Progetto Reti di calcolatori

FOOD DELIVERY

Romeo Velvi 0124001993
Pasquale Casoria 0124002016
Crescenzo Bencivenga 0124001916

Sommario

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.

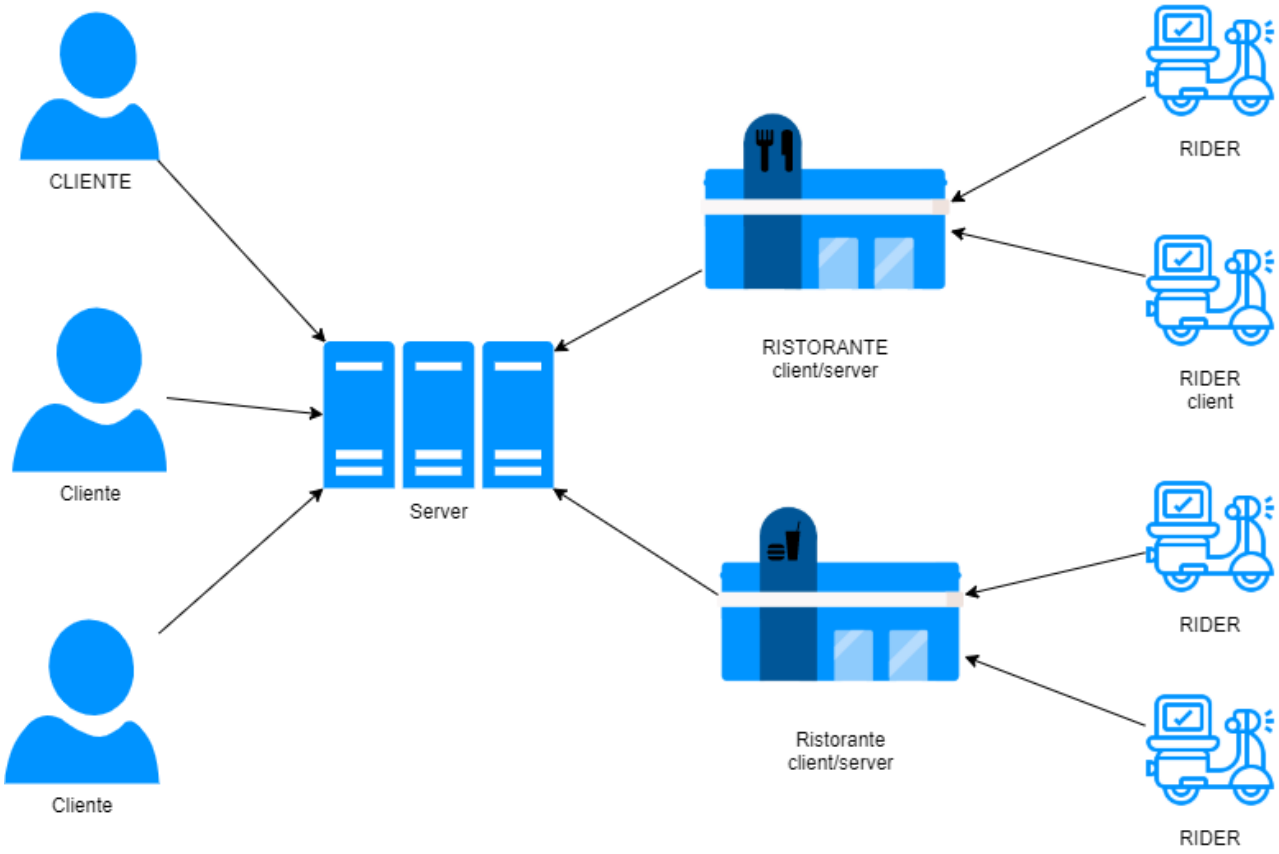
Tecnologie utilizzate

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

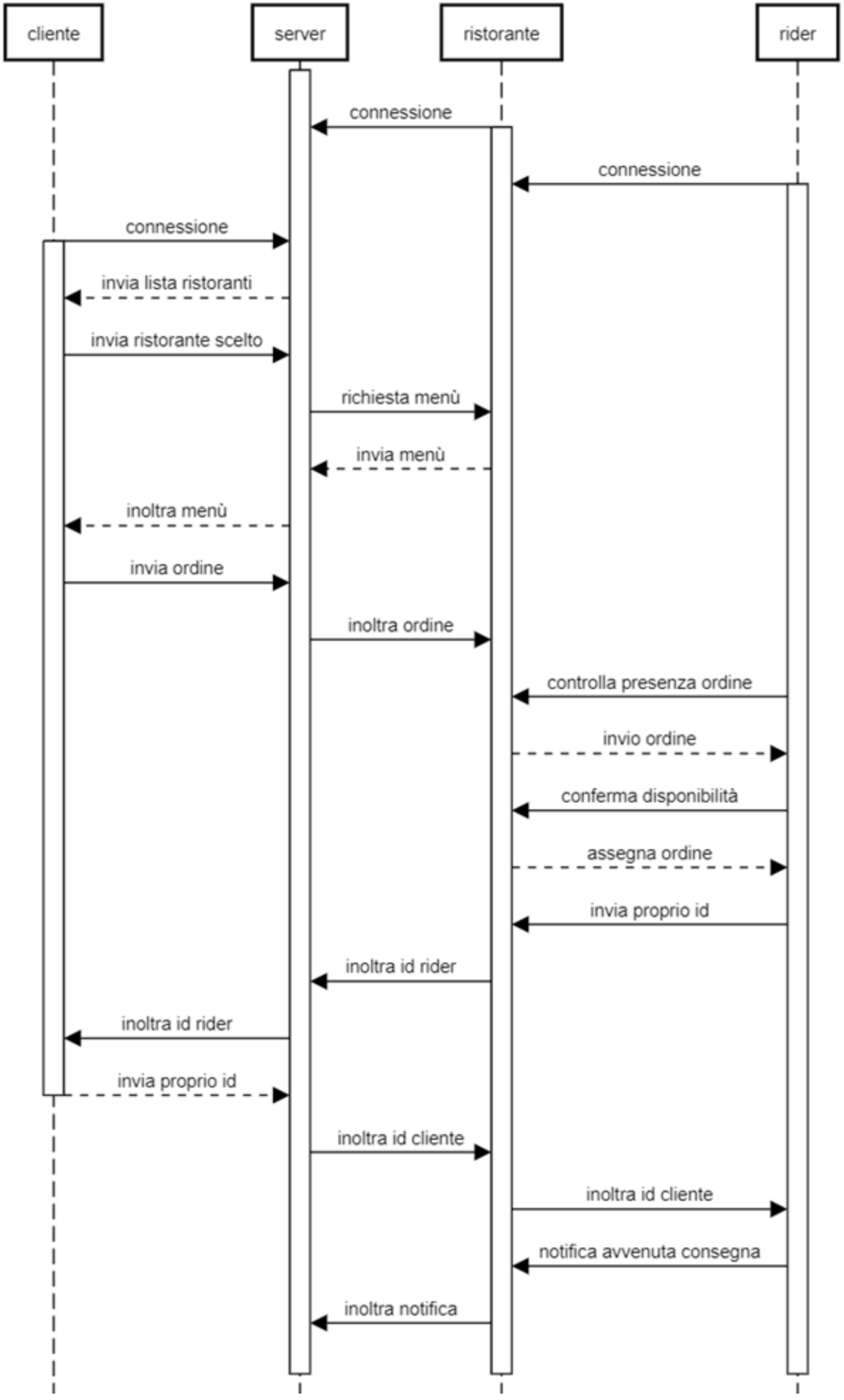
L'architettura proposta è composta da 4 entità:

- 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.



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
server
ristorante
rider
connessione
connessione
connessione
invia lista ristoranti
invia ristorante scelto
richiesta menù
invia menù
inoltra menù
invia ordine
inoltra ordine
controlla presenza ordine
invio ordine
conferma disponibilità
assegna ordine
invia proprio id
inoltra id rider
inoltra id rider
invia proprio id
inoltra id cliente
inoltra id cliente
notifica avvenuta consegna
inoltra notifica

Cliente

Il client si connette con il server, prima dell'invio di ogni messaggio al server 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 invia 3, 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 e 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 identificatori 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 il 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" il fd del cliente sul quale era in ascolto
9: Il ristorante si disconnette e il server "libera" il fd del ristorante su cui era in ascolto e lo elimina dalla lista dei ristoranti.

