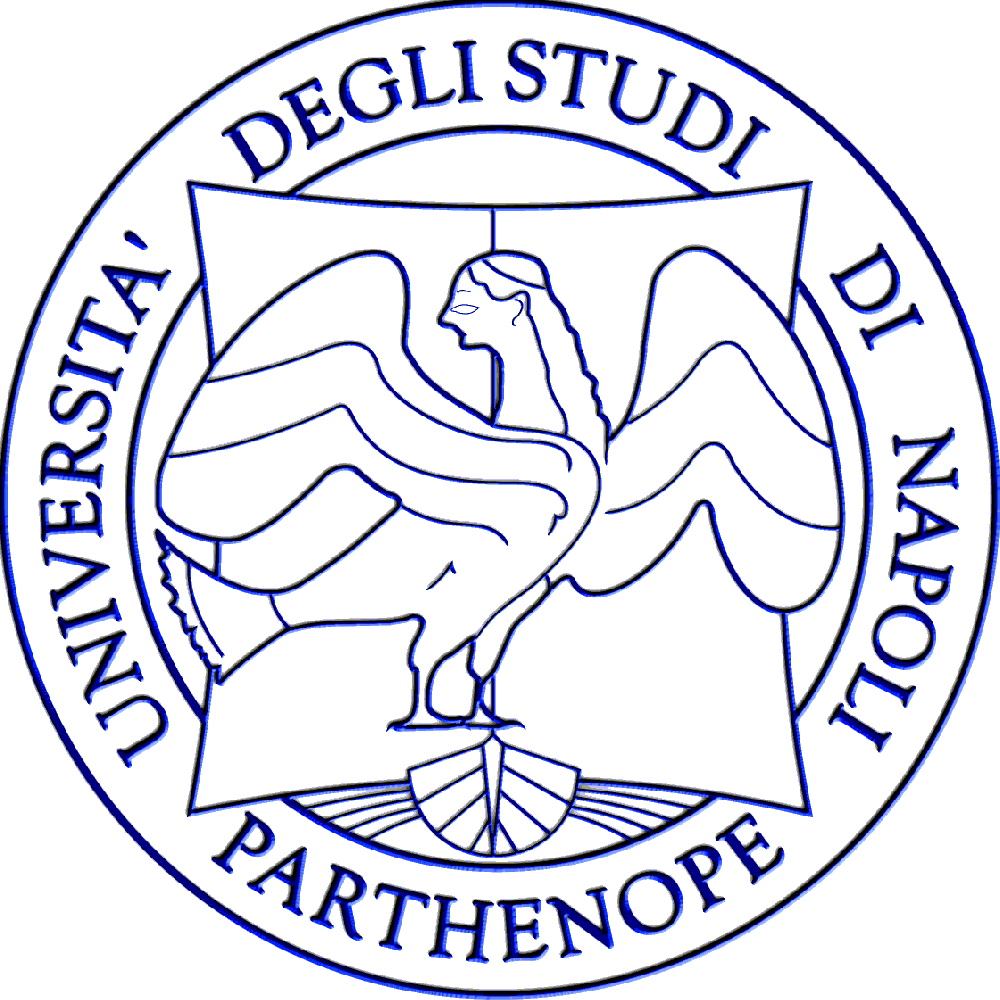
Progetto Reti di calcolatori

FOOD DELIVERY



Romeo Velvi 0124001993

Pasquale Casoria 0124001993

Crescenzo Bencivenga 0124001993

Sommario

[Descrizione del progetto 3](#_Toc76767292)

[Tecnologie utilizzate 3](#_Toc76767293)

[Descrizione e schemi dell’architettura 3](#_Toc76767294)

[Descrizione e schemi del protocollo applicazione 4](#_Toc76767295)

[Cliente 6](#_Toc76767296)

[Server 6](#_Toc76767297)

[Ristorante 7](#_Toc76767298)

[Rider 8](#_Toc76767299)

[Descrizione strutture dati 8](#_Toc76767300)

[Lista doppiamente lincata 8](#_Toc76767301)

[Strutture 8](#_Toc76767302)

[ Prodotto 8](#_Toc76767303)

[ Ordine 9](#_Toc76767304)

[ Ristorante 9](#_Toc76767305)

[ Rider 9](#_Toc76767306)

[ Operazione 10](#_Toc76767307)

[ Info\_ordine 10](#_Toc76767308)

