Progetto di Reti Informatiche [A.A. 2022/2023]

1. Visione d'insieme

Il progetto è strutturato come un'applicazione **client-server**, in cui il *server* **multiservizio** gestisce le richieste in arrivo da parte di tre tipologie di dispositivi (*client*, *table* e *kitchen*), comunicando mediante scambio di messaggi, sfruttando appositi protocolli e **strutture dati**. Il server è inoltre responsabile della gestione dei dati, i quali sono salvati su **file di testo**.

2. Modalità di trasmissione dei dati

La comunicazione tra dispositivi e server avviene tramite **socket TCP**, in modalità *TEXT*. Nello specifico, quando un dispositivo tenta di connettersi al server, invia un messaggio di inizializzazione (**handshake**) tramite cui il server può identificarlo e memorizzarne la tipologia.

Il dispositivo può poi inviare richieste (`request_t`) al server, il quale risponde specificando esito dell'operazione richiesta e, se necessario, dati aggiuntivi (`response_t`). Inoltre, anche il server può notificare direttamente i dispositivi mediante messaggi (`server_message_t`).

L'utilizzo di **strutture predefinite** consente di non dover trasmettere in anticipo la dimensione del messaggio, poiché conosciuto. Ognuna delle strutture dati sopra citate è composta da un campo contenente *tipo/esito* dell'operazione e uno contenente eventuali *dati* necessari all'elaborazione.

I sistemi sono **monoprocesso** e regolano la comunicazione tramite **I/O-Multiplexing**, in modo da gestire in maniera più semplice la comunicazione tra di essi, evitando l'overhead dovuto alla creazione di nuovi processi e la complessità dell'utilizzo di thread.

L'*utilizzo di thread* sarebbe potuta essere la scelta ottimale nell'esecuzione del server, ma ipotizzando un flusso di richieste per il ristorante non troppo elevato, è stato preferito un approccio più semplice (**server iterativo**), evitando la gestione della mutua esclusione alle risorse.

3. Gestione della memoria

Per mantenere i dati, è stata sfruttata sia **memoria dinamica** che **memoria persistente** (*file di testo*).

L'**utilizzo di file** consente la **persistenza** dei dati e garantisce maggiore **robustezza** ad eventuali guasti, permettendone nel caso un ripristino efficiente. Inoltre, per alcuni elementi (*menu, organizzazione dei tavoli* e *prenotazioni*), la persistenza risulta una proprietà **necessaria**, in quanto devono essere mantenuti anche oltre la chiusura del server.

Di contro, l'utilizzo di file comporta un overhead dovuto alla lettura/scrittura su disco. Per **ridurre il numero di accessi**, viene sfruttata anche la **memoria dinamica**, che mantiene i dati la cui **rilevanza è limitata** nel tempo: dati su *dispositivi connessi, tavoli attualmente occupati* e *proposte di prenotazione* sono infatti deallocati al termine della loro utilità. Per quanto riguarda le *comande*, è stato deciso di mantenerle anche in memoria dinamica finché non vengono servite, poiché in questa prima fase devono essere consultate più volte.

Per **ridurre l'overhead** dovuto all'accesso ai file, è stato optato per la **frammentazione** dei file in documenti più piccoli, **indicizzati** per data, ipotizzando di mantenere in memoria dati relativi alla

ristorazione anche dopo il loro periodo di utilizzo, come a creare uno storico relativo all'attività del locale.

4. Dettagli implementativi

L'I/O Multiplexing ha consentito ai dispositivi di essere sensibili sia a comandi in input, sia ai segnali ricevuti dagli altri dispositivi. Eventuali parametri di ogni operazione vengono convalidati sia dal device che dal server, il quale esegue eventuali controlli aggiuntivi e più approfonditi.

4.1 Client device.

Tramite il client l'utente può richiedere una prenotazione.

