Assignment #2: Relazione

Programmazione di reti

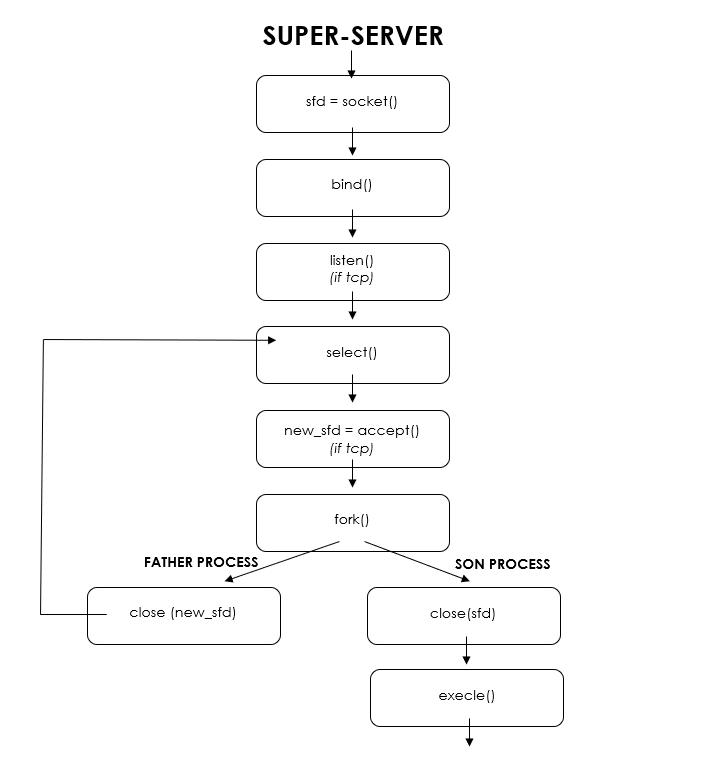
baiardi martina

lombardini alessandro

**TASK 1**

Il superserver crea una socket per ogni servizio che mette a disposizione e le associa, tramite delle chiamate alla funzione *bind(),* alle porte stabilite all’interno di inetd.conf. Se il servizio è TCP, viene invocata la funzione *listen()*, che pone in ascolto la socket creata di eventuali richieste di connessione. Viene infine invocata la funzione *select()*, che permette di monitorare contemporaneamente più socket in attesa che queste vengano contattate da un client.

Giunta una nuova richiesta, se la connessione è TCP viene invocata la *accept()* che apre una socket di connessione con il client richiedente. Viene effettuata una *fork()*, cioè viene copiata l’immagine del processo del superserver in un nuovo processo, chiamato processo figlio, che viene poi opportunamente sostituito con l’eseguibile del server richiesto per il servizio, attraverso la funzione *execle()*. Il processo padre, nonché il superserver, torna in ascolto di nuove richieste tramite una nuova chiamata alla funzione *select().*

****

**TASK 2**

# Implementazione del Super-Server

Il superserver, per prima cosa, effettua una lettura del file di configurazione *inetd.conf* e salva le informazioni contenute al suo interno dentro una struttura dati preventivamente creata. Al suo interno vengono memorizzati, per ogni tipologia di servizio:

* *Protocollo di trasporto*, TCP o UDP
* *Tipo di servizio*, wait o nowait
* *Porta del servizio*, porta su cui è possibile richiedere il servizio
* *Indirizzo completo del servizio*, indirizzo del file dell’applicativo server che offre quel servizio.
* *Nome del servizio*, nome dell’applicativo del server richiesto
* *Socket File Descriptor*, socket associata porta del servizio
* *Process Identifier*, PID del processo figlio creato per soddisfare una richiesta di quel servizio. Questo parametro serve per gestire i servizi di tipo *wait*.

