Documentazione Progetto Reti Informatiche
Università di Pisa – Anno 2022/2023

Edoardo Cremente – Mat. 564817

A primo impatto, aperta la cartella del progetto, ci si trova davanti a un gran numero di file. Il motivo dietro questa scelta è stato il fatto che ho preferito, anche se usando il linguaggio C, avere un approccio tipico della programmazione a oggetti. Ogni "Classe" di cui ho bisogno è stata posta, con la sua relativa struct, in un file .h, accompagnata dalla dichiarazione di tutti i suoi "metodi", che non sono altro che funzioni che riguardano da vicino la nostra struct, e sono quindi considerabili concettualmente come "metodi" della nostra "classe". Ogni file .h è accompagnato anche, ovviamente, da un file .c con lo stesso nome, che contiene l'implementazione di tutte le funzioni dichiarate nel .h. Le uniche eccezioni a questo modus operandi sono i file che costituiranno i nostri eseguibili: server.c, td.c, cli.c e kd.c. Anziché avere un file .h con lo stesso nome, questi file hanno tutti un relativo file funz_serv/td/cli/kd che agisce da .h, con la dichiarazione di tutte le funzioni utilizzate, ed è solo quest'ultimo ad avere un ulteriore file .c con l'implementazione di queste funzioni. È presente anche un file head e un file utility, con intestazioni varie e funzioni di utilità generica, non concettualmente attribuibili a nessuna classe in particolare. Questa scelta implementativa mi ha permesso di lavorare in maniera molto più ordinata anche dopo la progressiva crescita di dimensione del progetto, a costo di una leggera complicazione del makefile, che è risultato un po' più lungo da scrivere.

Per l'implementazione dei socket, ho scelto la modalità listen() e select() anziché la modalità fork(). Semplicemente, dato l'elevato potenziale numero di connessioni contemporanee, l'uso della listen e select semplifica enormemente l'implementazione del progetto. All'arrivo di un messaggio sulla rete, quando un socket è pronto, possiamo stabilire da quale fonte è arrivato il messaggio (che sia da stdin, o da un altro socket) e agire di conseguenza chiamando relative funzioni che rispondano al comando digitato. Il protocollo di trasporto è TCP, dal momento che per questo progetto era di gran lunga più importante la correttezza del messaggio, piuttosto che la velocità con il quale il messaggio viene recapitato. Tutto il progetto si basa sulla chiamata di funzioni in base al ricevimento di diversi comandi, i quali a volte possono anche essere accompagnati da diversi parametri. È di fondamentale importanza che i comandi arrivino, e vengano letti, in maniera corretta.

Non ho posto un limite al numero di client e kd che possono connettersi contemporaneamente al server, mentre ho presupposto invece che nel ristorante fossero presenti 9 tavoli, quindi il massimo numero di td che possono connettersi contemporaneamente è 9. Tenere comunque presente che la coda del listener è stata inserita a 10 puramente sapendo che essendo questo un progetto a scopo didattico, è sufficiente solo un numero limitato di connessioni per testarne le funzionalità. Quando un client (cli o device) si connette con il server, gli manda un codice identificativo, cosicché il server possa rispondere di conseguenza in base a chi gli ha mandato la richiesta di connessione. In caso di td o kd, i device vengono salvati in strutture dati con i loro relativi socket, in modo tale da poterli poi recuperare in futuro e poterci comunicare.

Per lo scambio di messaggi, il protocollo adottato è sempre il protocollo text. La maggior parte di istruzioni che viaggiano sulla rete sono messaggi composti da una parola che identifica il comando, più eventuali parametri che lo seguono. È più raro che vengano mandate per messaggio delle strutture dati, ma anche quando questo succede, ho preferito un approccio di tipo text per semplicità e tenere sott'occhio in modo chiaro i dati passati. Per fare ciò, costruisco un messaggio con i campi delle strutture dati, e il ricevente legge i campi dal messaggio e li adopera come necessario, a volte costruendo delle strutture dati ridotte con i dati che servono a lui (Esempio: il server manda una comanda al kd, e il kd si ricostruisce la comanda con la struttura ComInPrep, per "ricordarsi" delle sue comande in preparazione).

Dal momento che ho posto una limitazione sul numero di tavoli, ma per tutte le altre strutture dati non ci sono limitazioni prefissate, ho sfruttato l'utilizzo sia di strutture dati statiche che dinamiche, un array per i tavoli, e diverse liste per qualsiasi altra struttura dati, con le relative funzioni di implementazione contenute in ogni file .h/.c.

