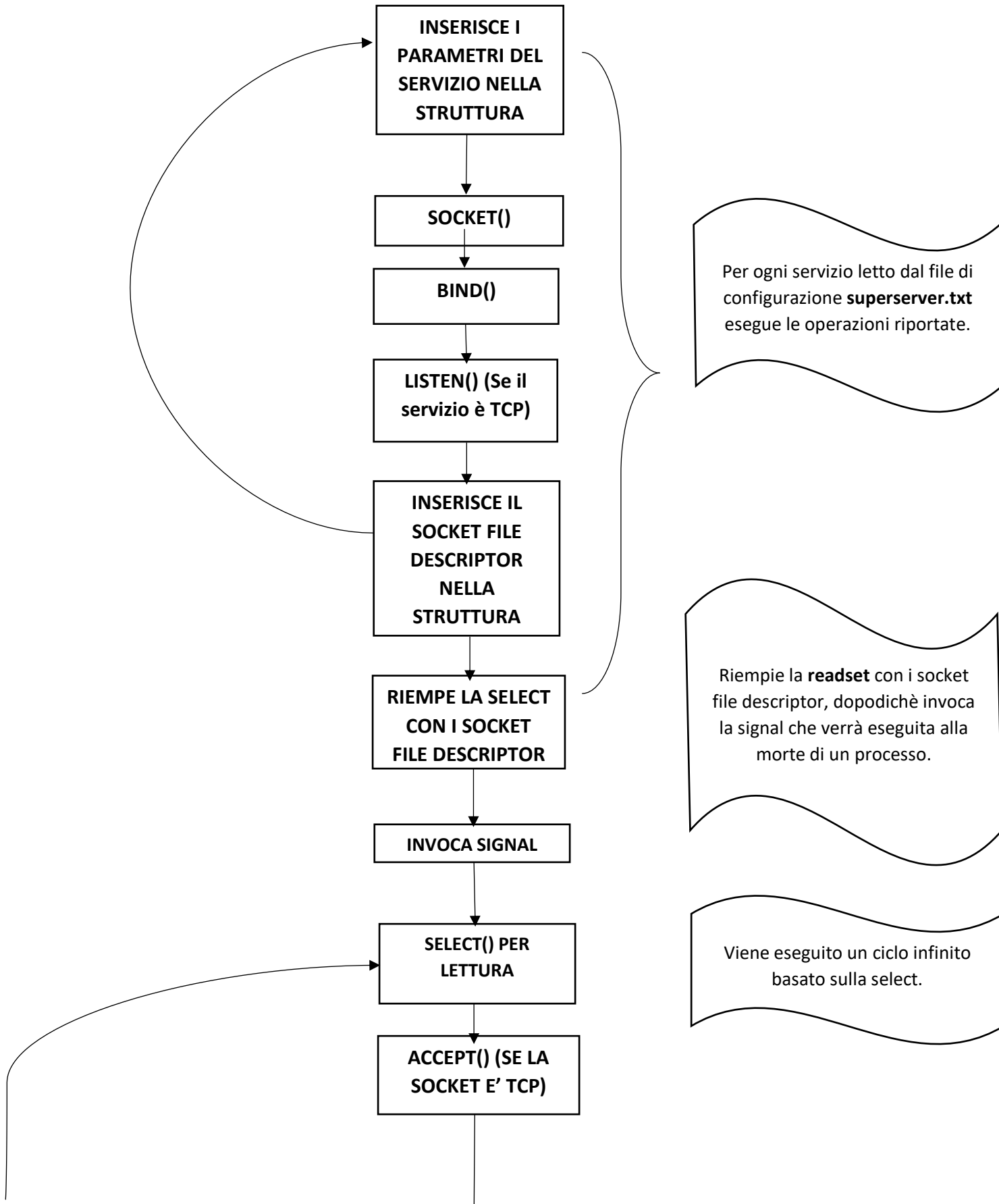
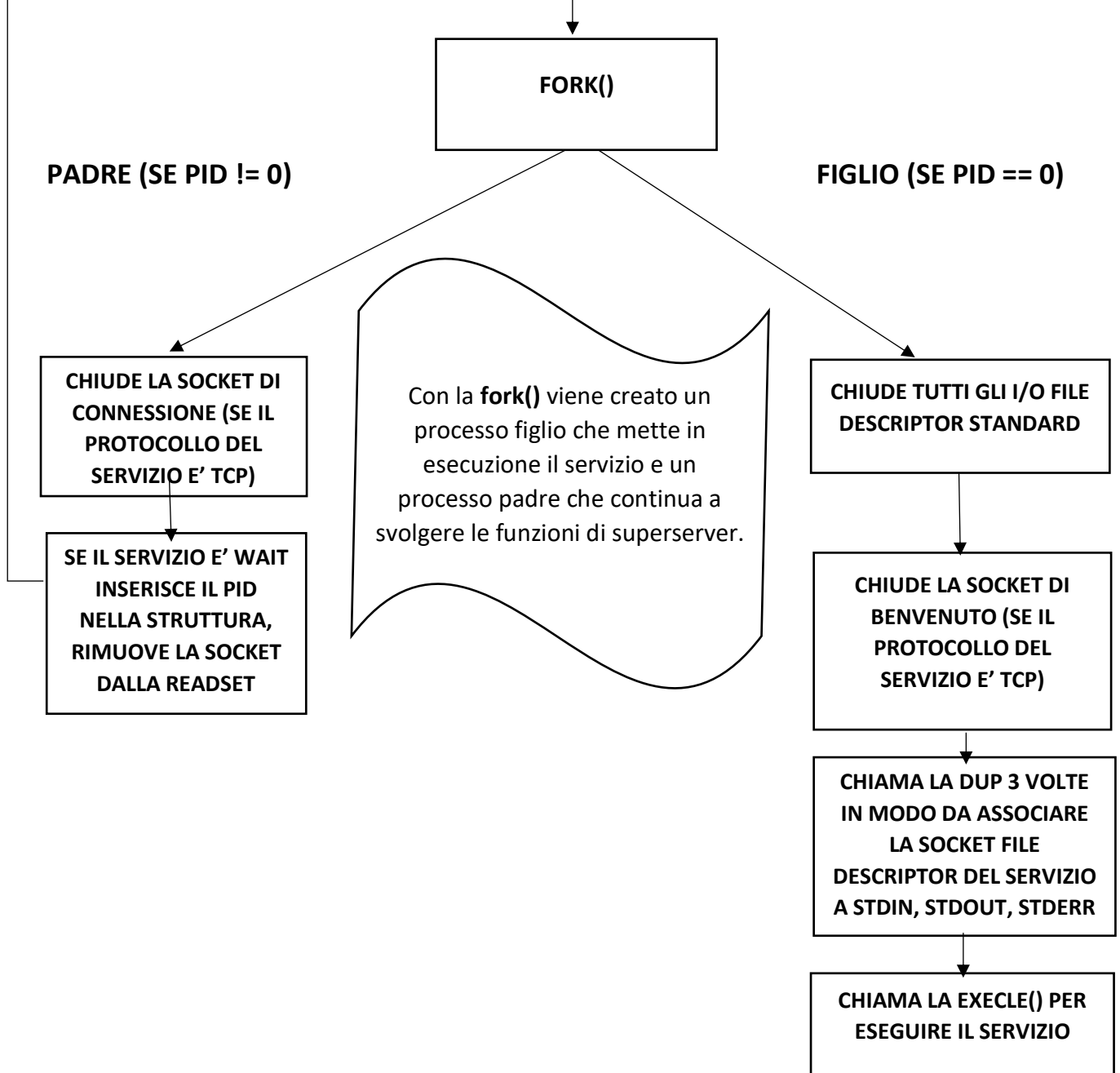