A questo punto il superserver procede alla creazione delle socket che devono essere associate ai diversi servizi disponibili. Per ciascun servizio:

1. Viene invocata la funzione ***socket()***
   * Se la socket è richiesta di tipo TCP, la funzione conterrà i parametri: AF\_INET (Famiglia di indirizzi IPv4), SOCK\_STREAM (Socket con connessione), IPPROTO\_TCP (Protocollo TCP).
   * Se la socket è richiesta di tipo UDP, la funzione conterrà i parametri: AF\_INET (Famiglia di indirizzi IPv4), SOCK\_DGRAM (Socket senza connessione), IPPROTO\_UDP (Protocollo UDP)
2. **Viene inizializzata la struttura dati contenente le informazioni sul server, di tipo *sockaddr\_in*.**
3. **Viene chiamata la funzione *bind()*, che associa alla nuova socket l’indirizzo del servizio descritto nella struttura dati.**
4. Se la socket generata è di tipo TCP viene invocata la funzione ***listen()***, che consente di porre una socket in ascolto di richieste di richieste di connessione.

Per gestire la chiusura dei processi figli del superserver, viene impostata la procedura ***handle\_signal()*** tramite la funzione ***signal()***.

Il superserver entra in un loop senza fine all’interno del quale si occupa di gestire le richieste dei client sulle socket precedentemente create; questa operazione viene gestita attraverso la funzione ***select()***. Quando arriva la richiesta di un servizio, per prima cosa si controlla se questo è di tipo TCP, in tal caso viene invocata la funzione ***accept()*** che crea una nuova socket su cui verranno trasmessi i dati della comunicazione con quello specifico client.

Viene poi invocata la ***fork()***, che crea un processo figlio con la stessa immagine di quello corrente. A questo processo figlio verranno chiusi i file descriptor 0,1,2, che corrispondono rispettivamente ai canali di input, output ed error; verranno poi sostituiti con il socket file descriptor della nuova socket creata, predisposta alla comunicazione con il client. Una volta chiusi, viene sostituita l’immagine del processo figlio con quella del processo server utile per soddisfare il servizio richiesto. Il processo padre ha il compito di rimuovere la socket di benvenuto del servizio aperto dalla lista fd\_set fornita alla select, solo nel caso in cui questo sia di tipo *wait*; in tal caso viene inoltre registrato, all’interno della struttura dati, il PID del processo figlio. Se di tipo *nowait*, invece, ritorna direttamente alla *select()*. Processo padre e processo figlio hanno rispettivamente il compito di chiudere la socket appena creata e lo socket di benvenuto.

Infine, per gestire i segnali SIGCHLD inviati dai processi figli quando terminano, è stata disposta la procedura *handle\_signal().* Viene controllato che il segnale arrivato sia effettivamente un SIGCHLD e, tramite una chiamata *wait()*, è possibile ottenere il PID del processo terminato. Essendo il processo già morto la chiamata, che di per se è bloccante, restituirà immediatamente il risultato. Se il processo terminato si occupava di soddisfare una richiesta di servizio di tipo wait, la socket di benvenuto di quel particolare servizio viene riaggiunta alla lista fd\_set; questo consentirà alla *select()* di riprendere a controllarla per gestire nuove richieste. Per sapere se il servizio è di tipo *wait* è sufficiente ricercare all’interno della struttura dati se uno dei tanti servizi *wait* disponibili abbia nel campo *Process Identifier* un PID segnato identifico a quello ottenuto; in tal caso viene rimosso. E’ importante ricordarsi che la signal interrompe la select(), è dunque necessario verificare questa eventualità.

