Sistema di ticketing help-desk

Giulio Carapelli — giulio.carapelli@edu.unifi.it — Matricola: 7169515

Richiesta

Il progetto richiede in sintesi la realizzazione di un sistema di client-server socket based per la creazione modifica e ricerca di strutture dati rappresentanti ticket. I campi dei ticket comprendono da specifica:

1) Titolo
2) Descrizione del problema
3) Data di creazione
4) Priorità
5) Stato (aperto, chiuso, in corso)
6) L'agente assegnato alla risoluzione

Il server immagazzina e gestisce i ticket secondo la loro priorità. I Client viceversa devono fornire un'interfaccia all'utente per la creazione, la ricerca e la modifica (la modifica è riservata solo ai client autorizzati attraverso un sistema di login).

Specifiche aggiunte

Considerando la libertà offerta dal progetto per quanto riguarda le policy di funzionamento della struttura, è stato deciso di specificare che:

- Ogni client può produrre un numero arbitrario di ticket i quali in fase di creazione hanno come client assegnato il creatore, gli agenti come da specifica potranno modificarlo.
- Un client in fase di creazione di un ticket può dunque impostare Titolo, descrizione, e priorità. Titolo e descrizione sono stati considerati campi essenziali per la creazione di un ticket. La priorità ove non specificata sarà impostata di default a 0.
- La data e lo stato invece sono sempre assegnati dal server nel momento in cui riceve un nuovo ticket. La data sarà quella del momento in cui il Ticket è stato ricevuto e lo stato sarà di default aperto. Come da specifica un agente potrà modificare lo stato.
- I ticket saranno identificati univocamente solo da un id impostato dal server così da permettere ticket con campi uguali ma discriminabili
- La ricerca di un ticket all'interno del server per la modifica o la consultazione avviene attraverso la specificazione da parte di un client di un insieme di filtri (tutti i campi possibili di un ticket), non per forza tutti contemporaneamente applicabili.
- La ricerca per consultazione mostrerà tutti i dati presenti nel server relativi a/ai ticket trovati.
- La modifica di molteplici ticket non sarà consentita, un agente quindi che effettua una ricerca per la modifica che produce risultati multipli dovrà essere più specifico dell'applicazione dei filtri.

Soluzione proposta

Vista la struttura richiesta e le policies definite, diviene necessario definire i sottoproblemi da risolvere ed implementare:

1) **Come strutturare la comunicazione client-server per far fronte a diverse richieste e diverse risposte possibili ?**
2) **Come strutturare una logica basata sulla comunicazione scelta?**
3) **Quale sistema applicare per la storicizzazione lato server dei Ticket?**
4) **Come gestire la registrazione ed il login dei client definiti "Agenti"?**
5) **Come strutturare un interfaccia grafica per il client e renderla utilizzabile con la struttura di comunicazione**

1 - Struttura di comunicazione

L'approccio di comunicazione adottato è a pacchetti, realizzando due pacchetti diversi, la "Request" e la "Response". L'utilizzo di AF_INET come metodo di comunicazione tra client e server vincola all'invio e al ricevimento di stringhe tra le due entità. Questo implica che i pacchetti di richiesta e risposta andranno serializzati nell'invio e deserializzati nel ricevimento.

Descrizione del pacchetto "Request": (Codice di riferimento in Lib/Packet.h e Src/Packet.c)

Il pacchetto Request viene inviato esclusivamente dal client al server e contiene, il tipo di richiesta per discriminare tra cosa il server dovrà fare con il resto del pacchetto, l'id del client che lo invia, e l'effettivo payload ovvero, tutte le informazioni necessario per svolgere l'operazione lato server. Consideriamo come esempio la creazione di un nuovo ticket. Il pacchetto di richiesta dovrà contenere un identificativo della richiesta "Aggiungi Ticket", l'id del client che la manda, poi titolo, descrizione ed eventualmente la priorità.