TASK 1

SCHEMA SUPERSERVER





TASK 2

Progettazione del superserver

Abbiamo progettato il superserver disaccoppiando la parte di configurazione e gestione delle strutture dati dall'effettivo nucleo operativo: abbiamo infatti gestito in apposite funzioni la lettura dal file di configurazione e la scrittura nel vettore di strutture dati (che descrivono le caratteristiche di ogni servizio), e la gestione del readset utilizzato dalla select, mentre all'interno del main abbiamo inserito il loop infinito del server all'interno del quale esso effettua I/O multiplexing tra i vari servizi avvalendosi della funzione select e della funzione fork (ogni servizio viene eseguito all'interno di un processo figlio lanciato dal superserver stesso).

Istruzioni sulla compilazione del superserver

Per comodità abbiamo creato un Makefile con le regole di compilazione del superserver e anche dei servizi descritti all'interno del file di configurazione.

In particolare:

```
superserver.exe: superserver.o
```

```
gcc -o superserver.exe superserver.o
```

```
superserver.o: superserver.c
```

```
gcc -c superserver.c
```

Test Report

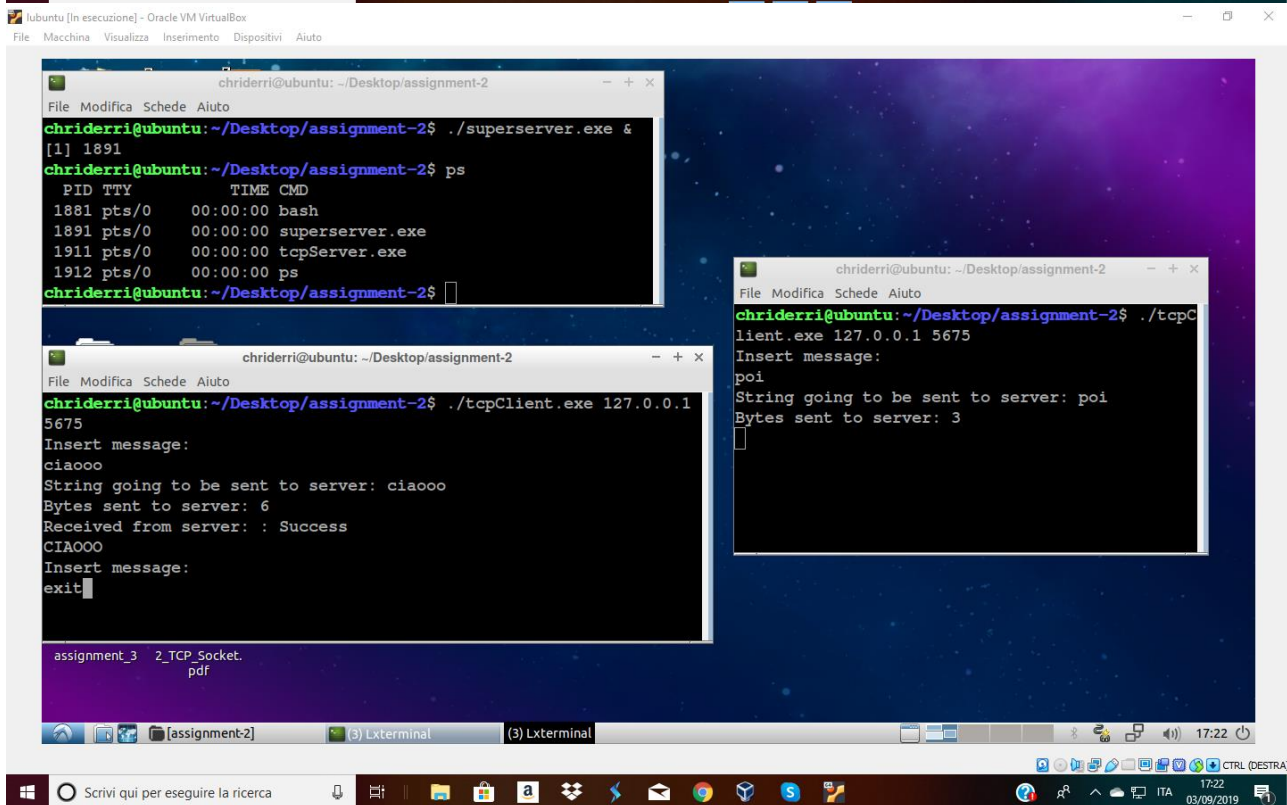
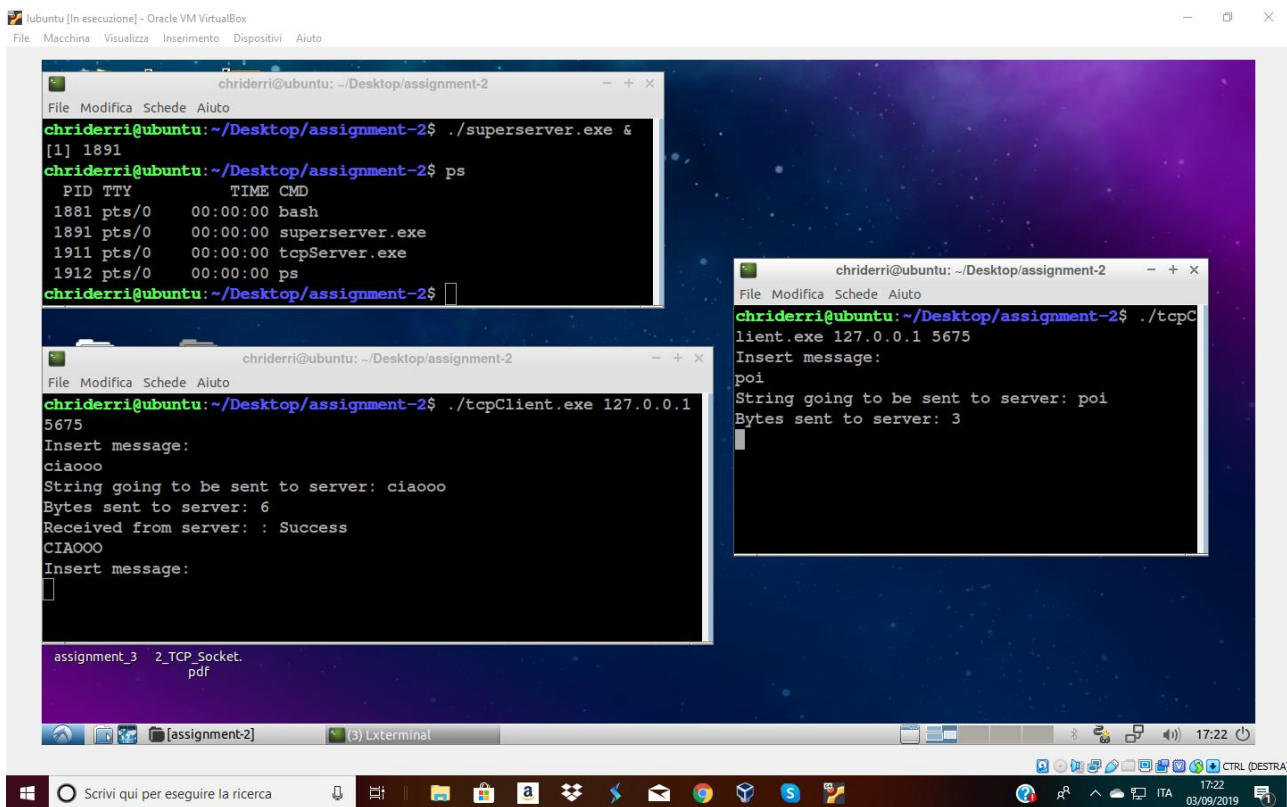
Abbiamo testato il superserver con i vari servizi presenti all'interno dell'assignment.