[Dettagli implementativi 11](#_Toc76767309)

[Manuale utente 12](#_Toc76767310)

[Guida delle sorgenti 12](#_Toc76767311)

[Istruzioni per la compilazione 13](#_Toc76767312)

[Istruzioni per l’esecuzione 13](#_Toc76767313)

[Cliente 13](#_Toc76767314)

[Rider 15](#_Toc76767315)

[Server e Ristorante 16](#_Toc76767316)

## Descrizione del progetto

Progettare ed implementare un servizio di food delivery secondo le seguenti specifiche. Il client si collega al server da cui riceve la lista dei ristoranti. L’utente, usando l’interfaccia del client, sceglie il ristorante. Il server richiede al ristorante scelto una lista di cibi e bevande ordinabili e la invia al client da cui l’utente effettua l’ordine. Una volta effettuato l’ordine, il server lo inoltra al ristorante che verifica la disponibilità di rider inviando la richiesta ai rider connessi e selezionando il primo da cui riceve la conferma. Ricevuta la conferma, il server invia l’id del rider al client e l’id del client al rider. Una volta effettuata la consegna il rider invia una notifica al ristorante che a sua volta la inoltra al server.

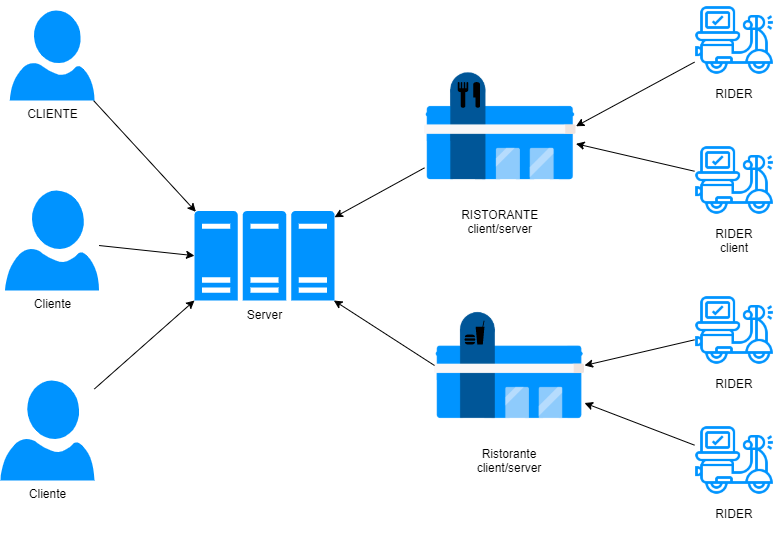
### Tecnologie utilizzate

Si è utilizzato il linguaggio C, implementando le socket per la comunicazione tra processi. Inoltre, si è scelto, come gestore di IO, per il server uno schema IO/Multiplexing (implementato tramite la select).

## Descrizione e schemi dell’architettura

L’architettura proposta è composta da 4 entità:

* Server, è il server fondamentale dell’applicazione, si tratta dell’entità che gestisce le varie comunicazioni tra le altre entità ad esso collegate, pertanto deve essere sempre attivo.
* Cliente, è un'entità client che può connettersi al Server per controllare la lista di ristoranti disponibili ed effettuare ordini.
* Ristorante, si tratta di un’entità che si comporta sia da client che da server, ha il compito di gestire gli ordini degli utenti e le connessioni dei rider.
* Rider, è un tipo client che si connette al ristorante e richiede ad esso gli ordini da consegnare.