Questa definizione del pacchetto Request in c si traduce come il frammento che segue:

```
typedef struct {
   RequestType type;
   int sender_id;
   union {
       Ticket new_ticket;
       TicketQuery Client_query;
       TicketQueryAndMod Agent_query;
       ...
   } data;
} RequestPacket;
```

La struttura RequestPacket contiene dunque, un intero che contiene tipo di richiesta, un intero per l'id del mittente e una struttura di tipo **union** che contiene tutte le sotto strutture rappresentanti quello che serve per le varie richieste.

Analizzando la specifica si deduce che i tipi di richiesta possibile sono:

```
typedef enum {
   REQ_SIGNIN,
   REQ_CREATE_TICKET,
   REQ_QUERY,
   REQ_QUERY_AND_MOD
} RequestType;
```

Descrizione del pacchetto “Response”:
Il pacchetto Response sarà invece inviato soltanto in risposta ad una qualsiasi richiesta, conterrà un codice che codifica il tipo di risposta ed messaggio per il client.

Questo si traduce in c come in questo frammento:

```
typedef struct {
   ResponseType type;
   char message[MAX_RESP_MSG_LEN];
} ResponsePacket;
```

Appunto come per il “RequestPackt” il tipo sarà rappresentato da un intero, il messaggio è una semplice stringa.

Analizzando la specifica si è capito che i tipi di Risposta possibili sono:

```
typedef enum {
   RESP_SING_IN_OK,
   RESP_TICKET_OK,
   RESP_QUERY_OK,
   RESP_QUERY_MOD_OK,
   RESP_AUTH_REQUIRED,
   RESP_ERROR
} ResponseType;
```

Serializzatori e deserializzatori
Per comodità il server e il client creeranno richieste e risposte come strutture che solo prima di essere inviate verranno serializzate in una stringa prima e poi deserializzate al contrario a partire da quella stringa. Sia per la RequestPacket che per la ResponsePacket si è deciso di optare per una stringa con separatore tra campi. Ogni campo viene scritto in una stringa e seguito da un simbolo di pipe. Un esempio nella creazione di un ticket.

```
Request:     |REQ_CREATE_TICKET|id_cliet|Title|Description|Priority
Response:    |RESP_TICKET_OK|Ticket added successfully|
```

2 - Logica interna

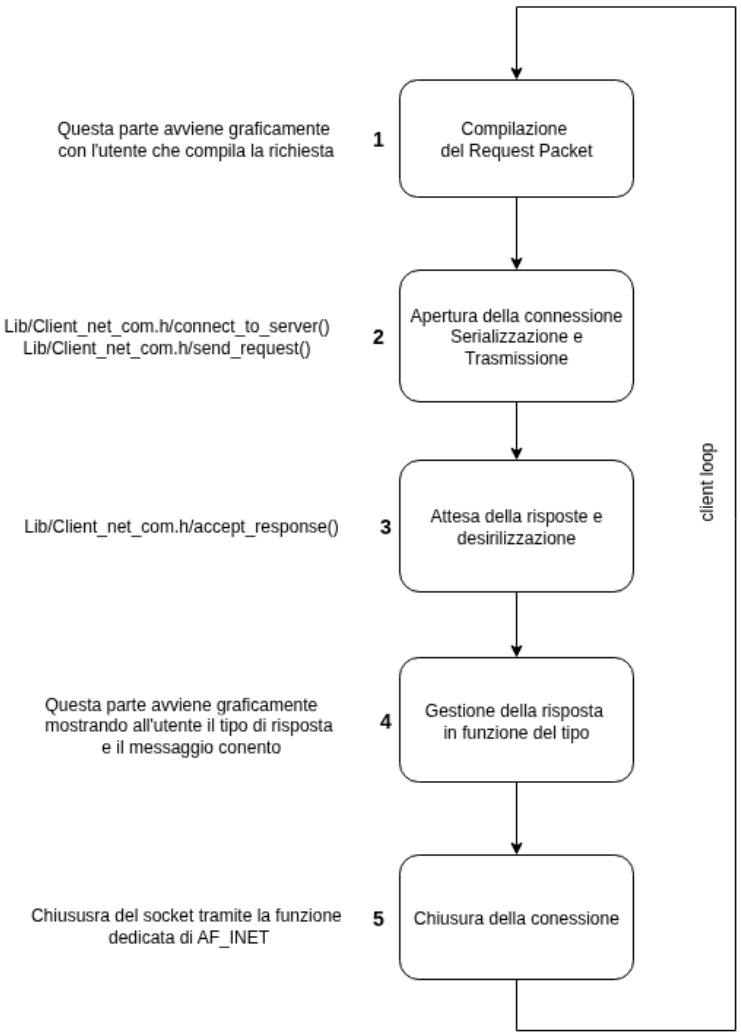
La logica interna si intende la struttura che nel client e nel server definisce come le due entità si relazionano con il socket (lettura e scrittura) per poi trattare le richieste/risposte. Ogni operazione per struttura sarà dunque associata ad una richiesta formulata dal cliente una risposta formulata dal server.