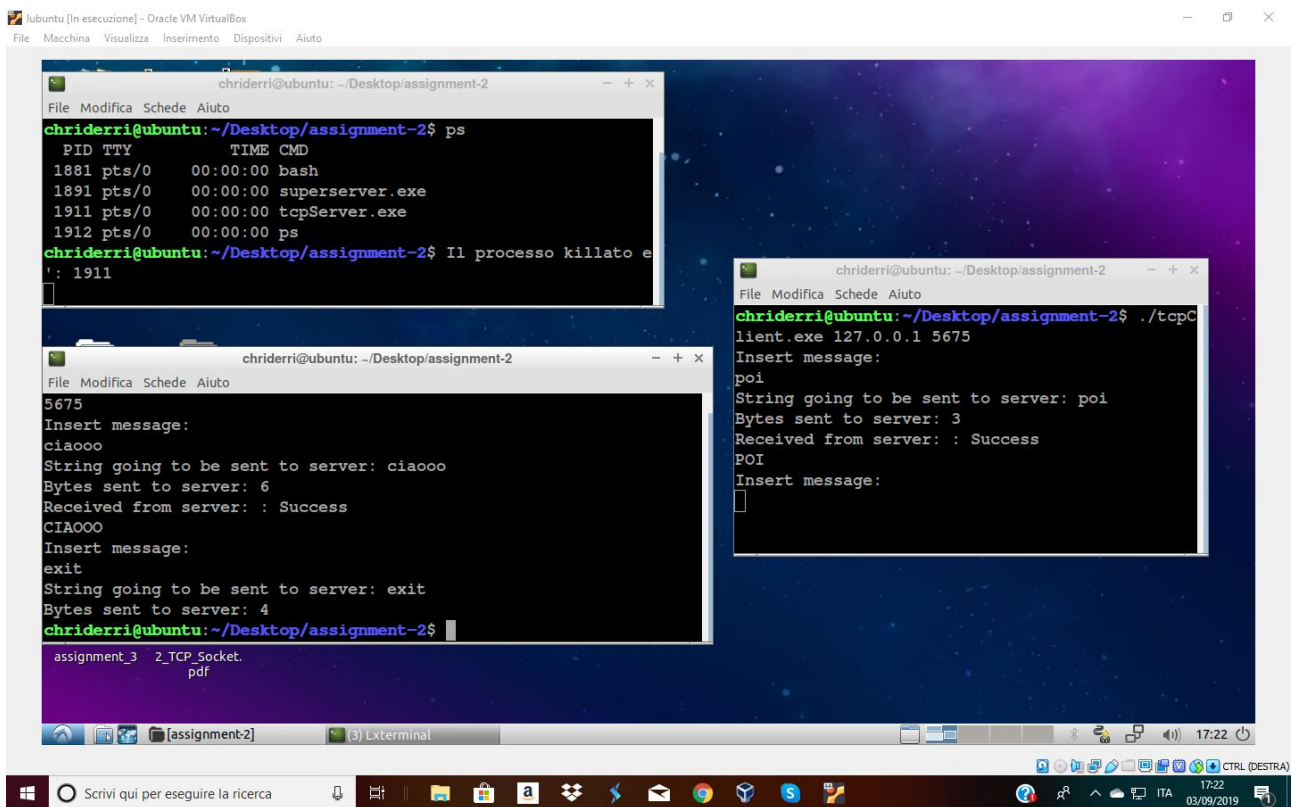
TCP

Per quanto riguarda i servizi che utilizzano il protocollo TCP abbiamo notato che:

WAIT

In modalità wait il superserver gestisce una richiesta alla volta evitando che vengano serviti due client contemporaneamente.

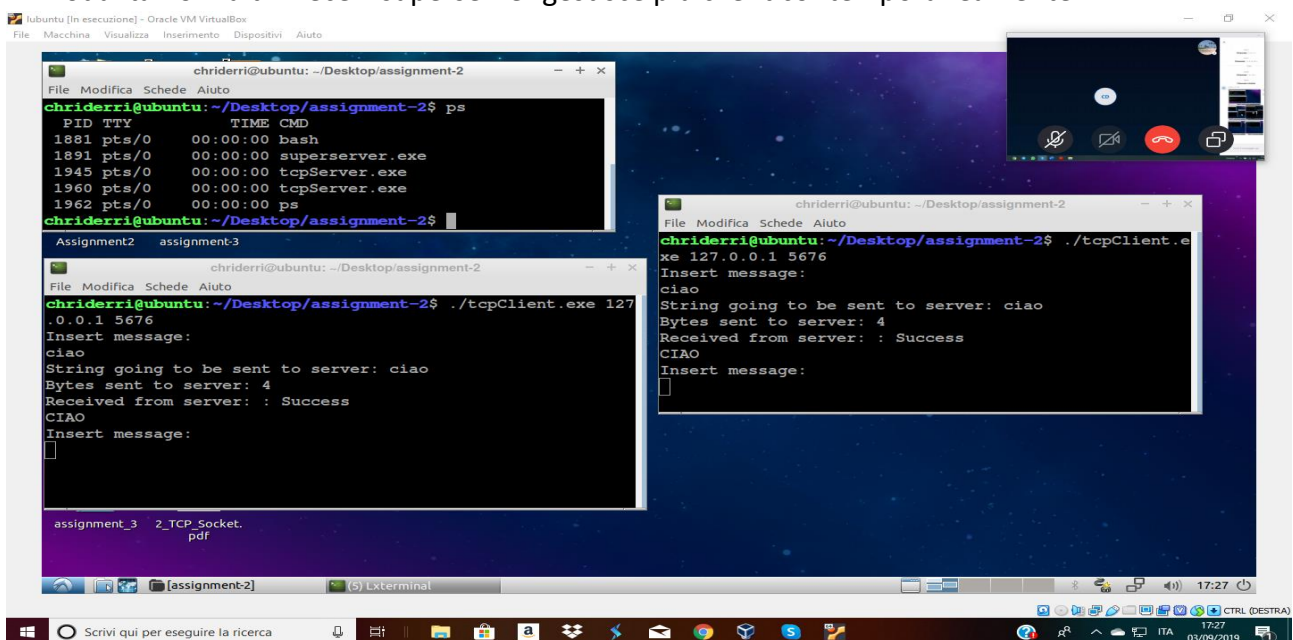




Il primo client viene gestito immediatamente mentre il secondo rimane in attesa che il primo finisca la comunicazione. Quando questa viene terminata, eventuali richieste in attesa vengono soddisfatte sempre secondo la politica "wait".