*Le frecce nella foto indicano le varie connessioni con le entità, ad esempio:*

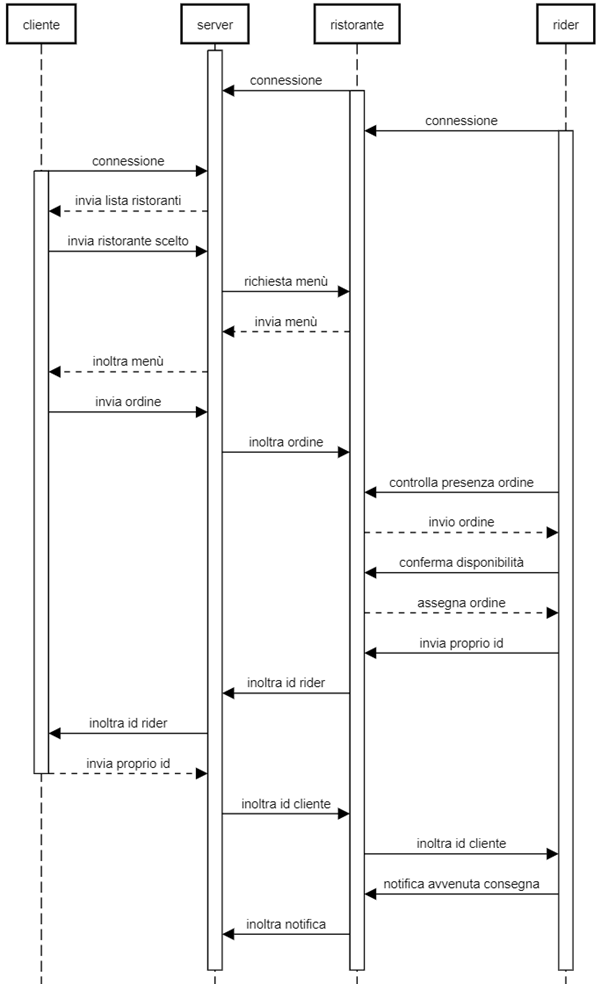
*indica che il client si connette al server*

*indica che il rider si connette al ristorante*

*indica che il ristorante si connette al server*

## Descrizione e schemi del protocollo applicazione

Di seguito illustrati il diagramma delle sequenze in modo generale e i vari schemi di protocollo adottati per il progetto.



.

Cliente

Il client si connette con il server.

Il cliente prima dell'invio di ogni messaggio al server gli invia un numero che gli permette di avviare la giusta procedura/routine per gestire la richiesta.

Il cliente si identifica come tale inviando 2 al server e si mette in attesa per la ricezione dei ristoranti attivi connessi ad esso.

Prima di inviare la scelta del ristorante al server gli manda 2, e attende l’invio del menu.

Ricevuto il menu il cliente provvederà ad effettuare l'ordine e mandare 5 al server, e attende la ricezione dell'id del rider da esso, ricevuto l'id del rider il cliente invierà il proprio id al server

Completati questi passaggi il cliente può scegliere se effettuare un altro ordine o chiudere la connessione

### Server

Il server è attivo e rimane in attesa di connessioni sia da parte di clienti che ristoranti.

Il server utilizza l’io-multiplexing per la gestione dell’I/O.

Il server, ogni volta che si sblocca dalla select, controlla se c'è qualche cliente o ristorante che si vuole connettere, e se possibile, accettarlo.

Il server grazie a delle strutture dati apposite riesce a gestire le varie fasi del servizio attraverso degli identificativi di fase che gli permettono di gestire in modo efficace lo scambio di dati tra le diverse parti in collegamento.

Identificatori:

1. Il ristorante si connette, il server deve ricevere il nome del ristorante ed aggiungerlo alla lista dei ristoranti attivi.
2. Il cliente si connette, il server provvede ad inviare al cliente la lista dei ristoranti attivi.
3. Il cliente ha scelto il ristorante, il server riceve l’fd del ristorante scelto, viene creata una struttura operazione che tiene conto dello stato di comunicazione tra il cliente e il ristorante e procede ad inviare 1 al ristorante per richiedere l'invio del menu.
4. Il ristorante invia il menu, il server inoltra il menu al cliente.
5. Il cliente invia l'ordine effettuato al server, il server invia 2 al ristorante per poi inoltrargli l'ordine. Inoltre, il server invia anche l'id dell’operazione.
6. Il ristorante elabora l'ordine, il server riceve l'id del rider cha ha preso in carico l'ordine e l'id dell’operazione per reperire l’fd del rider. Successivamente inoltra l'id del rider al cliente, riceve l'id del cliente e lo inoltra al ristorante.
7. Il rider ha consegnato l'ordine, di conseguenza il ristorante invia l'id dell'operazione al server che procede ad eliminare il nodo relativo all’operazione, in quanto la richiesta del cliente è stata soddisfatta.
8. Il cliente si disconnette e il server "libera" l'fd del cliente su cui era in ascolto
9. Il ristorante si disconnette e il server "libera" l'fd del ristorante su cui era in ascolto e lo elimina dalla lista dei ristoranti.