Ristorante

Il ristorante si connette con il server, e invia un messaggio contenente 1 per identificarsi come ristorante e successivamente invia il suo nome.
Similmente al server principale, il ristorante essendo sia client 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 invia al rider 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 un 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, 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 e lo inoltra al rider. Se il ristorante non ha ordini da consegnare, 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). Il ristorante invia 7 al server per poi inviargli l'id dell'operazione

Rider

Il rider si connette al ristorante, invia 3 e 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 invia un numero.
Se il numero è 1 allora il rider può consegnare l'ordine e invia il proprio id al ristorante. Il ristorante 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 nei file $llist.h$ e $llist.c$.

Lista doppiamente linkata

Si è ampiamente utilizzato una lista doppiamente linkata a contenuto generico per memorizzare e organizzare 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 data da memorizzare nella lista del menu del ristorante. Questa variabile verrà utilizzata di conseguenza per il passaggio del menu dal ristorante al server.
Essa è composta da: $items$, che contiene il nome dell'oggetto nel menu ed il $prezzo$.

- *Ordine*

```
typedef struct Ordine{
    char items[max_name];
    int qt;
} Ordine;
```

Questa struttura è utilizzata per memorizzare l'ordine del cliente, viene anche utilizzata per il passaggio dell'ordine dal cliente al server.
Essa è composta da: $items$, che contiene il nome dell'oggetto scelto dal cliente e da qt che ne contiene la quantità scelta.

- *Ristorante*

```
typedef struct ristorante{
    char nome_rist[max_name];
    int fd_rist;
} Ristorante;
```

Questa struttura è utilizzata dal server per memorizzare nel campo data della lista dei ristoranti contenente i ristoranti attivi.
Essa è composta da: $nome\_rist$ che è il nome del ristorante e da fd_rist che contiene 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 dal ristorante come campo data da memorizzare nella lista dei rider attivi connessi al ristorante.
Essa è composta da: id_rider che è l'identificativo del rider attivo connesso al ristorante e da fd_rider 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 dal server come campo data da memorizzare nella lista delle operazioni mantenute dal 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 da cui fare l'ordine.
La struttura Operazione è composta da: $id\_operazione$ che è un identificativo univoco dell'associazione client/ristorante. Inoltre, contiene anche fd_client e $fd\_ristorante$ che contengono rispettivamente il file descriptor del client e del ristorante. Il campo $stato\_operazione$ serve a identificare, in base al valore che assume:

1: Il client specificato da fd_client richiede il menu, e di conseguenza, quando il ristorante lo invia al server, il server lo inoltrerà al fd_client che ha come oggetto il ristorante in questione e l'$id\_operazione$ pari a 1.

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

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

4: Il rider sta consegnando l'ordine, di conseguenza, una volta ricevuto dal cliente, il server cancella 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 dal ristorante come campo data da memorizzare nella lista degli ordini. Questa struttura ha lo scopo di mantenere gli ordini fatti dal cliente e di poter gestire gli ordini da consegnare dai rider.
Questa struttura è composta da: $id\_operazione$, questo campo è settato dal server nel momento in cui il cliente sceglie il ristorante e inviato nel momento dell'inoltro dell'ordine da parte del server al ristorante, esso ha due scopi principali: il primo, è quello di poter identificare univocamente gli ordini, ed il secondo è per lo scambio di informazioni tra rider, ristorante e server in quanto questa variabile sarà uguale tra le varie entità in comunicazione. Il campo fd_rider identifica il rider che ha effettuato la consegna dell'ordine. Ed infine il campo $stato\_ordine$ identifica lo stato dell'ordine, in particolare si distinguono vari stati:

1: L'ordine è appena stato ricevuto, deve essere ancora elaborato, pertanto, non è stato ancora assegnato un 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": $FullRead()$ e $FullWrite()$, 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 (dichiarate e implementate in $llist.h$ e $llist.c$):

```
// stampa e scelta Ristorante nella list {usata dal CLIENTE}
int show_choose_resturant(list* llist);  // ritorna direttamente il fd del ristorante

// stampa e scelta Prodotti della 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 il nodo ristorante 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 il nodo ristorante nella lista dei ristoranti dato il fd {usata dal SERVER}
node* find_resturant_by_fd (list*l, int fd); // ritorna il nodo

// cerca l'Operazione dato il fd del client {usata dal SERVER}
Operazione* find_resturant_operation(list *l,int fd,int st); // ritorna fd ristorante

// cerca l'Operazione dato il 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 dell'Operazione {usata dal SERVER}
Operazione* find_id_operation(list *l, char*id);

// ritorna il nodo nella lista che ha id uguale a quello dato {usata dal SREVER}
node* find_id_operation_node(list*l, char*id);

// cerca tra gli info ordini il nodo con fd uguale 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 C contenenti i vari codici:

- $Client.c$
  Sorgente dell'applicativo del cliente.
- $Server.c$
  Sorgente dell'applicativo del server.
- $Resturant.c$
  Sorgente dell'applicativo del ristorante.
- $Rider.c$
  Sorgente dell'applicativo del rider.
- $llist.h$

Header comune contenente tutte le dichiarazioni delle strutture e delle funzioni necessarie per l'esecuzione dell'applicativo.

- $llist.c$
  Contiene l'implementazione delle funzioni specificati nell'header $llist.h$
- $Makefile$
  File per la compilazione dei vari codici.

Istruzioni per la compilazione

Nelle varie cartelle è presente un Makefile, per cui, già sono stati inseriti i comandi per la compilazione dei programmi. Basta eseguire il comando make nel terminale una volta dentro la cartella dell'entità. Per eseguirlo, poiché gli eseguibili hanno lo stesso nome, basta fare, per ogni terminale aperto dedicato ad ogni entità $./exe$.
Per chiarezza, e pulizia del terminale, si consiglia di eseguire tutto in un'unica istruzione:

$$clear;\ make;\ \ ./exe$$

Istruzioni per l'esecuzione

Per eseguire il codice, bisogna, far partire prima il Server, il Ristorante, il Rider poi il Cliente.
Questa sequenza di esecuzione è consigliata per un comportamento lineare, altri ordini di esecuzione sono accettati, con la condizione che parta prima il Server.

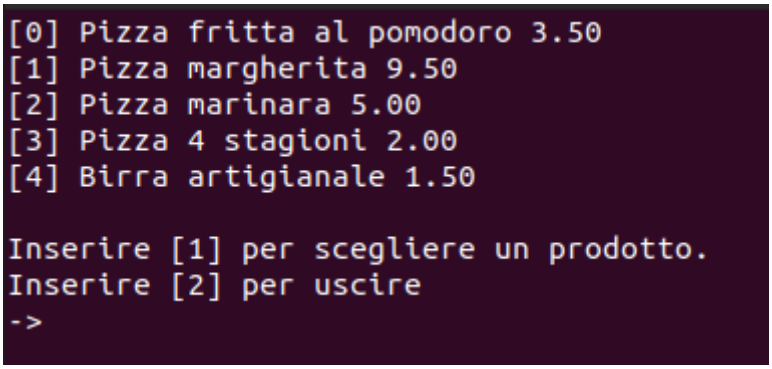
Cliente

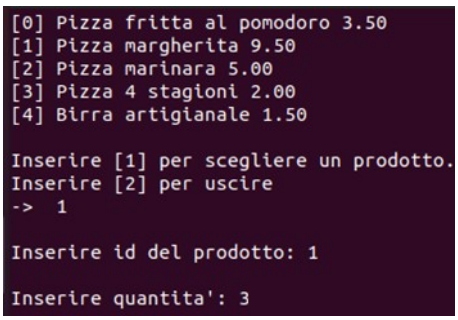
Appena si fa partire il cliente, viene mostrato l'id personale e una lista dei ristoranti attivi con la possibilità di scegliere quale selezionare (nell'esempio ci sono 2 ristoranti).

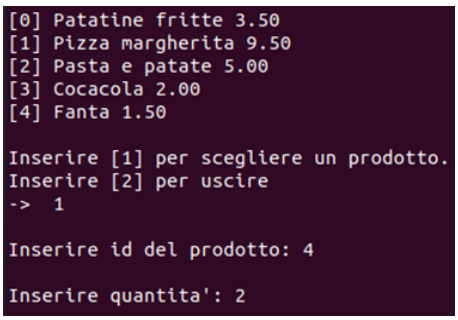
```
BENVENUTO client [7wBob]

