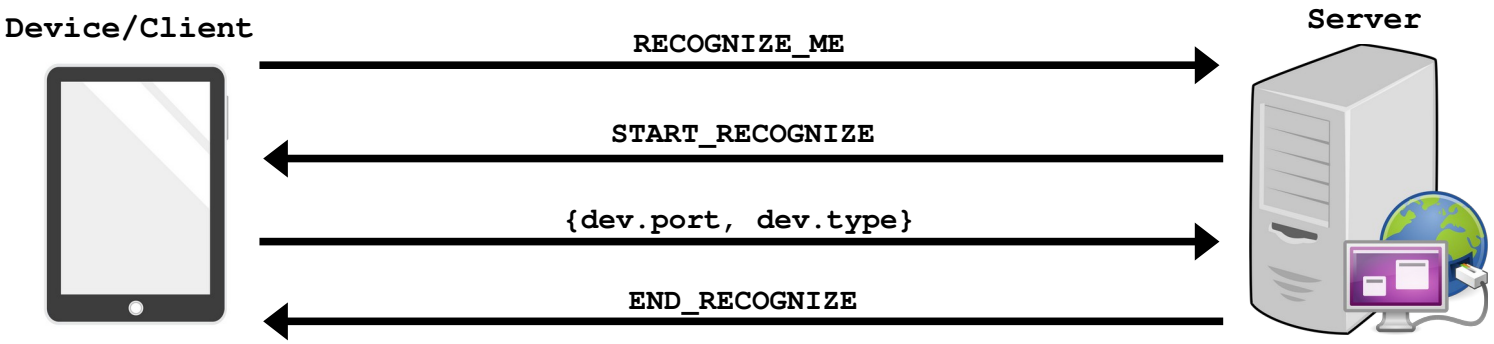
Relazione di Progetto
Reti Informatiche A.A. 2022/2023

Dario Antonio Lassoni
Matricola: 565721

L'architettura adottata è la client-server, poiché compatibile con lo scenario presentato. Quest'ultimo prevede un unico server multiservizio in comunicazione con i vari device, i quali agiscono da client.

Inoltre, si è deciso di implementare l'I/O Multiplexing, utilizzando la primitiva `select` per gestire in maniera concorrente più descrittori (socket e descrittore STDIN).

La scelta di utilizzare il socket con protocollo TCP è motivata dalla necessità di evitare la perdita di pacchetti durante la comunicazione. Tutte le informazioni scambiate tra i device e il server sono fondamentali fin dalle prime fasi, non solo per stabilire la connessione, ma anche per il successivo riconoscimento della tipologia di device mediante il comando **`RECOGNIZE_ME`**.



La fase di riconoscimento device è composta nel seguente modo:

1. Invio richiesta di riconoscimento (**`RECOGNIZE_ME`**) (client → server)
2. Risposta con segnale di inizio fase riconoscimento (**`START_RECOGNIZE`**) (client ← server)
3. Invio dei dati relativi al device {**`dev.port, dev.type`**} (client → server)
4. Risposta con ACK (**`END_RECOGNIZE`**) a segnalare il completamento del riconoscimento (client ← server)

Nonostante non fosse indispensabile, si è scelto comunque di utilizzare l'informazione della porta passata per argomento ai device in fase di avvio, trasmettendola al server.
Questa agisce come chiave univoca di sessione, garantendo che un solo device alla volta possa collegarsi al server con la stessa porta, indipendentemente dalla tipologia di device.

Lo scambio di messaggi tra i device e il server avviene attraverso le funzioni `send_data(int sd, void* buf)` e `receive_data(int sd, void** buf)` (definite in `libs/common_utils.c`), wrapper rispettivamente della `send()` e della `recv()`.
La `send_data()` svolge due operazioni distinte: inizialmente, esegue una `send()` per trasmettere l'informazione sulla lunghezza del dato che verrà inviato; successivamente, effettua una seconda `send()` contenente il dato effettivo.

La `receive_data()` riceve l'informazione della lunghezza del dato e successivamente dealloca lo spazio di memoria puntato dal buffer (utilizzando la funzione `free_mem(void** ptr)`) e alloca un nuovo spazio di memoria grande quanto la dimensione ricevuta dalla prima `send()` contenuta nella `send_data()`.

Ogni deallocazione di memoria viene effettuata attraverso la funzione `free_mem(void** ptr)` (definita in `libs/common_utils.c`), wrapper della `free()`, che si limita a fare dei controlli in più per limitare il più possibile gli errori nell'utilizzo della free().

Nel progetto si è deciso di adottare il Text Protocol, utilizzando sempre delle stringhe (`char*`) come buffer di comunicazione.

La scelta è stata guidata principalmente da due fattori:
1. il contesto non prevede necessariamente che i messaggi scambiati tra i device e il server debbano essere "mascherati", non trattandosì di informazioni "segrete";
2. le strutture condivise tra i vari attori sono immediatamente riconoscibili e il parsing è praticamente immediato, soprattutto per le stringhe.

Per evitare la progettazione di una struttura dati per ogni singolo comando dato in STDIN dai device/client, si è deciso di progettarne una generica che contenga tutte le informazioni necessarie.
Tale struttura (definita in `libs/common_header.h`) è definita in questo modo:

```
struct cmd_struct {
        char* cmd;
        void* args[6];
};
```

Il campo `char* cmd` contiene uno dei comandi dati in input (`find, book, login,` ecc...) mentre il campo successivo (`void* args[6]`) è formato da 6 puntatori a tipi generici, che possono variare in base al comando dato in input. La creazione e allocazione dei dati per questa struttura viene effettuata attraverso delle funzioni apposite per comando (es. per il comando `order` è presente la funzione `create_cmd_struct_order()`).

In questa famiglia di funzioni (`create_cmd_struct_<COMANDO>`) avviene inoltre un check preliminare dove si verifica che i dati passati in input siano corretti per il comando specificato. Queste funzioni sono dunque utili non solo al server in fase di ricezione del dato e successivo parsing, ma anche per il client/device così che possano accertarsi che il comando inviato in STDIN sia valido, ancora prima di inviarlo.
Per i comandi che non hanno invece bisogno di alcun argomento, ad esempio per il comando `take` del Kitchen Device, viene omesso l'utilizzo di tale struttura, in quanto la validazione è immediata e non sono presenti argomenti.

Per quanto riguarda lato persistenza, su file troveremo la mappa dei tavoli (`database/table_map.txt`), la lista delle prenotazioni effettuate (`database/booking.txt`) e il menu giornaliero (`database/menu_dishes.txt`), mentre si è scelto di mantenere in memoria le comande, legandole alla singola sessione del Table Device collegato con un codice di prenotazione valido.