Logica funzionamento Client

(Codice di riferimento in Src/Client.c e Src/ Client_net_com.c)

Il Client si occuperà di compilare un RequestPacket. In base al tipo specifico scelto della richiesta andranno compilati campi diversi della struttura, una volta pronta, la struttura verrà serializzata, il client tenterà di connettersi con il server. In caso di connessione riuscita, trasmette la serializzazione della richiesta e si mette in attesa di una risposta sul socket. Una volta ricevuta la risposta deserializzata ed ottenuto un ResponsePacket, il client lo "elabora" in base al RES_TYPE ed infine chiude la connessione con il server. Questa serie di operazioni si troverà dentro un loop per permettere la realizzazione di nuove richieste ad ogni ciclo.

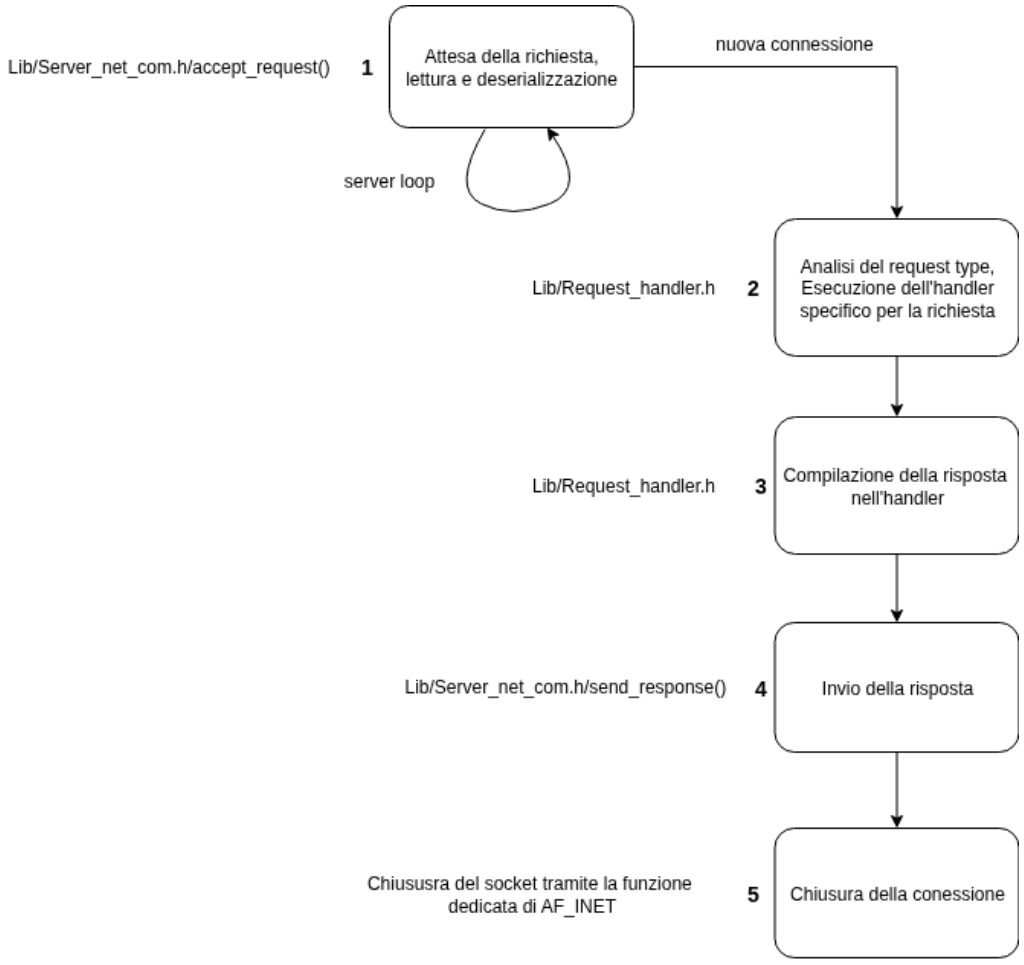
Schema logico di funzionamento:



Logica funzionamento Server (Codice di riferimento in Src/Server.c , Src/ Server_net_com.c e Src/Request_handler.c)

Il Server viceversa, inizializza il socket poi aspetta una connessione, una volta che un client si connette legge il contenuto lo del socket, lo deserializza in Un RequestPacket, in base al REQ_TYPE verrà creato un processo figlio il quale eseguirà asincronamente l'handler specifico, compilando il ResponsePackage. Fatto questo trasmette la risposta serializzata attraverso il socket e termina la connessione. Tutto ciò sarà eseguito in un loop per continuare ad accettare le richieste

Schema logico di funzionamento:



3 - Gestione ticket lato server

(Codice di riferimento in Lib/Db_interface.h e Src/Db_interface.c)

La gestione dei ticket per specifica richiede che vengano memorizzati, sia possibile consultarli e sia possibile modificarli. La gestione deve avvenire in maniera asincrona e ricordando che possono arrivare più richieste di letture contemporaneamente ma anche più richieste di scrittura contemporaneamente rischiando di creare delle inconsistenze sui ticket. Per far fronte a questo problema è stato deciso di utilizzare un database locale gestito attraverso la libreria "**sqlite3**" il quale in autonomia gestisce la concorrenza sulle scritture e permette di decidere quanto tempo rimanere in attesa in caso in cui il database sia occupato. Ogni processo figlio del server che dunque inserirà/modificherà un ticket avrà un "timeout" e quindi ritornerà un errore se una scrittura sul database richiede più di 3 secondi. Ogni Ticket sarà un record nel database in una tabella realizzata da questo frammento di codice c:

```
"CREATE TABLE IF NOT EXISTS tickets ("
     "id INTEGER PRIMARY KEY,"
     "title TEXT NOT NULL,"
     "description TEXT,"
     "date TEXT,"
     "priority INTEGER,"
     "status INTEGER,"
     "client_id INTEGER);";
```

4 - Gestione dei client con autorizzazione

(Codice di riferimento in Lib/Db_interface.h)

Per quanto riguarda la gestione degli utenti autorizzati alla modifica dei ticket si è scelto di riutilizzare la soluzione già adottata per i ticket. Gli agenti con il loro identificativo e la password saranno salvati in una seconda tabella presente nel database e gestiti mediante le funzioni di interfaccia allo stesso per aggiungere client registrati o per ritornare la password da confrontare nel momento dell'autenticazione.