### Ristorante

Il ristorante si connette con il server, e gli invia un messaggio contenente 1 per identificarsi come ristorante e successivamente invia il suo nome.

Similmente al server principale, il ristorante essendo sia cliente che server sfrutta l'io-multiplexing per la gestione dell'I/O con i rider e con il server principale.

Il ristorante ogni volta che si sblocca dalla select, controlla se c'è qualche rider che si vuole connettere ad esso.

Identificatori:

1. Il server richiede il menu, invia al server 4 e successivamente inoltra i vari prodotti.
2. Il server inviare l'ordine fatto da un cliente. Una volta ricevuto l'ordine lo aggiunge in coda alla lista degli ordini.
3. Il rider vuole sapere se ci sono ordini disponibili per la consegna, il ristorante gli inoltra il numero di ordini disponibili.
4. Il rider vuole effettuare una consegna, per cui, il ristorante controlla ogni ordine in lista, se c'è qualche ordine non ancora assegnato a nessun rider, il ristorante invia 1 e riceve l'id del rider e invia l'id dell'operazione presa in carico (che servirà successivamente al ristorante per controllare i dettagli dell'ordine consegnato). Il ristorante invia 6 al server e l'id del rider che ha effettuato la consegna e il relativo id dell'operazione assegnata (servirà al server per risalire al cliente dell'ordine consegnato). Successivamente, il ristorante, in risposta del server, riceve l'id del cliente che inoltra al rider. Se il ristorante non riesce a trovare nessun ordine disponibile per farlo consegnare al rider, allora invia 2.
5. Il rider ha consegnato l'ordine, il rider invia un messaggio contenente l'id dell’operazione (che servirà al ristorante per reperire l'ordine che ha inviato). Successivamente si invia 7 al server per poi inviargli l'id dell'operazione

### Rider

Il rider si connette al ristorante, invia 3 al ristorante e si aspetta la recezione degli ordini disponibili per la consegna. Se ci sono ordini disponibili, esso può decidere se accettarli, o meno.

Se decide di accettare un ordine, invia 4 al ristorante. Di risposta il ristorante gli invia un numero.

Se il numero è 1 allora il rider può consegnare l'ordine e invia il proprio id al ristorante. Il ristorante gli invia l'id dell'operazione (a cui è associato l'ordine) e l'id del cliente.

Una volta consegnato, il rider invia 5 al ristorante (ordine consegnato) e gli invia l'id operazione a cui era associato l'ordine.

## Descrizione strutture dati

Per quanto riguarda le strutture dati utilizzate, sono state dichiarate e implementate nel file e .

### Lista doppiamente lincata

Si è fortemente utilizzato una lista doppiamente lincata a contenuto generico per memorizzare e organizzare velocemente e al meglio i dati.

**typedef** struct lnode

**{**

struct lnode**\*** prev**;**

/\* Pointer nodo precedente \*/

struct lnode**\*** next**;**

/\* Pointer nodo successivo \*/

void**\*** data**;**

/\* User data \*/

**}** node**;**

**typedef** struct llist

**{**

struct lnode**\*** head**;**

/\* puntatore a testa \*/

struct lnode**\*** tail**;**

/\* puntatore a coda \*/

unsigned int size**;**

/\* Size della linked list \*/

**}** list**;**

### Strutture

Tra i vari dati di tipo strutturato, creati appositamente per il progetto ci sono:

#### Prodotto

**typedef** struct Prodotto**{**

char items**[**max\_name**];**

float prezzo**;**

**}**Prodotto**;**

Questa struttura è utilizzata come oggetto dati da memorizzare nella lista del menu del ristorante. Questa variabile verrà utilizzata di conseguenza per il passaggio del menu dal ristorante al client.

Essa è composta da: , che rappresenta il nome dell’oggetto nel menu ed il che definisce il costo del medesimo.

#### Ordine

typedef struct Ordine{

char items[max\_name];

int qt;

} Ordine;

Questa struttura è utilizzata come oggetto dati da memorizzare nella lista ordine del cliente. Ovviamente, verrà anche utilizzata per il passaggio del menu dal client al ristorante.

Essa è composta da: , che rappresenta il nome dell’oggetto scelto dal cliente e da che ne rappresenta la quantità scelta.

#### Ristorante

typedef struct ristorante{

char nome\_rist[max\_name];

int fd\_rist;

} Ristorante;

Questa struttura è utilizzata come oggetto dati da memorizzare nella lista dei ristoranti attivi mantenuta dal server.