NO-WAIT

In modalità no-wait invece il superserver gestisce più client contemporaneamente.



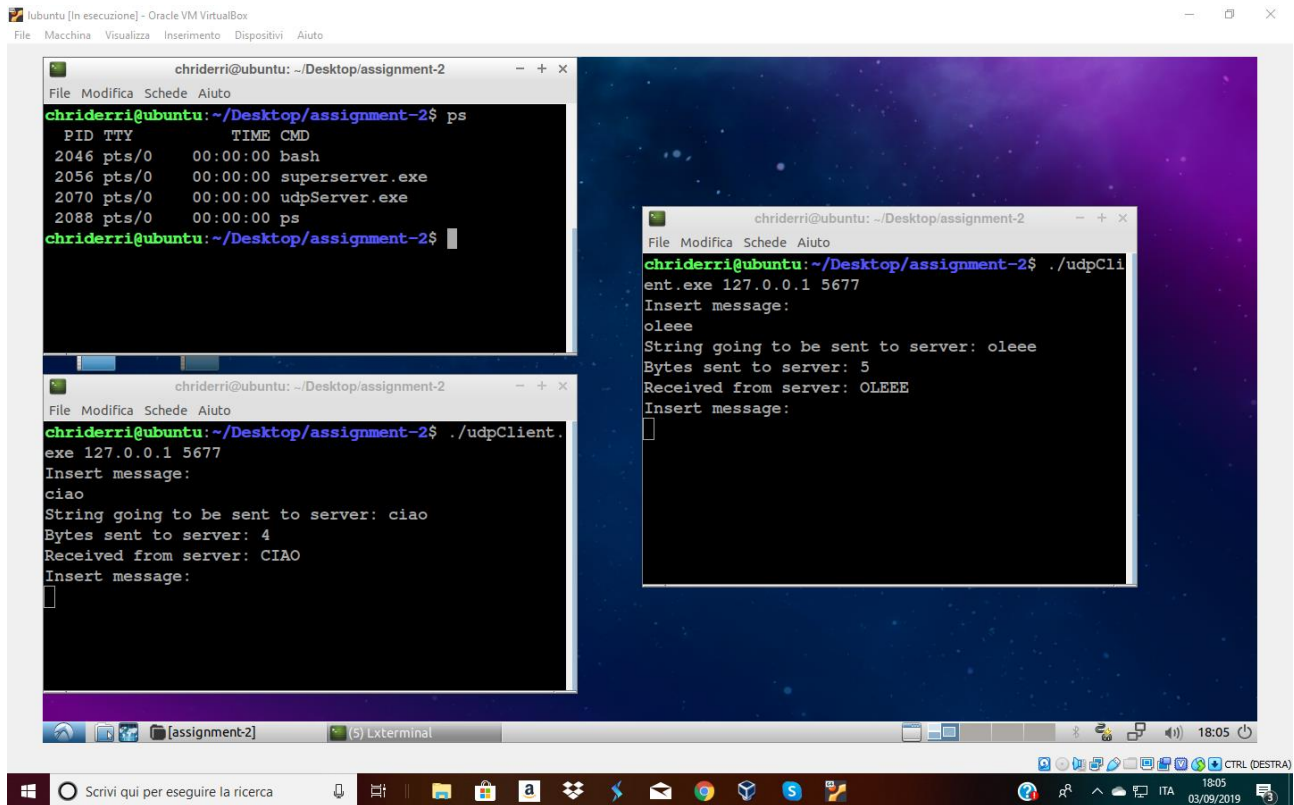
Il superserver è stato lanciato in background proprio per mostrare che i client vengono gestiti da processi eseguiti in parallelo.

UDP

Per quanto riguarda i servizi che utilizzano il protocollo UDP abbiamo notato che:

WAIT

A differenza del servizio con protocollo TCP in modalità WAIT, questa volta si è riscontrato che lo stesso servizio eseguito su due bash differenti che però vengono eseguite sulla stessa macchina e fanno richiesta sulla stessa porta vengono viste come un unico client, in quanto UDP non è un protocollo orientato alla connessione.



```
chriderri@ubuntu: ~/Desktop/assignment-2
File Modifica Schede Aiuto
chriderri@ubuntu:~/Desktop/assignment-2$ ps
PID TTY TIME CMD
2046 pts/0 00:00:00 bash
2056 pts/0 00:00:00 superserver.exe
2070 pts/0 00:00:00 udpServer.exe
2088 pts/0 00:00:00 ps
chriderri@ubuntu:~/Desktop/assignment-2$

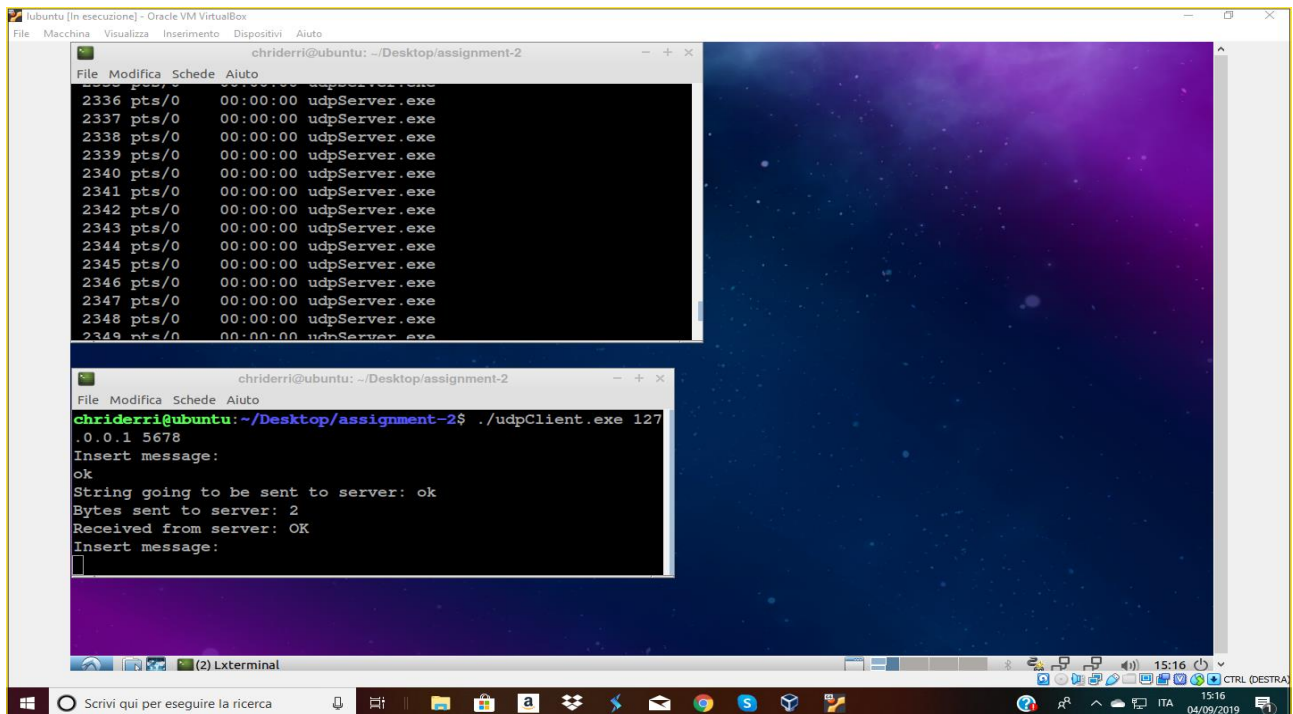
chriderri@ubuntu:~/Desktop/assignment-2$ ./udpClient.exe 127.0.0.1 5677
Insert message:
ciao
String going to be sent to server: ciao
Bytes sent to server: 4
Received from server: CIAO
Insert message:

chriderri@ubuntu:~/Desktop/assignment-2$ ./udpClient.exe 127.0.0.1 5677
Insert message:
oleee
String going to be sent to server: oleee
Bytes sent to server: 5
Received from server: OLEEE
Insert message:
```

Infatti viene lanciato un processo solo e all'interno di questo vengono soddisfatti entrambi i client.

