DOCUMENTAZIONE PROGETTO RETI INFORMATICHE

VISIONE GENERALE DEL PROGETTO:

Il progetto di reti informatiche è stato affrontato applicando le conoscenze viste a lezioni più alcune abilità sul linguaggio di programmazione C già conosciute e poi ampliate durante il percorso per la conclusione di tale progetto.

Per l'implementazione del progetto tramite programmazione distribuita è stata utilizzata la tecnica

dell' **I/O multiplexing**, principalmente è stato utilizzato il **protocollo text** e per la gestione delle informazioni sono state utilizzate più che altro **strutture dati inizializzate in memoria**.

Il progetto è **modulare**, infatti oltre ai file di codice richiesti alla consegna sono presenti anche i file di funzione prenotazione.c e connessioni.c ed il file d'intestazione header.h, questi file sono inclusi all'interno del server:

- **Prenotazioni.c**: inizializza le strutture dati e definisce le funzioni per la gestione delle prenotazioni ricevute dal client
- **Connessioni.c**: Inizializza le strutture dati e definisce le funzioni per la gestione dei vari device in arrivo, in particolare come spiegherò più avanti permettano di associare ad ogni nuovo socket che manda una richiesta al server un tipo (C, K,T)

SCELTE E DECISIONI TECNICHE

Tornando al file connessioni.c osserviamo come la struttura dati **device** associ un tipo tp a un socket sd, in questo modo per identificare un socket, ogni volta che un device si connette al server con la primitiva connect manda anche il suo tipo. In questo modo il server attraverso il file connessioni.c sarà in grado di registrare il nuovo socket in un file **connessioni.txt** se è un nuovo arrivato, oppure se esiste gli associo il corrispettivo tipo tramite le funzioni **carica_device** e **leggi tipo**. Grazie a queste funzioni posso differenziare all'interno del for infinito dell'I/O multiplexing ciascun file descriptor attraverso il suo tipo.

Anche i device client utilizzano delle strutture dati in particolare il client utilizza la struttura dati prenotazione per gestire i dati di una nuova prenotazione da inviare al server. Esso per gestire tutte le prenotazioni utilizza un array di strutture dati annidate tra di loro chiamato **Libro delle prenotazioni**, un array di giorni prenotabili struttura dati che oltre al giorno della settimana contengono anche un altro array di strutture dati detto orari_disponibili che infine a loro volta contengono la struttura dati **tavolo**.

Le funzioni **book** e **prenotazione** che leggono o sovrascrivono il libro delle prenotazioni gestiscono direttamente la struttura dati in memoria che alla fine di ogni operazione viene salvata in un file chiamato **libro_prenotazioni.txt**, tale file viene letto solo all'accensione dal server per inizializzare la struttura dati con le prenotazioni fatte fino a quel giorno. Per identificare o settare un tavolo prenotato basta vedere se/metterlo a 0. La funzione prenotazione genera un array di tavoli che soddisfi le richieste della prenotazione mandata dal client, mentre la funzione book gestisce l'omonimo comando direttamente sulla struttura dati.

Per quanto riguarda il table e il kitchen device la struttura dati principale utilizzata è **comanda** con al suo interno un array di **piatto** anche essa una struttura dati. Tale struttura dati seppur con alcune variabili al suo interno omesse poiché superflue è definita anche all'interno dei device, poiché alcuni comandi specifici come conto nella table e show nel kitchen richiedono la gestione di tali strutture dati. Ho preferito solo per queste funzionalità indipendenti da tutto il resto dell'organico la diretta gestione delle comande da parte dei device per un risparmio di traffico dati per agevolare altre funzioni le quali necessitano forzatamente l'utilizzo di cooperazione tra server e device.