5 - Interfaccia grafica del client

(Codice di riferimento in Lib/Graphics_interface.h e Lib/Graphic_primitives.c)

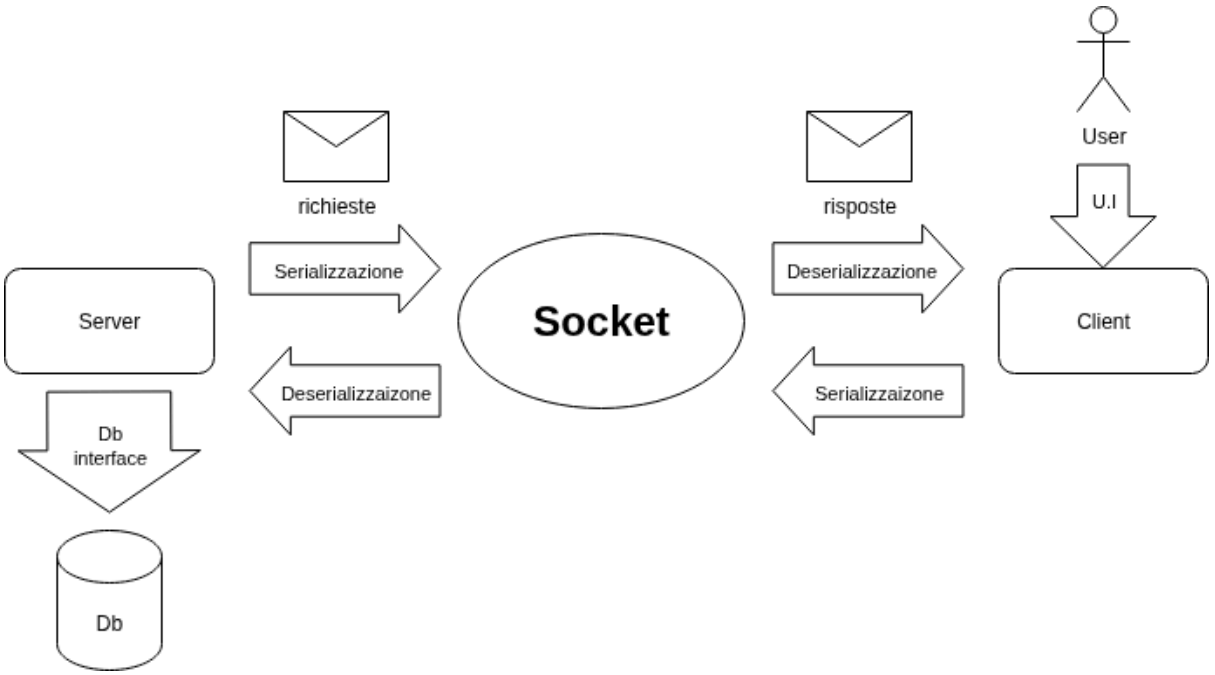
L'interfaccia grafica è un tassello fondamentale del client poiché permette in modo human-like all'utente di interagire con il server formulando delle richieste attraverso il RequestPacket. Si è deciso di realizzare una finestra principale che comprende un menù nel quale scegliere l'operazione che si desidera eseguire. In base all'operazione scelta compare una seconda finestra con il necessario. Inviata la richiesta, si presenterà sempre una finestra di risposta costruita con il ResponsePacket mostrando il tipo di risposta dividendoli tra errori e successi per poi mostrare sotto il messaggio inviato dal server. Nel menù principale a scopo di dimostrazione è stato inserito anche il bottone Sign-in al fine di dimostrare il processo di login anch'esso basato sul database. Per la realizzazione è stata scelta la "raylib" come libreria grafica.

