

1 Morale

Si presenta l'implementazione di un'applicazione distribuita che realizza il metodo di gestione del lavoro basato su **kanban**. I principi che sono si sono voluti seguire sono:

- Protocollo di trasmissione livello *application* basato su stringhe ASCII separate da ritorni carrello (\n), leggibili da uomo, ispirato a SMTP. Queste stringhe vanno a rappresentare *comandi* dell'applicazione;
- Comandi fra server e client inviati su connessioni TCP, comandi fra client inviati su UDP;
- Design il più semplice e robusto possibile: no allocazioni sull'heap, client e server in multiplexing su un singolo thread.

2 Moduli

La struttura del progetto è divisa in *moduli*. Sia client che server implementano 3 moduli relativi alle funzionalità di base:

- `core/`: implementa la logica vera e propria della lavagna;
- `net/`: il *modulo di rete*, gestisce i socket e li astrae dagli altri moduli. Nel server, fornisce il concetto di `connection` con un client;
- `watch/`: implementa il meccanismo di controllo sui client attraverso `PING_USER` e `PONG_LAVAGNA`.

Per il client si ha il modulo aggiuntivo `rev/`, che si occupa della componente peer-to-peer di revisione di card fra client.

3 Costanti e strutture condivise

Si definiscono alcune costanti e strutture condivise:

- `card/`: definisce il tipo `card` di una kanban:

```

1 typedef struct {
2     card_id id;           // id (se e' 0 la card e' nulla)
3     char desc[CARD_DESC_LEN]; // testo attivita'
4     int user;             // utente che la sta implementando o l'ha implementata
5     struct tm timestamp; // timestamp dell'ultima modifica
6 } card;

```

Le card vengono allocate in una pool statica: si possono ottenere e liberare nuove card con le funzioni `alloc_card()` e `free_card(card *p)`, dalla semantica simile alle `malloc()` e `free()` della stdlib.

- `cmd/`: definisce il tipo `cmd` per i comandi scambiati fra client e server:

```

1 typedef struct {
2     cmd_type type;        // tipo di comando
3     const char *args[MAX_CMD_ARGS]; // argomenti del comando, in una lista terminata da NULL
4 } cmd;

```

Per comodità si è scelto di rappresentare gli argomenti come una lista di puntatori a stringhe, terminati da NULL.

- Per la creazione di un comando, si possono sfruttare le initializer list del C:

```

1 cmd cm = { .type = TIPO_COMANDO,
2             .args = {"arg0", "arg1", /* NULL di default ! */ }};
    e quindi serializzare usando l'helper cmd_to_buf(const cmd *cm, char *buf), che assume una dimensione di buffer predefinita:
1 char buf[CMD_BUF_SIZE];
2 cmd_to_buf(cm, buf);
3 // qui si puo' fare la send() di buf

```

- Per la ricezione di comandi si può deserializzare usando l'helper `buf_to_cmd(char *buf, cmd *cm)`, che si aspetta di trovare un comando vuoto (principalmente per la sicurezza della terminazione della lista di argomenti):

```

1 cmd cm = {0}; // tutti gli elementi di .args sono NULL !
2 buf_to_cmd(buf, &cm);
3 // ora cm puo' essere gestito

```

- `core_const.h` e `net_const.h` contengono costanti relative rispettivamente ai `core` (numero massimo di client, ecc..) e ai moduli di rete (indirizzi, numeri di porta, ecc...).

4 Gestione rete

L'idea dietro i moduli di rete è che questi separano la logica di lettura e scrittura su socket dai moduli `core`, che devono preoccuparsi solo di logica relativa alla lavagna. In particolare, abbiamo che:

- Le operazioni di *ricezione* (`recv()`) avvengono in multiplexing, e quando possibile. Questo significa appena si ha tempo per il server, e quando si hanno *"momenti morti"* per il client: ad esempio quando si fanno richieste o quando si va in attesa simulando l'elaborazione di una card;
- Le operazioni di *invio* (`send()`) sono sincrone e bloccanti, in quanto la mole di comandi inviati è ridotta.

Il compito vero e proprio dei moduli di rete è quindi:

1. Ottenere comandi dallo stack di rete del sistema operativo;
2. Smistare tali comandi fra i vari moduli. Questo si fa sfruttando una mappa definita sui tipi comando, accessibile attraverso

```

type_to_mod(cmd_type type):
/*
 * Discrimina i tipi di comando sulla base del modulo che li deve gestire
 */
typedef enum {
    CORE, // comandi standard
    WATCH, // comandi di controllo client (PING_USER, PONG_LAVAGNA)
    PEER // comandi di revisione fra peer (REVIEW_CARD, ACK REVIEW_CARD)
} mod_type;

