso, all'uscita di questo utente, chiudiamo il socket di connessione, reinizializziamo il fd sull'array di struct, ed eliminiamo dall'originale del set il fd relativo.

```
if(FD_ISSET(multiplexedfd[i].fd,&rset)) {
        if( (r=full_read(multiplexedfd[i].fd,toreceive,300)) ==0) {

                if(multiplexedfd[i].nickclient!=NULL)
                        toall_mexSRV_uscitautente(multiplexedfd,i,iultimofd);

                if(clientdisconnesso_salvalog(multiplexedfd,i)<0) {
                        printf("\n***LOG-CORROTTO!\n");
                        fflush(stdout);
                }
                close(multiplexedfd[i].fd);
                FD_CLR(multiplexedfd[i].fd,&set1);
                multiplexedfd[i].fd=-1;

                if(multiplexedfd[i].nickclient!=NULL) {
                        free(multiplexedfd[i].nickclient);
                        multiplexedfd[i].nickclient=NULL;
                }

                stampa_arraygenerale(multiplexedfd,i,iultimofd);

        }
```

Altrimenti, se la full_read() legge almeno un char si entra nel corpo dell'if apposito.
Se questo messaggio inviatoci, e' il primo da parte del client da quando si e' connesso (quindi nick non presente, allora multiplexedfd[i].nickclient==NULL), allora possiamo salvare il suo nick e la stanza da lui scelta (il nick si trova nei primi 14 char del messaggio letto, la stanza e' il quindicesimo). Poi inviamo a tutti gli altri client connessi il messaggio di servizio che li avvisa che quel client si e' connesso (con la funzione toall_mexSRV_nuovoutente()) e salviamo l'evento dell'ingresso nel logfile. E stampiamo l'array risultante sulla console dell'applicazione della chat lato server.
In ogni caso il corpo dell'if prosegue, girando quel messaggio (il server gira il messaggio per come l'ha ricevuto, con tutto il nick e il char della stanza) a tutti gli altri client connessi, con un nuovo for che cicla sugli elementi dell'array di struct, e fermandosi solo su quelli che non sono inizializzati ancora a -1, che hanno lo stesso identificativo di stanza che ha quel client mittende del messaggio, e facendo attenzione a non far corrispondere mittente e destinatario.
Per ognuno di questi client validi (ovvero destinatari) viene scritto il messaggio con una write() sul socket relativo (multiplexedfd[i].fd) facendo attenzione a riavviare la write() nel caso in cui venga interrotta da un signal (e a riavviare il ciclo in caso di errore della write()).
Se invece la full_read() ritorna -1, stampa l'errore relativo con perror() e riavvia il ciclo.
In ogni caso, se decrementando il numero dei fd risultati ready con la select() si ottiene <=0, allora non c'e' bisogno di controllare i successivi socket ready in lettura.

```
else if(r>0){

                        if(multiplexedfd[i].nickclient==NULL) {
                            multiplexedfd[i].nickclient=calloc(15,sizeof(char));
                            strncpy(multiplexedfd[i].nickclient,toreceive,14);
                            multiplexedfd[i].nickclient[14]='\0';
                            multiplexedfd[i].stanza=toreceive[14];

                           toall_mexSRV_nuovoutente(multiplexedfd,i,iultimofd);

                           if(clientconnesso_getinfo_salvalog(multiplexedfd,i)<0) {
                                 printf("\n***LOG CORROTTO!");
                                  fflush(stdout);
                               }


                        stampa_arraygenerale(multiplexedfd,i,iultimofd);

                        }

                        for(j=1;j<=iultimofd;j++)
                             if(multiplexedfd[j].fd!=multiplexedfd[i].fd &&
multiplexedfd[j].fd>=0 && multiplexedfd[j].stanza==multiplexedfd[i].stanza) {

                                          do w=write(multiplexedfd[j].fd,toreceive,300);
                                               while(w<0 && errno==EINTR);

                                          if(w<=0) {perror("\nIMPOSSIBILE GIRARE UN
MESSAGGIO AD UNO DEI CLIENT"); continue;
                                          }
                                }

                }

                else {perror("\nREAD:"); continue;}


                if(--nfdready<=0) break;
          }
    } //chiusura for interno

} //chiusura for esterno

} //chiusura main
```