# Istruzioni utilizzate per la compilazione

Il codice è stato compilato attraverso il supporto di un Makefile, qui riportato:

all: superserver.exe udpClient.exe udpServer.exe tcpClient.exe tcpServer.exe

superserver.exe: superserver.o

gcc ${CFLAGS} -o superserver.exe superserver.o

superserver.o: superserver.c

gcc -c ${CFLAGS} superserver.c

udpClient.exe: udpClient.o

gcc ${CFLAGS} -o udpClient.exe udpClient.o

udpClient.o: udpClient.c

gcc -c ${CFLAGS} udpClient.c

udpServer.exe: udpServer.o

gcc ${CFLAGS} -o udpServer.exe udpServer.o

udpServer.o: udpServer.c

gcc -c ${CFLAGS} udpServer.c

tcpClient.exe: tcpClient.o

gcc ${CFLAGS} -o tcpClient.exe tcpClient.o

tcpClient.o: tcpClient.c

gcc -c ${CFLAGS} tcpClient.c

tcpServer.exe: tcpServer.o

gcc ${CFLAGS} -o tcpServer.exe tcpServer.o

tcpServer.o: tcpServer.c

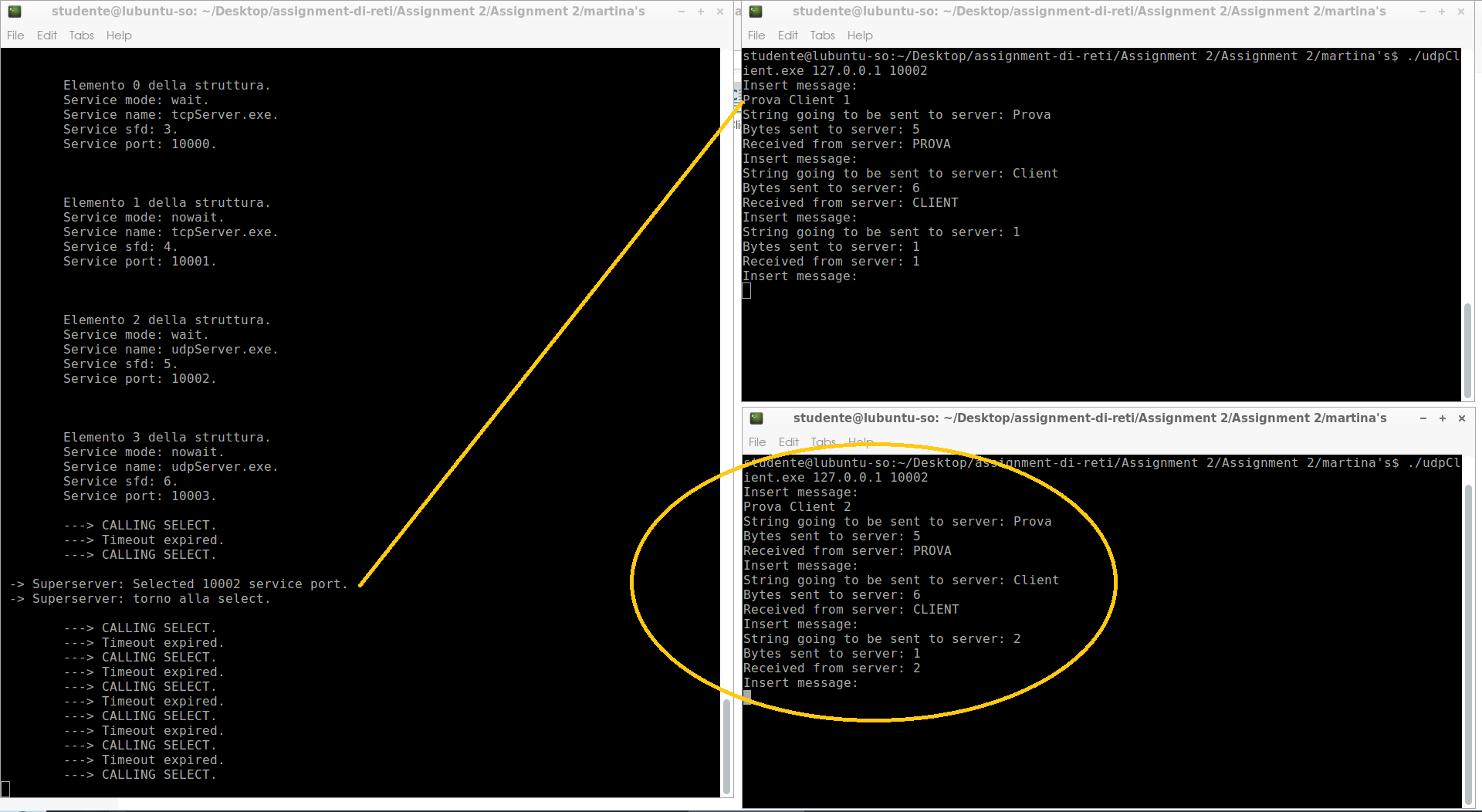
gcc -c ${CFLAGS} tcpServer.c

.PHONY: clean

clean:

-rm -f \*.exe \*.o

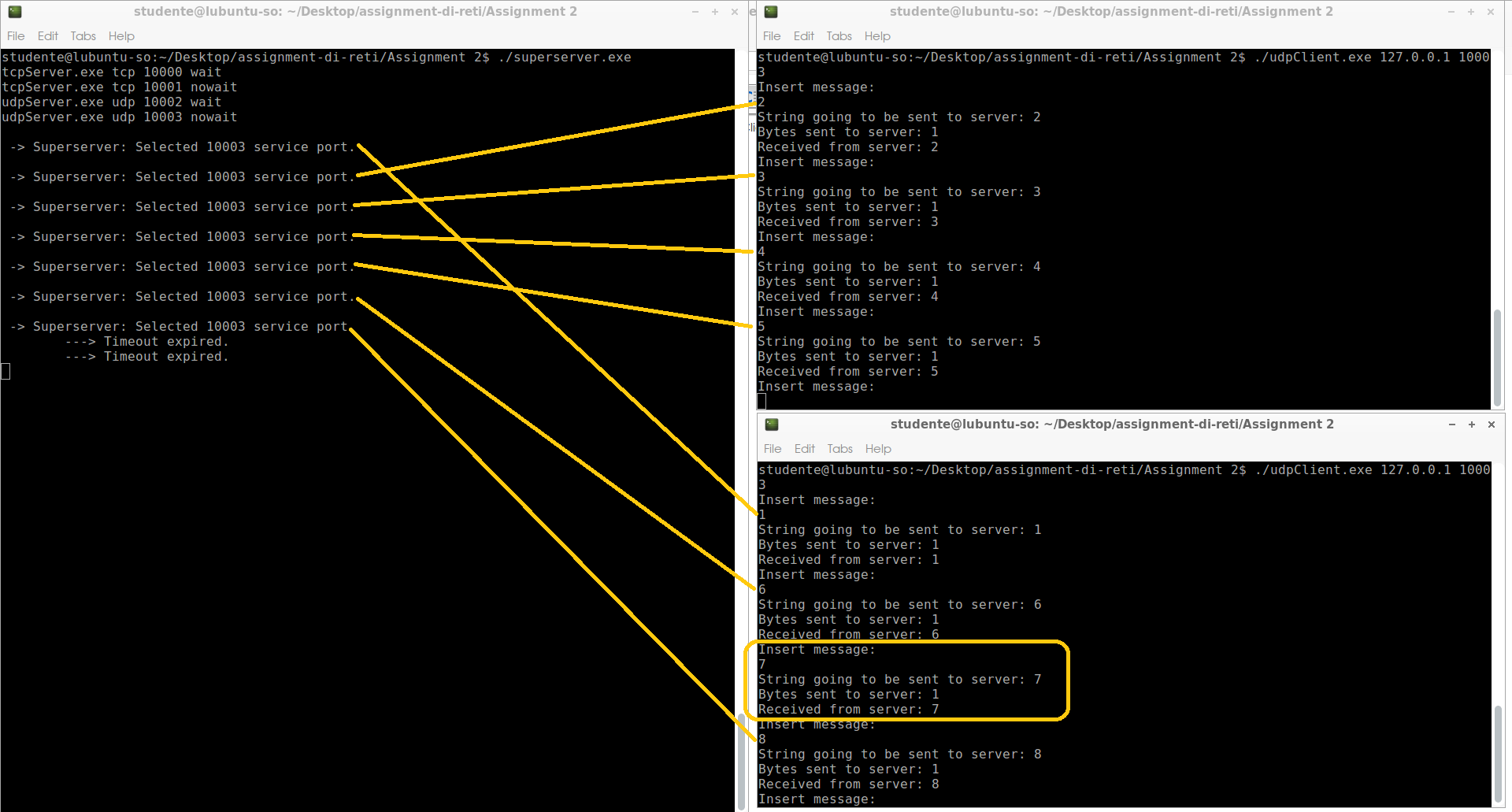
# udpServer behavior in wait mode

Quando il superserver riceve una richiesta da parte di un client udp per un servizio udp in wait mode, esso si occupa di eseguire un udpServer per gestirne la richiesta. Nel momento in cui sopraggiunge un nuovo client a richiedere lo stesso servizio, il superserver non accetta richieste in quanto in wait mode, dunque non vengono eseguiti ulteriori server. Il server precedentemente attivato rimane attivo sempre sulla stessa porta, quindi risponde anche al nuovo client.

Abbiamo effettuato con il client A la richiesta “Prova Client 1”, ottenendo l’allocazione di un processo server che risponde correttamente al messaggio e rimane in attesa di ulteriori messaggi. E’ stato poi utilizzato un secondo client (B) per effettuare una seconda richiesta, “Prova Client 2”, al quale però il superserver, rispettando i vincoli della wait mode, non ha allocato alcun server. Il client B ha ottenuto ugualmente risposta in quanto il server dedicato ad A si è occupato di tale richiesta.