- **Find**: Il client inoltra al server i dati di prenotazione. Questo recupera i dati sui tavoli disponibili e li invia in risposta, memorizzando in una lista la richiesta effettuata. Il client salva la risposta ricevuta.
- **Book**: Specificata l'opzione, viene recuperato il tavolo relativo all'opzione selezionata e inviato al server che tenta di inserire la prenotazione, verificando che non entri in conflitto con altre già registrate, nel caso più client tentino la stessa prenotazione insieme. Registrato, il server risponde con i dettagli dell'operazione ed elimina dalla memoria le informazioni della proposta.
- **Esc**: Il processo client termina, chiudendo la connessione.

4.2 Table device.

Al cliente, giunto al ristorante, viene dato il *TD* con il quale effettuare il login: il device è sbloccato inserendo il codice ricevuto in fase di prenotazione, che viene trasmesso al server e verificato. In caso di esito positivo, è aggiornato lo stato della prenotazione (in *CORSO*), viene memorizzata l'associazione *device-prenotazione* tramite un'apposita lista e il pasto può cominciare.

- **Menù**: Il *TD* contatta il server, il quale risponde inviando il menù più recente disponibile.
- **Comanda**: Il server controlla che i piatti siano effettivamente presenti. Poi, la comanda entra in *ATTESA* e inserita nel relativo file e in memoria, nella lista ordinata delle comande *non in SERVIZIO*. Viene poi notificata ai *KD* la presenza di una nuova comanda in attesa.
- **Conto**: Il cliente può in qualunque momento chiedere il conto. Il server addebita le comande in *PREPARAZIONE* o in *SERVIZIO* consultando i file, inviando la spesa al *TD*. Dopodiché, il server rimuove i dati in memoria dinamica relativi al pasto, inviando notifiche ai *TD* in caso ci fossero ancora comande in attesa. Infine, pone la prenotazione in stato *TERMINATO*.

4.3 Kitchen device.

Alla connessione, il server invia il numero di comande in attesa.

- **Take**: Il server scorre dal fondo la lista delle comande non in servizio, per cui sono memorizzati *TD* di appartenenza e *KD* che l'ha presa in carico (*-1* se in attesa). Imposta il *KD* della prima in attesa, inoltra lo stato (*PREPARAZIONE*) nel file e invia al *KD* la comanda, mentre agli altri una notifica dell'evento. Invia anche una notifica al *TD* per aggiornare lo stato della comanda.
- **Show**: Mostra le comande attualmente in carico, memorizzate dal *KD* al termine della *TAKE*.

- **Ready**: Il server verifica che la comanda sia sempre presente e che sia in carico al *KD*. Poi, ne imposta lo stato (*SERVIZIO*) nel file e la elimina dalla lista, notificando al *TD* l'update dello stato.

4.4 Server.

Per quanto riguarda le azioni disponibili:

- **Stat**: A causa della frammentazione, nel caso delle comande giornaliere, sono consultati i file della giornata corrente e precedente, in modo da gestire casi in cui il pasto si protrae oltre la mezzanotte. Negli altri casi, per ogni pasto in corso, si accede al file della relativa giornata.
- **Stop**: Il processo server termina se le comande sono in servizio. In caso, disconnette tutti i device con uno specifico messaggio e ne verifica la chiusura. Se è in corso un pasto viene inviato il conto.

5. Considerazioni

5.1 Scalabilità orizzontale.

Nella gestione di più ristoranti, il sistema **scalerebbe bene orizzontalmente**, nell'ottica di dedicare un server a ciascuno di questi.

5.2 Scalabilità verticale.

L'I/O multiplexing ha lo svantaggio che l'aumento della frequenza delle operazioni richiede un maggiore impiego di risorse. Il frequente accesso ai file costituirebbe un altro problema. La soluzione migliore sarebbe convertire, almeno il server, a un **sistema basato su thread** (più complesso), e/o **limitare ulteriormente l'accesso ai file**, per esempio registrando le comande solo quando entrano in servizio, soluzione che però diminuirebbe la robustezza ai guasti.

5.3 Robustezza ai guasti.

La soluzione è progettata per essere **abbastanza robusta ad eventuali guasti**, cercando di gestire al meglio la chiusura improvvisa dei device connessi e salvando in memoria persistente lo stato delle comande, a discapito dell'efficienza. Questo, implementando un'apposita procedura, potrebbe però permettere un completo ripristino del sistema.