- ✓ **FUNZIONI ESTERNE AL MAIN:**

La funzione toall_mexSRV_nuovoutente() prende in input l'array di struct multiplexedfd[],l'int i_nuovoclient che e' la posizione sull'array della struct che rappresenta il client in questione, e il valore dell'indice fino al quale ciclare sull'array. Per il dato i-esimo client, invia a tutti gli elementi validi (client connessi alla stessa stanza del nuovo utente) una stringa rappresentante un messaggio di servizio, ovvero interpretata come tale quando ricevuta da ogni client. La stringa in questione presenta 14 * nella posizione riservata al nick, e nella parte riguardante il messaggio, la comunicazione di servizio che deve essere stampata direttamente dal client, e che contiene il nick del client e l'avviso che si e' connesso alla stanza.

```
void toall_mexSRV_nuovoutente(struct fdmultiplexati multiplexedfd[],int i_nuovoclient,int
iultimofd)    {
int j,w;

char tosend[300]="**************";
strcat(tosend," IL CLIENT ");
strncat(tosend,multiplexedfd[i_nuovoclient].nickclient,strlen(multiplexedfd[i_nuovoclient].nic
kclient));
strcat(tosend," SI E' CONNESSO ALLA STANZA DI CHAT");


for(j=1;j<=iultimofd;j++) {

        if(j==i_nuovoclient) continue;

        if(multiplexedfd[j].fd>=0 &&
multiplexedfd[j].stanza==multiplexedfd[i_nuovoclient].stanza){
                do w=write(multiplexedfd[j].fd,tosend,300);
                        while(w<0 && errno==EINTR);

                 if(w<=0) {perror("\nIMPOSSIBILE GIRARE UN MESSAGGIO SRV AD UN
CLIENT");
                                continue;
                 }
        }

}
}
```

La funzione toall_mexSRV_uscitautente() fa la stessa cosa, avvisando pero' i client che quel dato utente sta uscendo.

La funzione clientdisconnesso_salvalog() prende in input l'array di struct e l'indice i del client che si sta sconnettendo. La funzione apre un file nella cartella dell'exe (ottenendo il percorso con getcwd) e salva l'evento nel file di log (aprendolo in append). Fa prima pero' un controllo, in modo da salvare una stringa particolare con un messaggio particolare come evento, se quel client si e' disconnesso senza scrivere almeno il primo messaggio (poiche' in tal caso non si conosce il nick e la

stanza di quel client) oppure un messaggio normale di log con nickname e stanza in caso contrario. In entrambi i casi viene segnata l'ora dell'occorrenza dell'evento. E viene stampato un messaggio di salvataggio del log sulla console della chat nel caso vada tutto a buon fine (altrimenti un messaggio di errore)

```
int clientdisconnesso_salvalog(struct fdmultiplexati multiplexedfd[],int i_uscente) {


FILE *pfile;
char percorso[FILENAME_MAX],nickname[15],stanza_lasciata;
time_t datacodif;
struct tm *dataleggibile;

time(&datacodif);
dataleggibile=localtime(&datacodif);
getcwd(percorso,sizeof(percorso));

strcat(percorso,"/");
strcat(percorso,"logfile");
strcat(percorso,".txt");

if(multiplexedfd[i_uscente].nickclient!=NULL) {
        strncpy(nickname,multiplexedfd[i_uscente].nickclient,15);
        stanza_lasciata=multiplexedfd[i_uscente].stanza;
}

else {
        strcpy(nickname,"-unknown-");
        stanza_lasciata='?';
}


if((pfile=fopen(percorso,"a"))==NULL)
   return -1; //non riesce ad aprire il file
else{
      if(stanza_lasciata=='?') {
            if(fprintf(pfile,"\n\n//%d:%d:%d %d-%d-%d// nick:%s (stanza %c) \n >>>Il
client ha effettuato connessione al server, ma ha scelto di disconnettersi prima di inviare
almeno 1 messaggio\n >>>Nick e stanza scelta dal client non pervenuti!",dataleggibile-
>tm_sec,dataleggibile->tm_min,dataleggibile->tm_hour,
            dataleggibile->tm_year,dataleggibile->tm_mon,dataleggibile-
>tm_mday,nickname,stanza_lasciata)<0)
                return -1;
      }



      else if(fprintf(pfile,"\n\n//%d:%d:%d %d-%d-%d// l'utente %s  si e' disconnesso (da
stanza %c) ",dataleggibile->tm_sec,dataleggibile->tm_min,dataleggibile->tm_hour,
            dataleggibile->tm_year,dataleggibile->tm_mon,dataleggibile-
>tm_mday,nickname,stanza_lasciata)<0)
```

```
            return -1;
        }
        fflush(pfile);
        fclose(pfile);
        printf("\n***LOG DISCONNESSIONE UTENTE (A SEGUITO DI FIN) SALVATO\n");
        fflush(stdout);
        return 0;
}
```