# udpServer behavior in no-wait mode

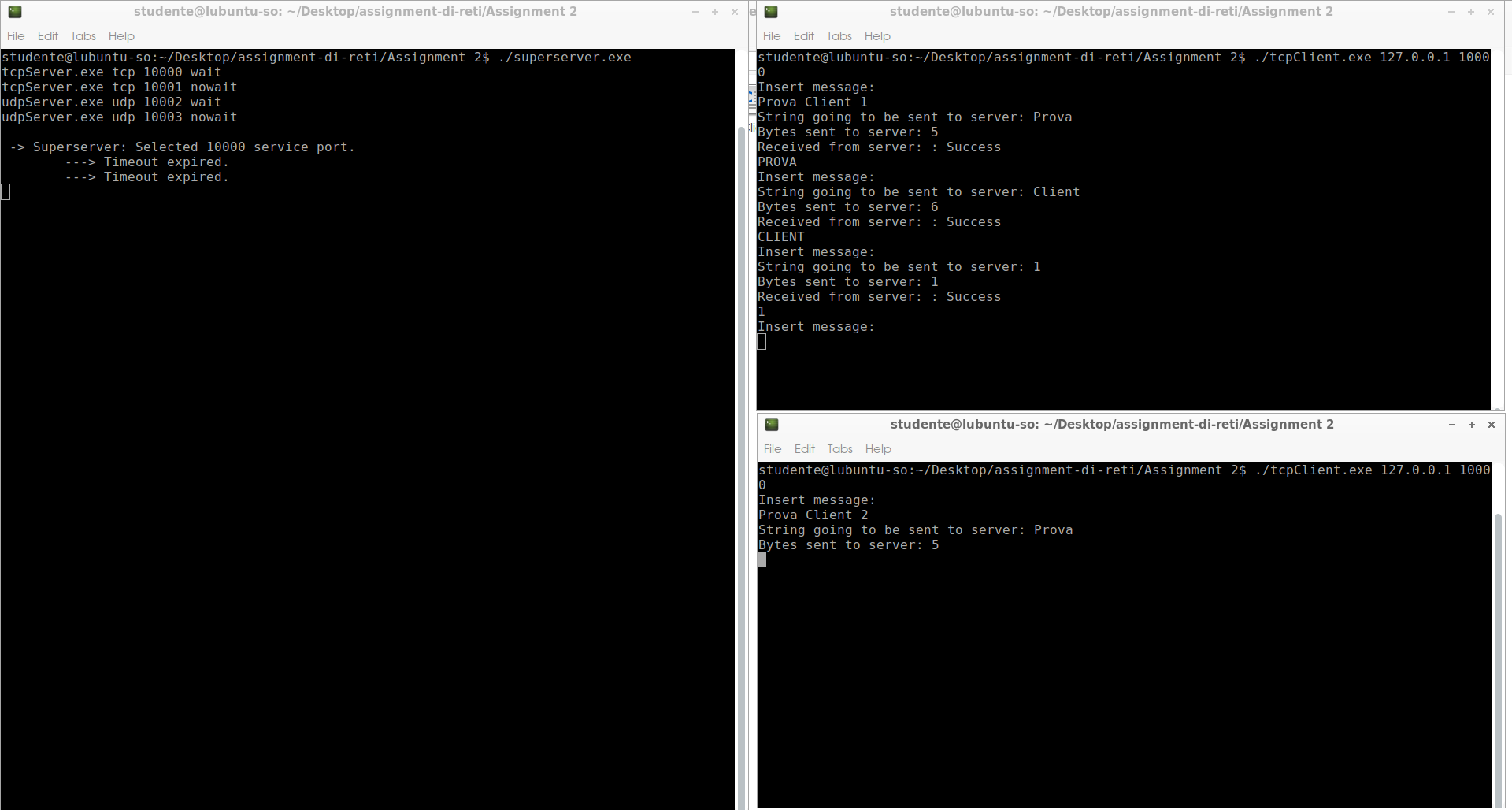
Il server udp, nel momento dell’invio di un messaggio, effettua una richiesta al superserver. Quest’ultimo genera un processo udpServer, che si occuperà di rispondere alla richiesta del client. Siccome il servizio richiesto è di tipo no-wait, il superserver non toglie mai l’ascolto sulla porta del servizio, creando così una situazione di “concorrenza” tra superserver e i server udp precedentemente creati, siccome tutti sulla stessa porta. Infatti, quando si inviano dei messaggi da parte del client, non è possibile prevedere chi tra superserver e udpServer risponderà alla nostra richiesta, infatti nello screen riportato, si è cercato di evidenziare questo fenomeno: se il messaggio viene ricevuto prima dal superserver, viene aggiunto un nuovo server sulla stessa service-port, che poi risponde alla richiesta; se invece il messaggio viene ricevuto prima da udpServer, quest’ultimo risponde direttamente al client.