NO-WAIT

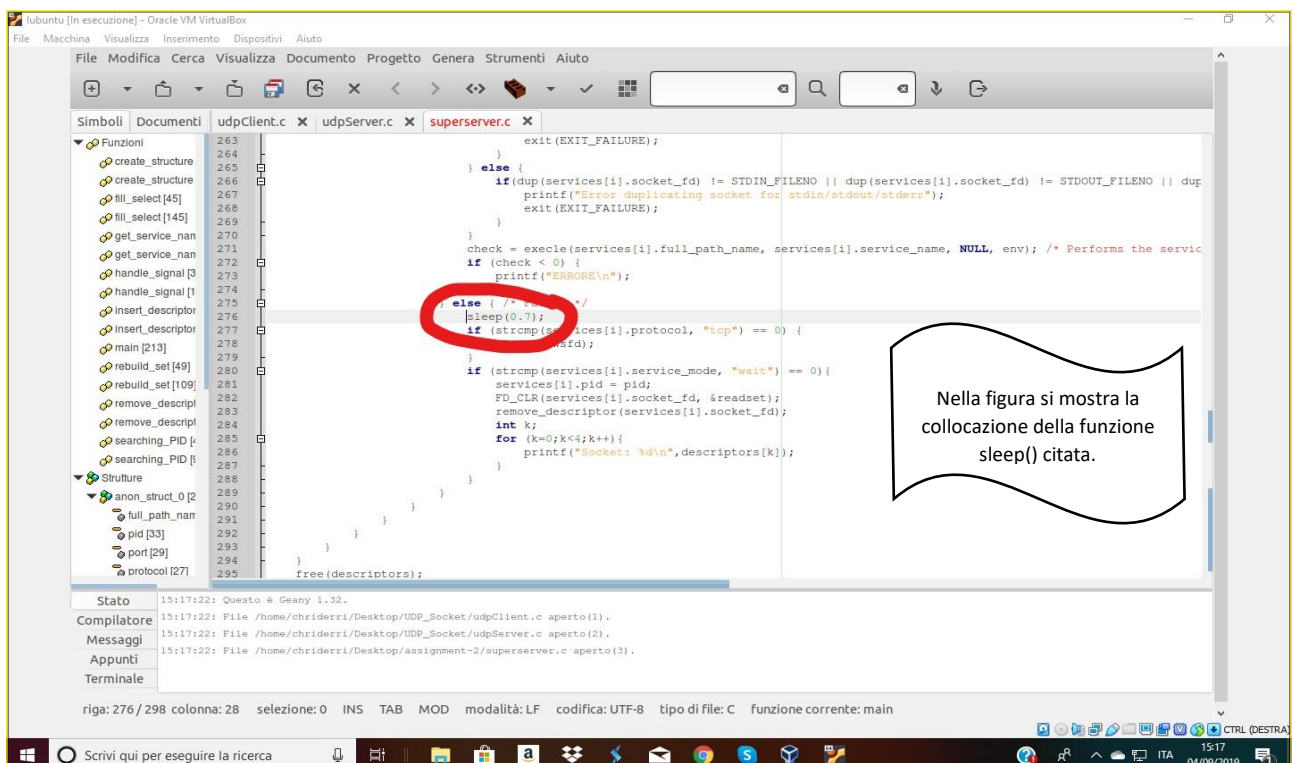
Per quanto riguarda invece la nostra esperienza con il servizio che si appoggia sul protocollo UDP in modalità NO-WAIT abbiamo notato che la non reliability di UDP comporta la generazione di un cospicuo numero di processi successivamente all'invio dei dati da parte del client.



```
chriderri@ubuntu: ~/Desktop/assignment-2
2336 pts/0 00:00:00 udpServer.exe
2337 pts/0 00:00:00 udpServer.exe
2338 pts/0 00:00:00 udpServer.exe
2339 pts/0 00:00:00 udpServer.exe
2340 pts/0 00:00:00 udpServer.exe
2341 pts/0 00:00:00 udpServer.exe
2342 pts/0 00:00:00 udpServer.exe
2343 pts/0 00:00:00 udpServer.exe
2344 pts/0 00:00:00 udpServer.exe
2345 pts/0 00:00:00 udpServer.exe
2346 pts/0 00:00:00 udpServer.exe
2347 pts/0 00:00:00 udpServer.exe
2348 pts/0 00:00:00 udpServer.exe
2349 pts/0 00:00:00 udpServer.exe

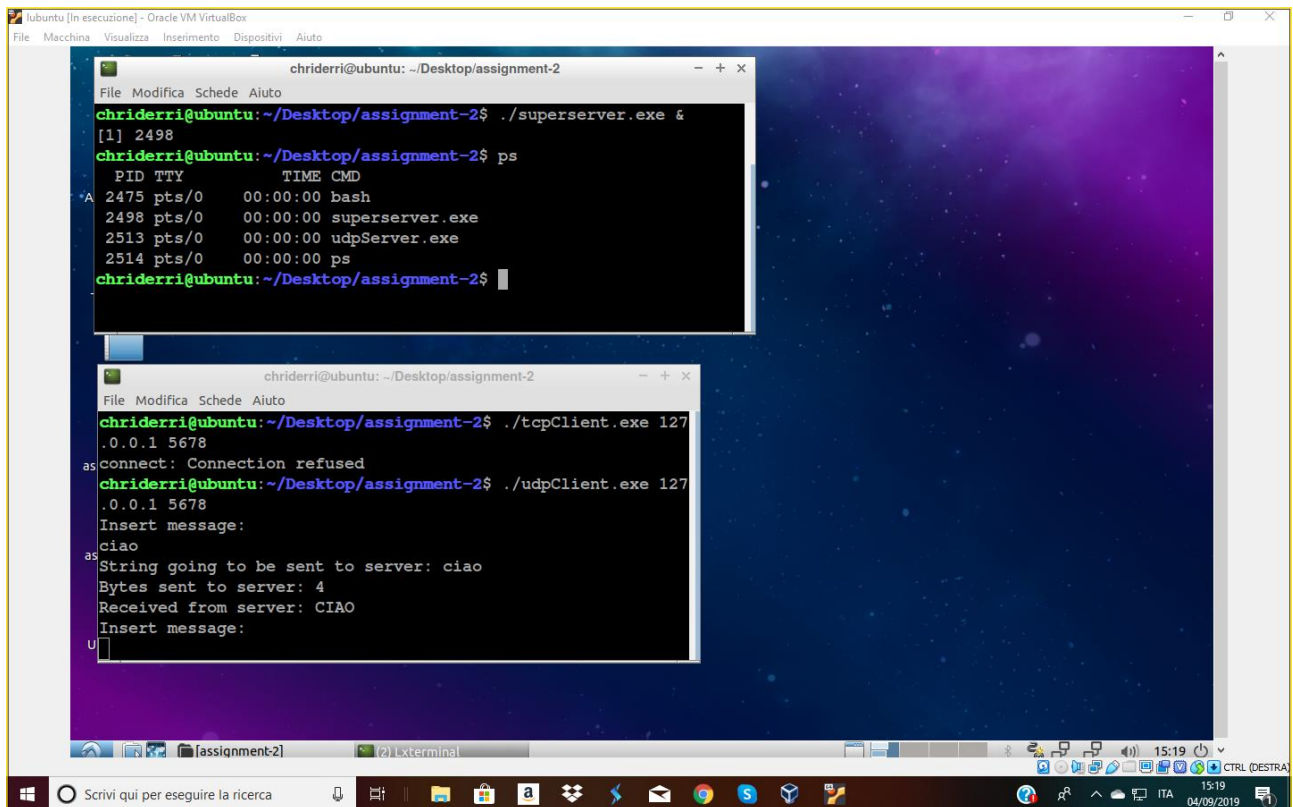
chriderri@ubuntu: ~/Desktop/assignment-2
chriderri@ubuntu:~/Desktop/assignment-2$ ./udpClient.exe 127
.0.0.1 5678
Insert message:
ok
String going to be sent to server: ok
Bytes sent to server: 2
Received from server: OK
Insert message:
```

Tale situazione è stata parzialmente “superata” ponendo una `sleep()` che permettesse il ritardo dell’esecuzione del processo padre in modo che lo scheduler desse la precedenza al processo figlio generato evitando così un’eccessiva proliferazione di processi. Tuttavia, permane il fatto che ogni invio di byte da parte del client verso il server comporta la creazione di almeno un nuovo processo. Questa situazione è dovuta alle caratteristiche intrinseche del protocollo UDP il quale regola il trasporto di dati senza controlli di connessione. La funzione `select()` in questo caso, rilascia il controllo del flusso di esecuzione del superserver ogni volta che i client inviano dei dati, poiché la socket associata al servizio risulta sempre pronta alla ricezione dei dati in quanto non viene vista come impegnata in una connessione.