GESTIONE DELLA MEMORIA E SCALABILITA' DEL CODICE

Come spiegato nel paragrafo precedente tutti i dati che il server riceve dai device vengono immagazzinati in strutture dati e in alcuni casi come, ad esempio, le prenotazioni e le comande alla fine di alcuni comandi (precisamente di book per le prenotazioni e di comanda per le comande) vengono salvati in memoria secondaria all'interno di file (**prenotazioni.txt comande.txt**). Il salvataggio di queste informazioni all'interno del file non interessa la gestione del server, ma sono stati creati come dati utili per i gestori del ristorante affinché tali informazioni restino anche dopo lo spegnimento del server, come ad esempio il caso del file libro_prenotazioni.txt. La scelta di utilizzare strutture dati è per velocizzare il tempo di attesa dei vari device, infatti se, come in questo caso, il ristorante non è di dimensioni troppo elevate e le prenotazioni/comande che deve gestire il server in contemporanea non sono eccessive risulta più efficiente a livello di risposta del server l'utilizzo di strutture e dati già presenti in memoria principale che l'accesso ogni volta in memoria secondaria. Tuttavia se il ristorante comincia a ricevere molte prenotazioni e ad espandersi o addirittura tale server gestisce una catena di ristoranti la complessità computazionale della gestione di tali strutture dati diventa troppo elevato. Infatti tale progetto **NON** è assolutamente una soluzione scalabile e per una moltitudine di device si avrebbero molti problemi di controllo di flusso da parte del livello trasporto, in particolare il buffer di sistema rischierebbe troppo frequentemente di andare in overflow perché il server non riesce abbastanza in fretta a gestire i dati in arrivo nelle varie strutture dati causando enormi ritardi visto tutti i controlli di flusso del protocollo TCP.

Tuttavia la scelta di un **server non scalabile** era già stata presa in partenza con la scelta di I/O multiplexing poiché tale tecnica permette di risparmiare notevolmente risorse per la gestione di più socket e dunque un'efficienza più immediata per il nostro piccolo ristorante, ma tale tecnica in caso dovesse monitorare file descriptor di tanti device quanti una catena di ristoranti generebbe grossi ritardi a differenza di gestione attraverso l'utilizzo dei thread. Notiamo come l'I/O multiplexing è stato utilizzato anche nei device per gestire il socket di invio richieste al server e il file descriptor per l'input file (entrambi operano in maniera bloccante);

GESTIONE DELLA MUTUA ESCLUSIONE E PROTOCOLLI UTILIZZATI

Affinché non si generassero problemi di mutua esclusione oltre a sfruttare il fatto che le primitive send e receive sono bloccanti, i device sono stati realizzati affinché gestissero meno funzioni possibili, ma una volta mandati i comandi ricevessero solamente le informazioni a loro necessarie. Un esempio concreto di questo mio approccio è l'utilizzo nel server della struttura dati **tavoli_in_servizio** che associa a ciascun socket che riceve la richiesta da un device il corrispettivo tavolo. In questo modo il server decide a chi mandare la comanda e il numero di comanda da mandare viene gestito esclusivamente dal server, inoltre anche l'indice delle comande utilizzato per mandare a schermo le varie comande sia nei table device che nei kitchen device fa sempre riferimento a l'indice del server che viene inviato all'interno del buffer così da evitare possibili modifiche di tali variabili da parte dei device.

Il protocollo utilizzato principalmente per lo scambio di messaggi tra il server e i device è il text protocol, questo perché il server agisce in risposta ai comandi dei device spesso in **modalità verbose**:

- I client spesso devono mandare oltre al comando una o più stringhe di caratteri (come nel comando find del client o nel comando comanda del table o nel comando ready del kitchen).
- Il server deve inoltrare in risposta ai comandi dei device e non solo delle stringhe, sia semplici come PRENOTAZIONE EFFETTUA o COMANDA RICEVUTA, ma a volte anche molto più lunghe come nel caso dell'invio delle opzioni possibili dopo la digitazione del comando find da parte del client
- Inoltre il protocollo text è molto più interpretabile e facile da debuggare rispetto a quello binary motivo per cui è stato scelto per questo progetto non del tutto banale