C'e' poi la versione equivalente, che salva l'evento che occorre alla connessione di un nuovo utente, sempre nel logfile, ed e' la funzione clientconnesso_getinfo_salvalog() che prende in input gli stessi parametri. L'unica differenza e' che quando registriamo l'accesso di un nuovo utente, otteniamo anche l'ip e la porta dell'end point (client) connesso, e li inseriamo nel log file.
Inoltre non ci sono problemi relativi alla mancanza del nick da registrare (come nel caso della disconnessione del client).

• CLIENT

✓ MAIN:

Il main e' un ciclo do-while, nel quale decidiamo le impostazioni tcp-ip, nickname e possiamo connetterci. Da qui viene mandata in esecuzione la parte centrale del codice del client, ovvero la funzione chat() che gestisce la sessione di connessione al server, e che prende in input dal main la struct con gli indirizzi da bindare sul socket, e quello a cui connettersi.
La stanza viene memorizzata in una variabile locale alla funzione chat()

int chat(struct sockaddr_in *servind, struct sockaddr_in *nostrind,char *nickname,char *percorso)

Locale a questa funzione e' dichiarato un array di char, dove vengono memorizzati temporaneamente i messaggi inviati/ricevuti, che verranno di volta in volta scaricati nel file della cronologia (nella cartella dell'exe).
Viene chiesta la stanza nella quale entrare, e poi vengono lanciate le system call() per bindare il socket di connessione (il bind del socket lato connessione non e' strettamente necessario) e connetterlo al server.
In caso vada tutto a buon fine, viene stampato un messaggio di conferma della connessione al server. Inizia poi il ciclo portante dell'applicazione lato client.
Viene stampato per prima cosa un messaggio che ci indica dove digitare il messaggio (questo viene stampato prima di lanciare la select(), in modo tale che compaia prima che vengano

digitate lettere e venga premuto invio –in quanto questo rende gia' ready lo stdin)
Viene preparato poi il set di fd che passeremo alla select(), svuotandolo, e settando a 1 i bit relativi al fd dello stdin (fileno(stdin)) e del socket di connessione al server.

```
do{
        printf("\n***[[SCRIVI QUI]]:");
        fflush(stdout);
        FD_ZERO(&set1);
        tempo2.tv_sec=1500;
        FD_SET(fileno(stdin),&set1);
        FD_SET(connsfd,&set1);
```

Poi lanciamo la select(), passando anche una struct contenente il tempo massimo di attesa.
Se la select ritorna (allo scadere del tempo) 0, allora non e' successo nulla, non abbiamo ricevuto alcun messaggio, e non abbiamo scritto niente. In tal caso quindi chiudiamo la connessione, impostando su 0 la variabile keep4us che portera' all'interruzione del ciclo, indicando che siamo stati noi a terminare la sessione di chat ,e non il server.

```
if(select(connsfd+1,&set1,NULL,NULL,&tempo2)==0) {
                printf("\n\aSIAMO STATI INATTIVI PER TROPPO TEMPO
(1500s). AUTODISCONNESSIONE IN CORSO...\n");
                close(connsfd);
                keep4us=0;
        }
```

In caso contrario, continuiamo allo sblocco della select() e controlliamo se e' pronto in lettura il socket di connessione. Se la FD_ISSET() ritorna 1, allora abbiamo ricevuto dati dal server, quindi lanciamo una full_read() sul socket.
Se la full_read() ritorna un valore errato, se ci sono char nel buffer temporaneo della cronologia, li salva, e in ogni caso la funzione chat() chiude la connessione e ritorna -1.
Se invece la full_read() ritorna 0, allora il server ci ha mandato un FIN, quindi ha chiuso la connessione: in questo caso impostiamo a 0 keep4srv, e questo interrompera' il ciclo della sessione, e sara' interpretato facendo ritornare 0 dalla funzione chat.

```
if(FD_ISSET(connsfd,&set1)) {
                toreceive=calloc(300,sizeof(char));

                r=full_read(connsfd,toreceive,300);



                if(r<0) {printf("\nImpossibile leggere dal socket\n");
                        if(strlen(cronologiatemp)>0) // && if
```

```
if(salva_cronologia(cronologiatemp,nickname,percorso)<0)
                                        printf("\nImpossibile salvare parte della
cronologia");
                                free(toreceive);
                                //close(connsfd);
                                return -1;

                        }

                        if(r==0) {keep4srv=0;
                                 free(toreceive);
                        }
```