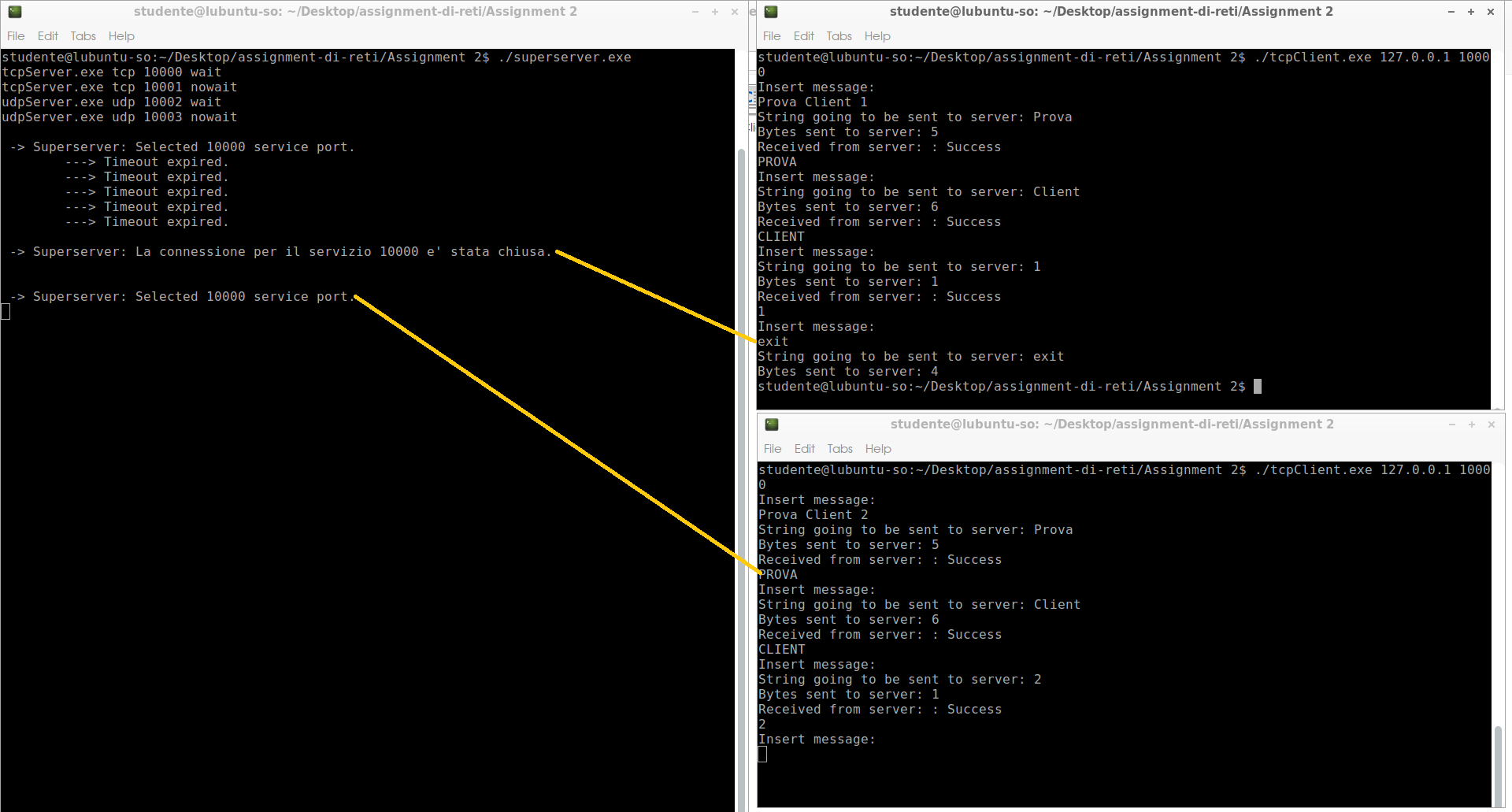
Per evidenziare meglio questo fenomeno, è stata inserita una *usleep()* al processo padre subito prima di tornare alla select, la quale permette di effettuare una attesa in micro-secondi, dei quali ne abbiamo inseriti 5. Tale scelta è stata valutata perché altrimenti non sarebbe stato possibile controllare il flusso di pacchetti inviati dal client.

# tcpServer behavior in wait mode

Nel momento in cui il client effettua una richiesta al server, questo alloca il server tcp sulla connessione e rimuove il servizio dalla lista della select(). Da questo avvenimento in avanti, il client ha una connessione privilegiata con il TCP Server, con il quale si scambia i messaggi. Se un nuovo client effettua una richiesta di connessione, non riceve risposta dal server e il superserver non lo notifica, perché quel servizio lo ha disabilitato.



Nel momento però in cui il primo client chiude la connessione, il server a lui connesso viene terminato e subito dopo il superserver alloca il tcp server con il secondo client che era in attesa di una risposta dal server da prima.



# tcpServer behavior in no-wait mode

durante una comunicazione no-wait, i client effettuano due richieste e per ciascuno di essi viene allocato un TCP Server che risponde ai loro messaggi, tutto ciò avviene parallelamente senza problemi di sincronizzazione tra loro.

