DOCUMENTAZIONE PROGETTO RETI INFORMATICHE 2022-23

INFORMAZIONI GENERALI

L'applicazione segue il paradigma client-server: *cli*, *td* e *kd*[1] comunicano esclusivamente con il *server*, scambiando messaggi attraverso socket bloccanti. In questo modo, sotto l'ipotesi che non si verifichino mai invii/ricezioni parziali, basterà una sola chiamata a send()/recv() per inviare/ricevere un messaggio. Per evitare poi che un processo si blocchi cercando di leggere da un socket non pronto, i socket creati e lo stdin vengono inseriti in un set, e viene letto da essi solo se, a seguito della chiamata alla primitiva select(), questi risultano pronti in lettura.

E' stato adottato TCP come protocollo di trasporto, ritenendo per questo tipo di applicazione più importante l'affidabilità della comunicazione rispetto alla velocità (es. che le comande arrivino correttamente ad un *kd*, o che il conto arrivi correttamente ad un *td*).

Un *device*, dunque, una volta avviato, deve prima instaurare una connessione TCP con il server. A questo segue una fase in cui gli invia un codice di lunghezza fissata per identificarsi (e cioè, per indicare se è un *cli*, *td* o *kd*). Questo è utile al server, che così può sapere in ogni momento quanti sono i processi connessi per ogni categoria e quali socket di comunicazione usare per contattarli. Questa fase può fallire nel caso in cui il server avesse raggiunto il numero massimo di connessioni attive con la categoria di processi cui appartiene il *device*; in tal caso, il *device* viene terminato.

Passata l'identificazione con successo, può avvenire lo scambio di messaggi. E' stato adottato un protocollo text, comodo per scambiare informazioni che sono soprattutto testuali: prima viene inviato un dato di dimensione fissata contenente la lunghezza del messaggio; poi una stringa contenente il messaggio vero e proprio, che ha tipicamente il seguente formato:

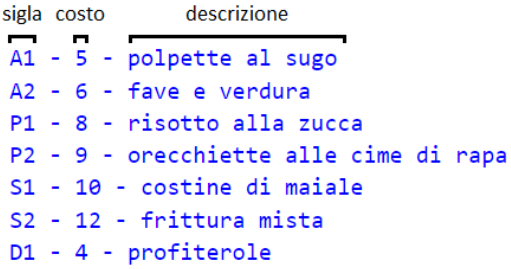
COMANDO {argomento1, … , argomentoN}

La lunghezza del messaggio non viene mandata nel caso in cui il destinatario si aspettasse un messaggio di lunghezza fissata. Similmente, "COMANDO", che serve al destinatario per capire per quale motivo è stato contattato, viene omesso nel caso in cui lo sapesse già (un esempio di questi due casi si ha nella fase di identificazione precedentemente descritta).

L'applicazione pone diversi limiti: il *server* può avere un numero massimo di connessioni attive con ogni categoria di *client*; si può avere un numero massimo di comande in attesa nello stesso momento; si può inviare un numero massimo di comande per pasto; un *kd* può gestire contemporaneamente un numero massimo di comande… Questo è stato fatto per poter usare strutture dati statiche, più semplici da controllare rispetto a strutture dati dinamiche. Le costanti che indicano tutti questi numeri massimi sono modificabili nei relativi file sorgente.

SERVER

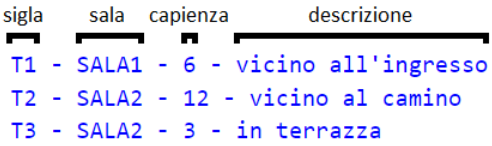
Le informazioni sul menu e sui tavoli presenti nel ristorante vengono recuperate da file, così che sia più semplice modificarle. Il file "menu" deve avere il seguente formato:



Vanno scritti in ordine prima gli antipasti, poi i primi, i secondi e infine i dolci, scegliendo numeri progressivi per piatti della stessa categoria (es. dopo P1 può esserci necessariamente P2). La voce può essere lunga un numero massimo di caratteri e si può inserire un numero massimo di piatti per ogni categoria.

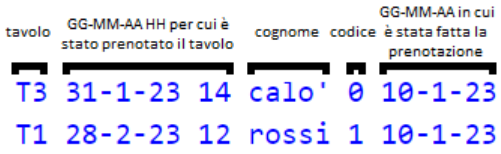
[1] Verrà usato il termine *device* per indicare genericamente un processo che appartiene ad una di queste tre categorie.

Il file “tavoli” deve avere il seguente formato:



Vanno scelti dei numeri progressivi (es. dopo T1 può esserci necessariamente T2). La descrizione della sala e la voce complessivamente devono essere lunghe un numero massimo di caratteri.
Poiché questi due file saranno modificati esclusivamente dal gestore del ristorante (di cui ci si può fidare), il server assume che siano scritti nel modo corretto.

Anche le prenotazioni sono gestite attraverso il file “prenotazioni.txt”:



In caso di errore nella comunicazione con un *device* o di arresto forzato dello stesso, il *server* chiude il socket di comunicazione interessato e rimuove dalle proprie strutture dati tutte le informazioni riguardanti tale *device*. In più:

- Se è un *td*, il *server* avvisa i *kd* di smettere di preparare le comande uscite dal tavolo e rimuove tutte le comande in attesa che aveva inviato;
- Se è un *kd*, il *server* fa passare allo stato “in servizio” tutte le comande che aveva accettato. Questo viene fatto per evitare che rimangano nello stato “in preparazione” anche quando non lo sono, e che quindi il *server* non possa arrestarsi (quando invece potrebbe) invocando il comando “stop”.

CLI

Quando l’utente invoca il comando “find”, prima di inviare la richiesta di prenotazione al server, il *cli* controlla che la data di prenotazione GG-MM-AA HH inserita sia una data futura. Questa viene poi ricontrollata quando l’utente invoca il comando “book”, in quanto potrebbe non essere più futura. In questi controlli, “AA” viene considerato un anno del secolo corrente.

TD

Superata con successo la fase di identificazione, il *td* chiede il numero del tavolo in cui è installato. Inserito un numero valido, si blocca in attesa che l’utente inserisca un codice di prenotazione; questo viene poi inviato al *server* che controlla se, per la data corrente, esiste una prenotazione per quel tavolo identificata da tale codice.
Inserito un codice valido, l’utente può chiedere il menù, che verrà mandato dal *server* e mostrato poi a video, senza essere memorizzato nel *td*. Questo è stato fatto per non avere nel *td* la stessa struttura dati presente nel *server*, con l’onere di doverla mantenerla coerente tra i due codici.
Letto il menù, l’utente può inviare una comanda. Questa passerà allo stato “in attesa” solo se è valida, e cioè se contiene almeno un’ordinazione di un piatto effettivamente presente nel menù.
Alla fine, l’utente può chiedere il conto. Le comande ancora “in attesa” non verranno conteggiate.

KD

Attraverso il comando “take”, il *kd* riceve la comanda “in attesa” da più tempo e la memorizza in questa struttura:

```
struct comanda{
    int numero_tavolo;          //tavolo da cui e' uscita la comanda
    int numero_comanda;         //numero della comanda
    char descrizione[BUFLEN];   //contenuto della comanda
};
```