Nel caso in cui invece la full_read() legga qualcosa, allora abbiamo ricevuto un messaggio. Innanzitutto controlliamo che non sia una comunicazione di servizio da parte del server: se il messaggio contiene 14* nella parte iniziale, allora il client stampa direttamente quello che c'e' dal 15esimo char in poi (che e' il messaggio di servizio)
Altrimenti se non c'e' la stringa riservata all'inizio,tiriamo fuori il nick del mittente, e stampiamo secondo la forma adatta a rappresentare il messaggio ricevuto dall'utente mittente.
In entrambi i casi (comunicazione di servizio o messaggio normale) dobbiamo salvare il messaggio ricevuto nel buffer temporaneo di cronologia (se questo ha superato un certo limite pero', scarichiamo prima tutto sul file log del disco, utilizzando la funzione apposita)

```
else {

                                if(strncmp("**************",toreceive,14)==0) {
                                        printf("\n---SERVER:%s",toreceive+14);
                                        fflush(stdout);
                                }

                                else{
                                        strncpy(nickmittente,toreceive,14);
                                        nickmittente[14]='\0';
                                        printf("\n<<:%s
\nscrive:>>%s",nickmittente,toreceive+15);
                                        fflush(stdout);
                                }

                                if( (1200-strlen(cronologiatemp))<strlen(toreceive) )

if(salva_cronologia(cronologiatemp,nickname,percorso)<0)
                                          printf("\nImpossibile salvare parte della
cronologia");
                                strcat(cronologiatemp,"\n");
                                strcat(cronologiatemp,toreceive);
```

```
}
```

Poi proseguiamo, controllando se lo stdin e' risultato ready sotto la select().
Se lo stdin e' ready, vuol dire che abbiamo digitato dei char sulla tastiera, e abbiamo premuto invio.
Quindi copiamo il nickname sulla parte riservata del messaggio da inviare, eliminiamo da questa parte riservata tutti gli spazi, i newline e terminatore stringa che possono esserci, in modo tale che potremo creare un'unica stringa unendo nickname,char stanza, e messaggio vero e proprio. Il quindicesimo char diventa il numero della stanza +48, quindi convertito in char questo int diventa proprio il carattere rappresentante il numero. Come sedicesimo elemento mettiamo un \0, in modo tale che facendo poi uno strcat tra la stringa cosi' ottenuta ed il testo precedentemente preso con la fgets dallo stdin (che conteneva i char precedentemente inviatigli da tastiera) venga ottenuto il risultato cercato. A questo punto controlliamo che il messaggio digitato da tastiera non sia il comando quit (ed in tal caso viene posto a 0 keep4us…)

```
if(FD_ISSET(fileno(stdin),&set1))  {

                tosend=calloc(300,sizeof(char));
                strncpy(tosend,nickname,15);
                f=0;
                while(f<15)
                   {if(tosend[f]=='\n' || tosend[f]=='\0')
                      tosend[f]=' ';
                      f++;
                   }
                testo=calloc(285,sizeof(char));
                fgets(testo,285,stdin);
                tosend[14]=(char)(stanza+48);
                tosend[15]='\0';
                strcat(tosend,testo);

                if(strcmp(tosend+15,"quit\n")==0) {
                        printf("\nChiusura sessione chat in corso\n");
                        free(tosend); free(testo); keep4us=0;
                }
```