Essa è composta da: che è il nome del ristorante attivo sul server e dal che rappresenta l’indice corrispondente al file descriptor utilizzato per l’interscambio di dati/messaggi da e verso server e ristorante.

#### Rider

typedef struct rider{

char id\_rider[id\_size];

int fd\_rider;

} Rider;

Questa struttura, similmente alla precedente, è utilizzata come oggetto dati da memorizzare nella lista dei rider attivi connessi sul ristorante.

Essa è composta da: che è l’identificativo del rider attivo connesso al ristorante e dal che rappresenta l’indice corrispondente al file descriptor utilizzato per l’interscambio di dati/messaggi da e verso ristorante e rider.

#### Operazione

typedef struct operazione {

char id\_operazione[id\_size];

int fd\_client;

int fd\_ristorante;

int stato\_operazione;

} Operazione;

Questa struttura è utilizzata come oggetto dati da memorizzare nella lista delle operazioni mantenute sul server. Questa è una particolare e fondamentale lista che permette una gestione efficace sulle varie associazioni e operazioni avente come oggetti cliente e ristorante tenendo traccia del relativo stato di comunicazione. La struttura Operazione si crea quando le coppie cliente/ristorante sono stati selezionati, ossia, quando il cliente sceglie il ristorante su cui fare l’ordine.

La struttura Operazione è composta da: che è un identificativo univoco dell’associazione client/ristorante. Inoltre, contiene anche l’e l’ che rappresentano rispettivamente il file descriptor del client e del ristorante. Il campo serve a identificare, in base al valore che assume:

1: Il client specificato da richiede il menu, e di conseguenza, quando il ristorante glielo invia, il server lo inoltrerà al che ha come oggetto il ristorante in questione e l’ pari a 1.

2: Il cliente ha effettuato l’ordine, per cui, il server, invia al ristorante specificato nel campo una notifica di recezione ordine.

3: Il ristorante ha elaborato l’ordine, quindi, come da protocollo, il server deve inviare l’id del rider, incaricato della consegna dell’ordine dal ristorante al cliente specificato da .

4: Il rider sta consegnando l’ordine, di conseguenza, una volta ricevuto dal cliente, il server a breve cancellerà questo nodo in quanto il servizio e le operazioni da compiere sono state tutte portate a termine.

#### Info\_ordine

typedef struct Info\_ordine{

list\* ordini;

char id\_operazione[id\_size];

int fd\_rider;

int stato\_ordine;

} Info\_ordine;

Questa struttura è utilizzata come oggetto dati da memorizzare nella lista degli ordini mantenuta dal ristorante. Questa struttura ha uno scopo abbastanza importante, ossia, essa si occupa di mantenere gli ordini fatti dal client e di poter gestire eventuali ordini da consegnare ai rider.

Questa struttura è formata da: , questo campo (inviato dal server) è settato al momento della consegna dell’ordine da parte del server, esso ha due scopi principali: il primo, è quello di avere un’ulteriore identificazione dagli altri ordini, ed il secondo è per l’utilità e la praticità quando vi è lo scambio di informazioni tra il rider, ristorante e server in quanto questa variabile sarà uguale tra le varie entità in gioco. Il campo identifica il rider che ha effettuato/effettuerà la consegna dello stesso ordine. Ed infine lo , questa identifica, lo stato dell’ordine, in particolare si distinguono:

1: L’ordine è appena stato ricevuto, deve essere ancora elaborato, pertanto, non c’è ancora nessun rider per la consegna.

2: L’ordine è stato elaborato ed è stato assegnato un rider per la consegna.

3: L’ordine è stato consegnato dal rider.

## Dettagli implementativi

Le entità, siano esse client o server, fanno uso di socket sulle quali effettuano operazioni bloccanti, in particolare si utilizzano due funzioni “wrapper”: e , le quali garantisco l’intera lettura e scrittura delle informazioni, tutelando così le operazioni in caso di occorrenze di eventi che determinano l’interruzione di esse.

Il ristorante e il cliente hanno un gestore del segnale SIGINT che, prima di uscire, invia un messaggio al server, che a sua volta, libera il descrittore ad essi associato in modo da poter accettare ulteriori connessioni.

Lato server si è scelto l’utilizzo dell’io-multiplexing come gestore sincrono di I/O ed è di tipo iterativo.