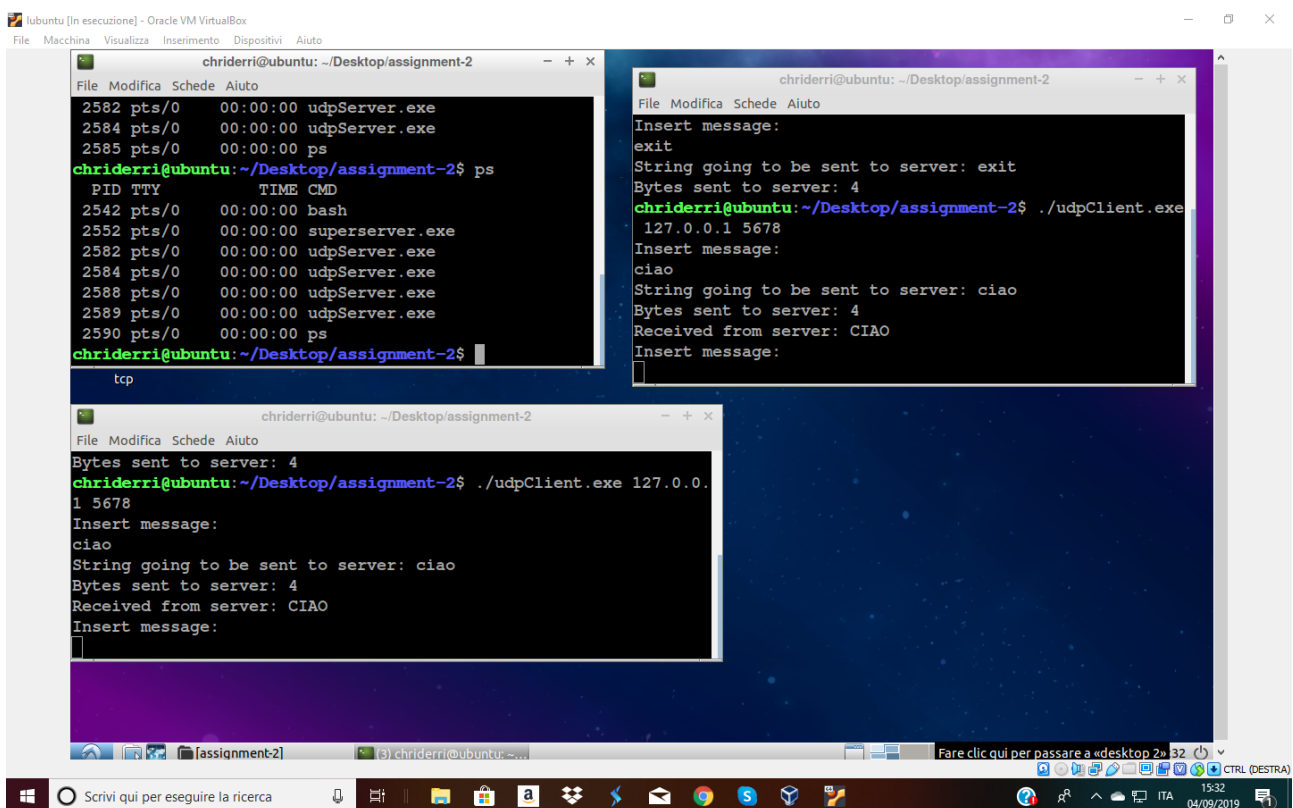


```
...
} else {
    if (dup(services[i].socket_fd) != STDIN_FILENO || dup(services[i].socket_fd) != STDOUT_FILENO || dup
        printf("Error duplicating socket for stdin/stdout/stderr");
        exit(EXIT_FAILURE);
    }
    check = execl(services[i].full_path_name, services[i].service_name, NULL, env); /* Performs the servic
    if (check < 0) {
        printf("ERRORE\n");
    }
    else { /* sleep */
        sleep(0.7);
        if (strcmp(services[i].protocol, "tcp") == 0) {
            if (strcmp(services[i].service_mode, "wait") == 0) {
                services[i].pid = pid;
                FD_CLR(services[i].socket_fd, &readset);
                remove_descriptor(services[i].socket_fd);
                int k;
                for (k=0; k<4; k++) {
                    printf("Socket: %d\n", descriptors[k]);
                }
            }
        }
    }
}
free(descriptors);
}
```

Nella figura si mostra la collocazione della funzione `sleep()` citata.



A seguito dell'introduzione della funzione `sleep()` il numero dei processi figli è notevolmente diminuito, ma ad ogni invio di byte viene generato comunque un nuovo processo figlio.



La foto sopra mostra il corretto comportamento del servizio in modalità `no-wait` con i due client serviti contemporaneamente.