Altrimenti inviamo il messaggio appositamente costruito al server. Se la write ritorna un valore <=0, se ci sono messaggi memorizzati nel buffer temp della cronologia, scarichiamo tutto nel file apposito, e in ogni caso chiudiamo la connessione e facciamo ritornare -1 alla funzione.
Altrimenti se la scrittura sul socket e' andata a buon fine, salviamo il messaggio nel buffer della cronologia (e se questo e' gia' pieno oltre un certo limite, agiamo come sopra)

```
else{
                                do w=write(connsfd,tosend,300);
                                        while(w<0 && errno==EINTR);

                                if(w<=0) {
                                        printf("\nImpossibile scrivere sul socket\n");
perror("\nERRORE:\n");
                                        if(strlen(cronologiatemp)>0) // && if

if(salva_cronologia(cronologiatemp,nickname,percorso)<0)
                                                printf("\nImpossibile salvare parte della
cronologia");
                                        //close(connsfd)
                                        free(testo);
                                        free(tosend);
                                        return -1;
                                }


                                if( (1200-strlen(cronologiatemp))<strlen(tosend) )

if(salva_cronologia(cronologiatemp,nickname,percorso)<0)
                                                printf("\nImpossibile salvare parte della
cronologia");

                                strcat(cronologiatemp,"\n");
                                strcat(cronologiatemp,tosend);



                        }
```

Il ciclo continua se entrambe le variabili keep4us e keep4srv sono impostate ancora su 1.

```
…}
… while(keep4us && keep4srv);
```

All'uscita del do-while, vuol dire che la sessione sta chiudendo, e quindi prima di ritornare controlla se e' necessario salvare residui di messaggi rimasti nel buffer della cronologia temp. Ed in base al valore assunto dalle variabili di controllo del ciclo, ritorna il valore indicativo al main, che interpretera' in modo tale da poter stampare un messaggio che avvisi l'utente di chi ha concluso la sessione di chat (se lui o il server)

```
if(!keep4us) {close(connsfd); return 1;}
```

if(!keep4srv) {close(connsfd); return 0;}

Una funzione importante lato client e' quella di salvataggio della cronologia.

• FUNZIONE ESTERNA AL MAIN

int salva_cronologia(char *dasalvare,char *nickname,char *percorsofinale){…

Questa funzione prende in input il percorso del file sul disco e la stringa da salvare (il nickname lo passavo in fase di debugging per controllare che partisse per il nick esatto, ora e' lasciata come commento l'istruzione relativa).
All'interno della funzione della cronologia viene preso il tempo di sistema, e salvato in una variabile di tipo time_t. Poi questo viene trasformato in un formato leggibile riempiendo la relativa struct di tipo struct tm:

```
time_t datacodificata;
struct tm *data;
time(&datacodificata);
data=localtime(&datacodificata);
```

Poi apriamo il file nel percorso sul disco indicato, e ci scriviamo sopra con la funzione fprintf, salvando i parametri di tempo ed il buffer temporaneo della cronologia, passato in input alla funzione.

```
//printf("%s",nickname); istruzione lasciata ora come commento//
FILE *pfile;
if((pfile=fopen(percorsofinale,"a"))==NULL)
    {
      return -1;
    }




else{
```

```
    if(fprintf(pfile,"\n\n%d:%d:%d  %d/%d/%d\n",data->tm_sec,data->tm_min,data->tm_hour,data->tm_year,data->tm_mon,data->tm_mday)<0) return -1; //ogni volta che scarichiamo il buff temp della cronologia sul file, scriviamo anche prima la data
    if(fprintf(pfile,"%s",dasalvare)<0) return -1;
    fflush(pfile); // svuotamento del pnt a file prima di chiuderlo
    fclose(pfile);
    return 0;
}
```

• Istruzioni per la compilazione

Come ambiente di sviluppo ho usato codeblock, e come compilatore gcc (gnu compiler) su Ubuntu.
Compilare possibilmente i 2 file .exe in 2 cartelle diverse, anche se tuttavia non dovrebbero esserci comunque problemi in quanto gli applicativi ottengono dinamicamente il percorso sul disco.

• Istruzioni per l'esecuzione

Per eseguire basta lanciare gli exe, lanciare il server e poi il client, e settare lato client le impostazioni per la connessione TCP/IP ed il nickname.
I file della cronologia e del log del server vengono salvati nelle stesse cartelle dove sono contenuti gli exe.
Gli applicativi sono stati testati sia sull'indirizzo 127.0.0.1 di loopback, sia su due macchine diverse (entrambe su ubuntu) connesse alla rete universitaria, e non sono sorti problemi.
Fare attenzione che certe volte, se si chiude in maniera anomala (ma non solo) l'applicazione, e si prova a bindare dopo pochissimo sulle stesse porte, non essendo ancora state liberate essendoci un tempo minimo di attesa, quest'ultime non sono disponibili (viene stampato un messaggio apposito per informare l'utente).
Il modo corretto di chiudere una sessione di chat lato client, e' di scrivere la parola quit