Funzioni utilizzate (per l’implementazione e la dichiarazione e ):

// stampa e scelta Ristorante nella list { usata dal CLIENTE}

int show\_choose\_resturant**(**list**\*** llist**);** // ritorna direttamente l'fd

// stampa e scelta Prodotti nella list (menu) {usata dal CLIENTE}

list**\*** show\_choose\_product**(**list**\*** llist**);** // ritorna la lista contenente l’ordine

// generare un stringa random

char **\***rand\_string**(**char **\***str**,** size\_t size**);**

// Function che cerca nella lista dei ristoranti dato il nome {usata dal SERVER}

node**\*** find\_resturant\_by\_name **(**list**\***l**,**char**\***nome\_rist**);** // ritorna il nodo

// Function che cerca nella lista dei ristoranti dato l’fd {usata dal SERVER}

node**\*** find\_resturant\_by\_fd **(**list**\***l**,** int fd**);** // ritorna il nodo

// cerca l’Operazione dato l'fd del client {usata dal SERVER}

Operazione**\*** find\_resturant\_operation**(**list **\***l**,**int fd**,**int st**);** // ritorna fd ristorante

// cerca l’Operazione dato l'fd del ristorante {usata dal SERVER}

Operazione**\*** find\_client\_operation**(**list **\***l**,**int fd**,**int st**);** // ritorna fd client

// cerca l’Operazione dato l'id della Operazione {usata dal SERVER}

Operazione**\*** find\_id\_operation**(**list **\***l**,** char**\***id**);**

// ritorna il nodo nella lista che ha id pari a quello dato {usata dal SREVER}

node**\*** find\_id\_operation\_node**(**list**\***l**,** char**\***id**);**

// cerca tra gli info ordini il nodo con fd pari a quello dato come argomento.

Info\_ordine**\*** find\_Info\_ordine**(**list**\***l**,** char **\***id**);**

//\* FULL READ \*//

void FullRead**(**int fd**,** void **\*** buf**,** size\_t count**);** //funzione wrapper fullRead

//\*FULL WRITE\*//

void FullWrite**(**int fd**,** const void **\*** buf**,** size\_t count**);** //funzione wrapper fullWrite

## Manuale utente

### Guida delle sorgenti

Di seguito elencati i vari file scritti in contenenti i vari codici:

Sorgente dell’applicativo del cliente.

Sorgente dell’applicativo del server.

Sorgente dell’applicativo del ristorante.

Sorgente dell’applicativo del rider.

Header comune contenente tutte le dichiarazioni delle strutture e delle funzioni necessarie per l’esecuzione del sistema.

Contiene l’implementazione delle funzioni specificati nell’header

File “shortucut” per la compilazione dei vari codici.

### Istruzioni per la compilazione

Nelle varia cartelle è presente un Makefile, per cui, già sono state inserite le linee di codice per far partire i programmi. Basta inserire nel terminale, una volta dentro la cartella dell’entità, è si avvierà automaticamente la procedura per la compilazione del programma. Per eseguirlo, poichè l’eseguibile ha lo stesso nome, basta fare, per ogni terminale aperto dedicato ad ogni entità .

Per chiarezza, e pulizia del terminale, si consiglia di eseguire tutto in un’unica istruzione:

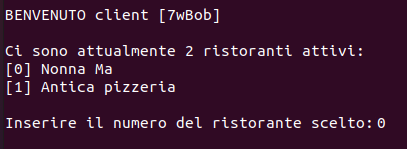
### Istruzioni per l’esecuzione

Per eseguire il codice, bisogna, far partire dapprima il Server, il Ristorante, il Rider poi il Cliente.

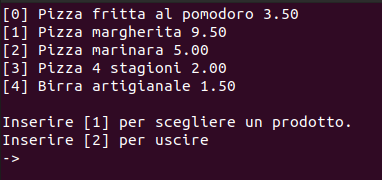
Attenzione è consigliata questa sequenza di esecuzioni per un comportamento lineare, altri ordini di esecuzione sono accettati, con la condizione che parta prima il Server.