Server

All'avvio del server, mi recupero alcune informazioni che ho precedentemente inserito su file, in modo tale che il progetto sia testabile sin da subito senza aver bisogno di inserire una prenotazione a mano dal client. Le informazioni che vengono prese da file sono salvate su relative strutture dati, e sono (con esempio a seguire): i tavoli del ristorante (struct Tavolo), con il relativo id, la sala in cui sono collocati, una breve descrizione della sala (un punto di riferimento per la posizione del tavolo), e il numero di posti a sedere del tavolo;

T1 SALA1 FINESTRA 4

le prenotazioni (struct Prenotazione) effettuate da clienti in precedenza, con il relativo codice della prenotazione, il cognome del cliente, il numero di persone, la data (gg-mm-aa) e l'orario;

il menu (struct Piatto) del ristorante, con i relativi codici dei piatti, il prezzo, il nome, e una breve descrizione del piatto.
A1 - 4 - 小春卷 xiao chun juan - mini involtini primavera
È importante fare presente alcune scelte implementative. Nel menù, i codici dei piatti sono stati usati come nelle specifiche, senza particolari cambiamenti. L'utilizzo di caratteri cinesi è stata pura scelta estetica per curiosità di vedere se dessero qualche problema su Linux. Così non è stato, e li ho lasciati. Nelle prenotazioni, il codice della prenotazione è costituito nel seguente modo: codice della prenotazione precedente + 1, seguito da numero del tavolo sul quale la prenotazione è stata effettuata. Questo metodo implementativo mi ha permesso di semplificare molte delle funzioni che dovevano lavorare con prenotazioni o tavoli specifici, semplicemente avendo entrambe le informazioni in un unico dato. Questo meccanismo è trasparente al cliente, che una volta effettuata la prenotazione riceve il codice della prenotazione da lui effettuato (che per lui è solo un numero privo di significato), e una volta che il numero verrà inserito nel table device quando richiesto, penseranno le funzioni a salvare tutti i dati dove opportuno, capendo dal codice della prenotazione sia a quale prenotazione ci stiamo riferendo, sia da quale tavolo il nostro table device ci sta parlando (e quindi assegnare un numero di tavolo al table device)

Client

Il funzionamento del client è come da specifiche, non ci sono particolari scelte implementative da considerare. Di risposta al comando "find" il client riceve a schermo una lista di tavoli che soddisfano i requisiti scelti, e che il client può selezionare a piacimento con il comando "book". Tutti i tavoli che rispecchiano la scelta vengono mostrati, dal momento che da specifiche del progetto, la scelta del tavolo spetta al Client e non al ristorante. Questo vuol dire che potenzialmente un client può scegliere un tavolo da 20 posti, anche se la sua prenotazione è da 3 persone… Che dire, pagherà di più.

Table Device

All'avvio, il td chiede immediatamente di inserire il codice di una prenotazione. Finché non lo riceve, non si "avvia" e non mostra a schermo nessun dato utile. Una volta inserito un codice valido, comincia a mostrare i suoi comandi con una breve descrizione su ciò che fanno. Il cliente al tavolo è libero di chiedere il menù, e mandare un numero qualsiasi di comande, senza limitazione di quantità o tempo fra di esse. Segnalazione da fare sul comando "conto", che per mia scelta non è stato implementato in modo tale da "chiudere" il pasto, e quindi magari di conseguenza anche il processo. Questa scelta è stata presa per girare attorno al concetto del "supporre che il cliente non sia intelligente" ed evitare che appena seduto al tavolo, il cliente inserisca il comando conto e sia poi impossibilitato a mangiare, dovendosene andare. Ciò che fa il comando conto è mostrare a video il prezzo di tutte le comande ordinate fino a quel momento (indipendentemente dallo stato in cui si trovano). Può essere considerato come un promemoria per il cliente per tenere sempre sott'occhio la cifra che sta per spendere, se magari è limitato a un certo budget. Quando il cliente è soddisfatto, può semplicemente chiamare un'ultima volta il comando conto per vedere quanto deve pagare, e di fatto andare a pagare. Si può supporre che alla visione del tavolo vuoto, un cameriere venga a prelevare il table device e lo spenga (oppure il cliente lo porta alla cassa al momento di pagare…). Ovviamente tutti i table device vengono anche spenti alla chiamata del comando "stop" da parte del server, ma questo vale anche per i kd.

Kitchen Device

Anche questo implementato come da specifiche senza particolari appunti da fare. Nelle specifiche è presente una piccola ambiguità con il comando "set", che non è chiaro se deve essere "set" o "ready". Nel mio progetto è stato implementato come "set".

Strutture Dati

Breve overview delle strutture dati dinamiche utilizzate:

- struct Prenotazione: contiene le prenotazioni prese da file, e conterrà le nuove prenotazioni dei client. Quando viene inserito un codice prenotazione in un td, la prenotazione viene tolta.
- struct Tavolo: contiene i tavoli presi da file.
- struct TavoloTrovato: contiene gli id dei tavoli trovati dopo una find del client per poterli gestire meglio.
- struct Piatto: contiene i vari piatti presenti nel menu, preso da file.
- struct Comanda: contiene una comanda specificando da quale tavolo è stata mandata, che numero di comanda è, e i piatti che contiene, oltre ovviamente allo stato.
- struct Ordine: contiene la lista di piatti che compongono una comanda.
- struct TableDevice: contiene la lista di table device con i loro socket.
- struct KitchenDevice: contiene la lista di kitchen device con i loro socket.
- struct ComInPrep: struttura di ausilio per i Kitchen Device, contiene le comande in preparazione che hanno preso in carico.