Ci sono attualmente 2 ristoranti attivi:
[0] Nonna Ma
[1] Antica pizzeria

Inserire il numero del ristorante scelto: 0
```

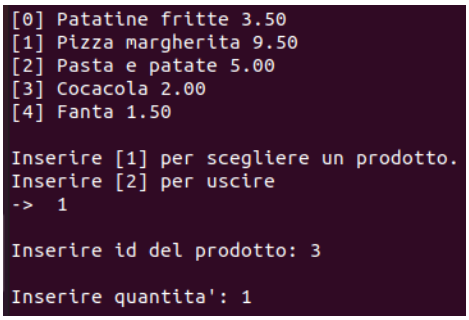
Subito dopo la selezione del ristorante, apparirà, come da protocollo, il menu del ristorante scelto con la possibilità di eseguire due principali azioni. La prima

sceglie un prodotto, la seconda, uscire dalla selezione dei prodotti e assemblare l'ordine. (Nell'esempio mostrato si ordinano tre alimenti, in differenti quantità)

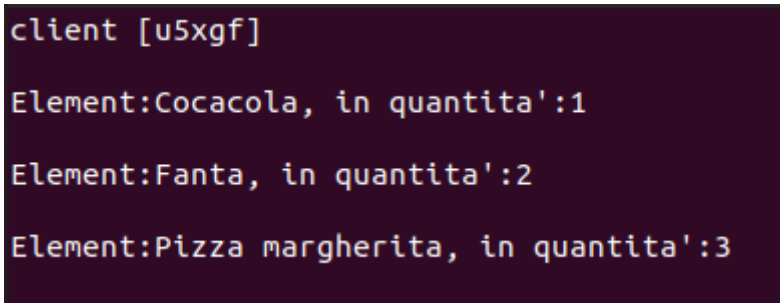




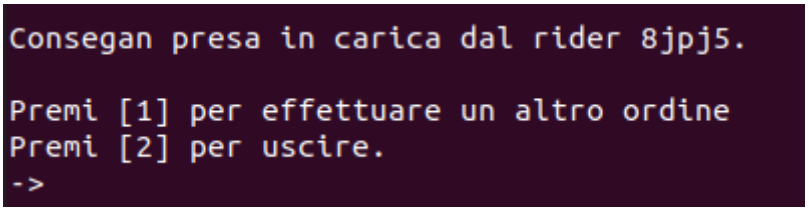




Subito dopo la scelta dell'ordine, il programma mostrerà il resoconto dei prodotti scelti.



Quando l'ordine ha raggiunto il ristorante ed un rider è disponibile per la consegna, una volta avvenuta l'assegnazione del rider all'ordine, il server invia il messaggio contenente l'id del rider al cliente. Una volta fatto ciò, è discrezione dell'utente scegliere se abbandonare l'applicativo o rimanere e fare un altro ordine



Rider

Nel momento in cui il rider si connette al ristorante non è detto che ci siano ordini disponibili per la consegna.

```
RIDER CON ID: 8jpj5

Attualmente non c'è nessun ordine da consegnare.
Premi [0] per aggiorare.
-> 0
```

Il rider può premere 0 per controllare se ci siano nuovi ordini.

```
Ci sono nuovi ordini da consegnare
Premere [1] per prenderlo in carico
Premere [2] per rifiurare
->1
```

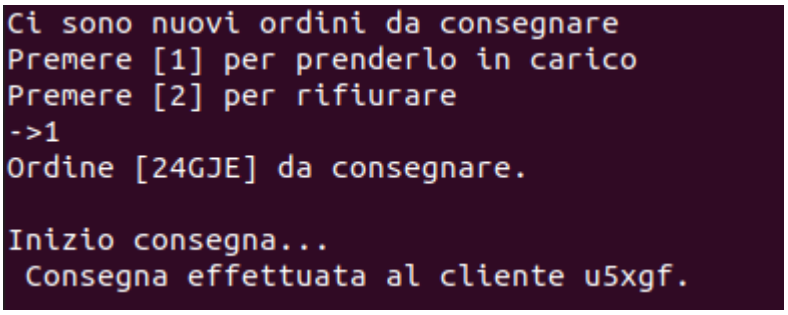
Quando si sceglie di prendere in carico un ordine, non si ha la certezza di ottenerlo per la consegna.
Per cui, nel caso in cui il ristorante abbia già assegnato l'ordine ad un altro rider, viene visualizzato un messaggio con la possibilità di aggiornare per trovare nuovi ordini disponibili per la consegna.

```
Ci sono nuovi ordini da consegnare
Premere [1] per prenderlo in carico
Premere [2] per rifiurare
->1
Non è stato possibile prendere in carico un ordine

Attualmente non c'è nessun ordine da consegnare.
Premi [0] per aggiorare.
->
```

Altrimenti, se si è preso l'ordine in carico viene visualizzato:

```
Ci sono nuovi ordini da consegnare
Premere [1] per prenderlo in carico
Premere [2] per rifiurare
->1
Ordine [24GJE] da consegnare.

Inizio consegna...
```

Dopo 5 secondi, per simulare la consegna dell'ordine, viene visualizzato 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.