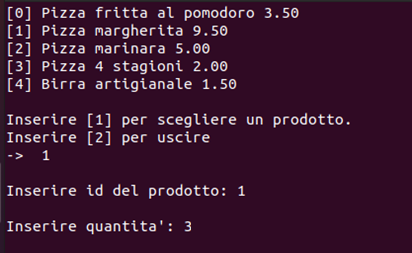
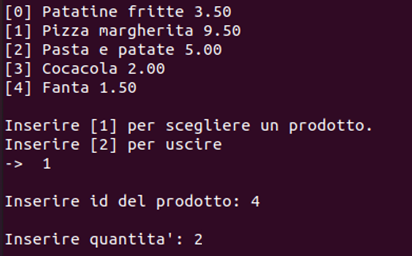
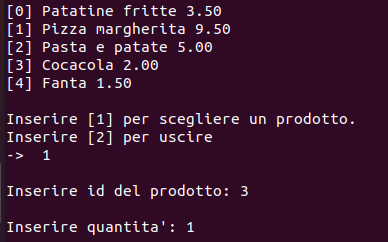
#### Cliente

Appena si fa partire il cliente, ci viene mostrato l’id personale e una lista dei ristoranti attivi con la possibilità di scegliere quale selezionare (nell’esempio ci sono 2 ristoranti).

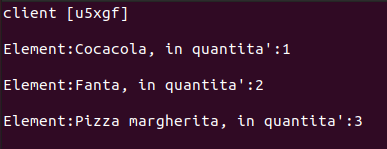


Subito dopo la selezione del ristorante, apparirà, come da protocollo, il menu del ristorante scelto con la possibilità di eseguire due principali azioni. La prima sceglie un prodotto, la seconda, esce dalla selezione dei prodotti e assembla l’ordine. (Nell’esempio mostrato si ordinano tre alimenti, di differenti quantità)

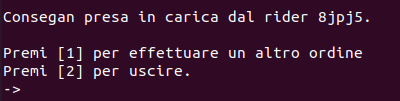


Subito dopo la scelta dell’ordine, il programma visualizzerà il resoconto degli alimenti scelti.

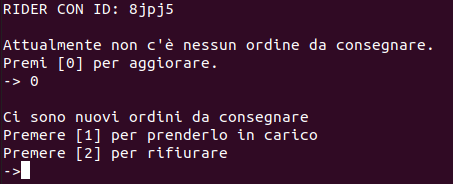


Quando l’ordine ha raggiunto il ristorante ed un client era disponibile per la consegna, una volta avvenuta l’assegnazione il server invierà il messaggio contenente l’id del rider. Una volta fatto ciò, è discrezione dell’utente scegliere se abbandonare l’app o rimanere e fare un altro ordine

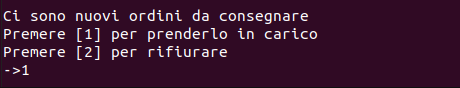


#### Rider

Il rider che dapprima si connette con il Ristorante, giustamente non avrà nessun ordine da consegnare in quanto non ci sono ancora, o, a momento della connessione non è detto che ci siano di disponibili.

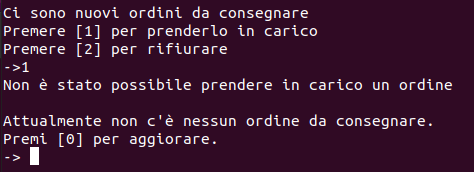


Il rider, può aggiornare la pagine per controllare se ci siano nuovi ordini.

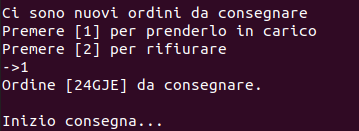


Quando si sceglie di prenderlo in carico un ordine, non è detto che si abbia l’assoluta certezza che lo si ottenga.

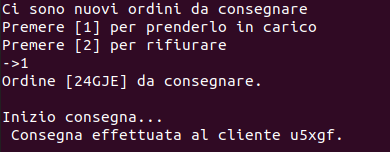
Per cui, nel caso in cui il ristorante lo abbia già assegnato, allora visualizzerà un messaggio con la possibilità di aggiornare per trovare nuovi ordini disponibili per la consegna.



Altrimenti, se si è preso l’ordine in carico visualizzerà questo:



Attenderà 5 secondi, per simulare l’effettiva presa e consegna dell’ordine e visualizzerà il messaggio di ordine consegnato



#### Server e Ristorante

Per quanto riguarda il server e il ristorante, non prevedendo interazioni dirette (a linea di comando) con l’utente, si è implementato un sistema di log, che permette di monitorare le varie operazioni avvenute ed effettuate.