Windows tree:

```
Menu Principale
│
├── Nuovo Ticket
│    ├── Errore (se malformato)
│    └── Risposta (se corretto)
│
├── Query Ticket
│    ├── Errore (se malformata)
│    └── Risposta (se corretta)
│
├── Login
│    └── Risposta
│
└── Modifica Ticket
     └── Form di Ricerca
          ├── Errore (se malformata)
          └── Form di Modifica
               ├── Errore (se malformato)
               └── Risposta
```

Schema totale di funzionamento

Si può ora tracciare uno schema totale di funzionamento della struttura:



Dettaglio tecnico

1- Struttura del progetto

Per quanto riguarda l'implementazione della soluzione proposta il codice è strutturato principalmente in 4 cartelle:

- **Lib** -> contiene i file header delle librerie di funzioni realizzate.
- **Src** -> contiene i file .c con l'implementazione delle suddette funzioni.
- **Obj** -> contiene i file .obj originati dalla compilazione.
- **Bin** -> contiene l'eseguibile del client e l'eseguibile del server.

Sono presenti altre cartelle, **Assets** contiene le risorse grafiche necessarie alla u.i, in questo caso un font per avere una migliore leggibilità. Poi si ha la cartella **Db** contenente il database locale. Nella root del progetto è presente un Makefile per compilare tutto il progetto.

2- Compilazione ed uso

Come detto nella descrizione della soluzione proposta, le librerie utilizzate per la grafica e il database sono raylib e sqlite.

Rispettivi comandi di installazione di raylib e sqlite (su Ubuntu)

```
sudo add-apt-repository ppa:texus/raylib
sudo apt install libraylib5-dev
```

```
sudo apt install sqlite3 libsqlite3-dev
```

Per compilare è stato usato gcc attraverso un makefile il quale dovrebbe essere sufficiente in caso di presenza delle librerie richieste.

Per quanto riguarda l'uso, è possibile trovare entrambi i binari dopo la compilazione nella cartella **Bin**. Il server non prende alcun parametro in ingresso, il client invece più prendere un intero in ingresso che verrà utilizzato come id, non va per forza specificato, in caso in cui non lo sia il client avrà id 0.

Frammento esecuzione:

```
./Server
./Client <client id>
```

L'interfaccia grafica del client al netto di alcuni possibili bug è stata resa indipendente dalla risoluzione dello schermo, per riferimento comunque i test sono avvenuti su uno schermo: 2560 x 1440 con aspect ratio di 16:9.

Limiti strutturali

Con l’implementazione della soluzione proposta si ottiene il risultato desiderato al netto di alcune situazioni critiche derivanti da alcuni limiti strutturali.

1) Nel caso in cui una ricerca per consultazione ritorni molti record dal database o più in generale dei record con dei campi molto lunghi, questo può portare ad un troncamento forzato poiché le varie righe selezionate del database verranno incluse nella stringa di risposta prodotta dal server. Se questa inclusione eccede il limite di grandezza della stringa il troncamento è una “soluzione” che permette comunque la normale esecuzione ma ovviamente non rispetta fino in fondo il vincolo aggiunto alla richiesta. Ovviamente questa cosa sarebbe sistemabile attraverso una serie di risposte multiple inviate dal server.

2) Per quanto riguarda la ricerca con modifica non è possibile verificare subito dopo aver compilato il form per la consultazione la presenza di molteplici record, bensì bisogna compilare anche quello per la modifica ed inviare la richiesta. Questo problema è relativo alla strutturazione del pacchetto, il tipo di richiesta per la modifica è diverso da quello di ricerca e quindi anche il payload è diverso. Il server per struttura nella richiesta di modifica riceve tutti i campi per filtrare i ticket e i campi che si desidera modificare entrambi nello stesso pacchetto. Questo fa sì che la verifica di unicità possa avvenire solo una volta ricevuto tutto il pacchetto ben formattato. Questo problema sarebbe risolvibile ristrutturando completamente la richiesta di ricerca e modifica attraverso molteplici richieste

Se può dunque capire che il limite comune in entrambe le criticità e il sistema mono richiesta e mono riposta per singola operazione intrapresa dall'utente. Una possibile miglioria diviene quindi la realizzazione e la strutturazione di un sistema in cui le connessioni rimangono aperte fornendo molteplici richieste o risposte spalmate su più pacchetti.

Consegna: /08/2025