```

3. Inviare indietro risposte, ottenute sempre dai moduli.

5 Server

Per il server questa corrispondenza è diretta, in quanto tutti i comandi ricevuti sono provenienti da client, e il modulo di rete deve quindi limitarsi a smistarli fra modulo `core` e modulo `watch`. Non esiste quindi una funzione `recv()`, ma una `listen_net()` che gestisce in multiplexing le richieste arrivate al server, oltre che ai timer usati per gestire il modulo `watch`. La sua implementazione è basata sulla `select()`:

```

1 void listen_net() {
2     // gestisci i timer
3     for (int i = 0; i < MAX_CLIENTS; i++)
4         handle_timer(&connections[i]);
5
6     // scansiona con la select
7     select(fdmax + 1, &read_set, NULL, NULL, &tv);
8
9     for (int i = 0; i <= fdmax; i++) {
10        // controlla che ci qualcosa da leggere
11    }

```

```

11     if (!FD_ISSET(i, &read_set))
12         continue;
13
14     if (i == listen_sock) { // accetta nuovo client
15         accept_client();
16     } else if (i == STDIN_FILENO) { // gestisci console amministratore
17         handle_client(&admin_conn);
18     } else { // gestisci client
19         handle_client_sock(i);
20     }
21 }
22 }
```

Notiamo che si prevede un client speciale, con socket `STDIN_FILENO`, inteso come l'"amministratore". Questo non è altro che il client di cui si prende il controllo interagendo con la riga di comando offerta dal server stesso, e viene fornito per creare nuove card effettuare operazioni di debugging (e.g. spostare card).

Si rende poi disponibile una `send_client(client_id cl_id, const cmd *cm)`.

Notiamo infine che modulo `net` e `core` del server usano astrazioni diverse per i client. Un client per il modulo rete è infatti rappresentato da una *connessione*:

```

1 typedef struct {
2     int sock;           // socket della connessione (se e' 0 la connessione e' nulla)
3     unsigned short port; // porta della connessione
4     char read_buf [CMD_BUF_SIZE]; // buffer di lettura
5     int read_len;       // caratteri letti in buffer
6 } connection;
```

mentre nel modulo `core` si mantiene informazione relativa alla kanban:

```

1 typedef struct {
2     client_id id;        // id (se e' 0 il client e' nullo)
3     client_sts sts;      // stato
4     card *handling;     // puntatore alla card che sta gestendo (se e' NULL sta aspettando una card)
5     struct timespec deadline; // deadline del timer
6     int sent_pong;       // un flag che rappresenta se si sta aspettando un ping dal client
7 } client;
```

Si nota la stretta dipendenza fra il modulo `watch` e il modulo `core`: quest'ultimo fornisce appositamente due header, `core.h` e `core_watch.h`, di cui il secondo più intimo e riservato al modulo `watch`.

6 Client

Per il client si ha una complicazione data dal fatto che è il modulo `core` a portare avanti l'elaborazione, effettuando operazioni di ricezione e invio su propria discrezione. Allo stesso tempo, il modulo di rete deve essere pronto a gestire comandi provenienti sia dal server che da altri client, e questi comandi potrebbero dover essere gestiti da uno qualsiasi fra i moduli `core`, `watch` e `rev`. Per questo si rende disponibile una funzione `recv_multi(cmd *cm, int block)`, che gestisce in multiplexing le richieste, e smista finché non si ottiene un comando diretto al modulo `core`. In tal caso si restituisce il comando, rimandando alla prossima chiamata la gestione delle richieste successive. Il flag `block` permette inoltre di specificare se si vuole che la funzione blocchi fino alla ricezione di un comando diretto a `core`, o se si vuole uscire dopo un timeout di 1 secondo. In questo modo si riescono a gestire le richieste dirette al client nei *"momenti morti"* accennati sopra.

```

1 int recv_multi(cmd *cm, int block) {
2     while (1) {
3         // scansiona con la select
4         if(select(fdmax + 1, &read_set, NULL, NULL, block ? NULL : &tv) == 0) return 0;
5
6         // c'e' qualcosa sul socket UDP?
7         unsigned short who;
8         if (FD_ISSET(udp_sock, &read_set))
9             if (handle_cmd(who, cm) > 0)
10                 return 1; // e' core, esci
11
12         // c'e' qualcosa sul socket TCP?
13         if (FD_ISSET(tcp_sock, &read_set))
14             if (handle_cmd(0, cm) > 0)
15                 return 1; // e' core, esci
16     }
17 }
```

Si rendono poi disponibili una `send_server(const cmd *cm)` per l'invio a server e una `send_peer(unsigned short who, const cmd *cm)` per l'invio a server.

L'implementazione del client diventa quindi semplice:

```

1 int client_loop() {
2     while (1) {
3         // ottieni una card
4         card c = {0};
5         unsigned short clients[MAX_CLIENTS];
6         int num_clients;
7         get_card(&c, clients, &num_clients);
8
9         // aspetta un tempo casuale
10        rand_wait();
11
12         // chiedi review
13         for (int i = 0; i < num_clients; i++) {
14             unsigned short client = clients[i];
15             req_review(client);
16         }
17
18         // processa la card
19         card_done();
20     }
21 }
```

7 Compilazione

La compilazione è gestita da `make`, con i target `make lavagna` e `make client`. Sono disponibili i target `make run_lavagna` (avvia una lavagna predefinita) e `make run_clients` (avvia 4 client) per il testing.