Documentazione progetto di Reti Informatiche

Ingegneria Informatica UNIPI, A.A. 25/26

Baldacci Giorgio (673006)

L'applicazione distribuita **Kanban** è stata implementata con un'architettura ibrida: modello *client-server* per la connessione fra lavagna e utente, modello *peer-to-peer* per la connessione fra utenti. Questa scelta è dettata dal fatto che:

- la **lavagna** ha un ruolo centrale di gestione delle card e degli utenti connessi, quindi deve essere un riferimento costante per questi.
- gli **utenti** comunicano fra di loro saltuariamente, solo quando la lavagna avvia una nuova asta, e assumono sia il ruolo di server (attesa passiva di ricezione di tutti i costi) che di client (invio attivo del costo a tutti gli utenti). Nota inoltre che gli utenti connessi cambiano nel tempo, anche fra un'asta e quella successiva, quindi l'architettura peer-to-peer è più adatta anche grazie alla sua flessibilità.

La **lavagna** è implementata attraverso la struttura *Board* definita nell'omonimo file e, una volta inizializzata, si mette in attesa di nuove connessioni su un socket di ascolto. Una volta ricevuta una nuova connessione provvederà a creare un nuovo socket dedicato all'utente, inserito nella di descrittori da controllare *master_fds*. Tutti questi socket (ascolto e utenti) sono implementati con protocollo **TCP** per garantire affidabilità nella comunicazione. Le operazioni effettuate sono piuttosto semplici e richiedono tempi di esecuzione brevi, quindi ho ritenuto che creare un nuovo thread per ogni richiesta fosse "*over-kill*" ed ho optato per l'IO-multiplexing con la primitiva **select**. La select è bloccante, quindi per poter permettere al server di eseguire controlli periodici l'ho implementata con un **timeout** di 1 secondo. Questo è necessario per permettere al server di controllare, al più una volta al secondo, i timestamp delle cards in *DOING* e mandare **PING** o segnalare la mancanza del **PONG** di un utente inattivo.

L'**utente** è invece il cuore del progetto, dato che tutta la complessità dell'asta è delegata ad esso. Implementato tramite la struttura *User*, definita nell'omonimo file, deve inizializzare due socket: quello per la comunicazione col server, dove poi si registrerà con messaggio di **HELLO**, e quello per la comunicazione *peer-to-peer*. Anche questo è implementato con protocollo **TCP** dato che non esiste un organo regolatore delle aste ma sta ad ogni utente convergere su chi è il vincitore, quindi è fondamentale che tutti ricevano tutti i costi. Oltre a questi due socket l'utente può anche ricevere comandi dal terminale, quindi il descrittore *STDIN_FILENO* è inserito nella lista di descrittori da controllare *master_fds*. L'utente quindi si blocca con una **select** in attesa che arrivi un messaggio dal server, da un altro utente o input dal terminale. Anche qua è necessario utilizzare un **timeout**, diverso da *NULL* solo se ho una card assegnata. In tal caso mi serve a simulare il lavoro e risvegliarmi al termine di questo (impostato a 10 secondi).

L'**asta** per assegnare card in testa alla colonna *TODO* viene lanciata dal server con messaggio **AVAILABLE_CARD** quando sono rispettate le seguenti condizioni:

- almeno due utenti connessi
- almeno un utente libero
- nessun'altra asta in corso

Un utente può avere al più una card assegnata ma, secondo le specifiche, deve comunque partecipare all'asta. Il costo è quindi calcolato randomicamente in un intervallo *[0, COST_MAX-1]* se questo è libero, altrimenti è impostato a *COST_MAX* per poter perdere matematicamente l'asta. Questa scelta è sensata perché un utente occupato deve comunque poter assistere alle aste e vedere chi vince, in modo da essere sempre al corrente dello stato attuale della lavagna.
Calcolato il proprio costo, l'utente lo invia a tutti i socket peer-to-peer partecipanti all'asta e attende a sua volta di riceverlo da ogni utente. Quando entrambe le condizioni sono verificate (inviato a tutti e ricevuto da tutti) allora può, in caso di costo minimo, "festeggiare" inviando **ACK_CARD** alla lavagna e simulare il lavoro impostando il timeout.

I **messaggi** sono divisi in *header* e *payload*. L'header è comune a tutti i messaggi, contiene il tipo di messaggio e la lunghezza del payload, mentre il payload è specifico per ogni messaggio, e per alcuni non è neanche presente (**QUIT**, **SHOW_LAVAGNA**, **SEND_USER_LIST**). Tutte queste strutture sono definite in *utils.h*.
L'invio di messaggi avviene tramite le funzioni *send_msg* e *recv_msg*, che prima inviano/ricevono l'header e poi il payload, se presente. L'invio vero e proprio di ogni struttura è implementato con protocollo **binary**: secondo specifiche tutte le componenti dell'applicazione saranno eseguite sullo stesso host, quindi non è stato necessario gestire l'*endianness*.

La **struttura** del progetto è **modulare**. I file sono divisi in:
- directory *include* (file di *intestazione*), dove sono presenti definizioni di strutture dati e funzioni
- directory *src* (file *sorgente*), dove sono presenti implementazioni di funzioni e il main di lavagna e utente

La struttura delle due cartelle è speculare, eccetto per la directory *main* in *src*, quindi ciò che segue vale per entrambe. Sono suddivise in:
- directory *classes*, dove sono presenti le tre classi utilizzate nel progetto (*board, card, user*)
- directory *handlers*, dove sono presenti le funzioni utilizzate per gestire i socket attivati nel main utente o lavagna. Gli handlers del client sono a loro volta suddivisi a seconda dell'origine della richiesta in *cli_handlers* (terminale), *p2p_auction* (utenti) e *server msg* (server), raccolti nella directory *client*
- directory *network*, dove sono presenti costanti, funzioni e strutture utili alla comunicazione fra componenti dell'applicazione

Per **compilare** è presente un makefile. I comandi utilizzabili sono due:
- *make* per compilare, ottenendo così gli eseguibili *lavagna* e *utente*
- *make clean* per eliminare gli eseguibili precedentemente creati