1 Morale

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

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

```
typedef struct {
  cmd_type type;                  // tipo di comando
  const char *args[MAX_CMD_ARGS]; // argomenti del comando, in una lista terminata da NULL
} 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:

```
cmd cm = {.type = TIPO_COMANDO,
          .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:

```
char buf[CMD_BUF_SIZE];
cmd_to_buf(cm, buf);
// 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 (per essere sicuri della terminazione della lista di argomenti):

```
cmd cm = {0}; // tutti gli elementi di .args sono NULL !
buf_to_cmd(buf, &cm);
// 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()`:

```
void listen_net() {
  // gestisci i timer
  for (int i = 0; i < MAX_CLIENTS; i++)
    handle_timer(&connections[i]);

  // scansiona con la select
  select(fdmax + 1, &read_set, NULL, NULL, &tv);

```

```
  for (int i = 0; i <= fdmax; i++) {
    // controlla che si qualcosa da leggere
    if (!FD_ISSET(i, &read_set))
      continue;

    if (i == listen_sock) {         // accetta nuovo client
      accept_client();
    } else if (i == STDIN_FILENO) { // gestisci console amminisratore
      handle_client(&admin_conn);
    } else {                        // gestisci client
      handle_client_sock(i); // chiama handle_client()
    }
  }
}
```

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)`.

A questo punto la gestione dello stato del sistemo e della logica di competenza è delegata al modulo `core`, che deve sempre portare avanti l'operazione richiesta sulla base del tipo di `cmd` fornito in `handle_command(client_id cl_id, const cmd *cm)` (chiamata da `handle_client()`).

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*:

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

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

```
typedef struct {
  client_id id;             // id (se e' 0 il client e' nullo)
  client_sts sts;           // stato
  card *handling;           // puntatore alla card che sta gestendo (se e' NULL sta aspettando una card)
  struct timespec deadline; // deadline del timer
  int sent_pong;            // un flag che rappresenta se si sta aspettando un ping dal client
} 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 le 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. Questo risulta particolarmente utile per l'implementazione dell'attesa casuale.

In questo modo si riescono quindi a gestire le richieste dirette al client nei *"momenti morti"* accennati sopra.

```
int recv_multi(cmd *cm, int block) {
  while (1) {
    // scansiona con la select
    if(select(fdmax + 1, &read_set, NULL, NULL, block ? NULL : &tv) == 0) return 0;

    // c'e' qualcosa sul socket UDP?
    unsigned short who;
    if (FD_ISSET(udp_sock, &read_set))
      recv_peer(&who, cm);
      if (handle_cmd(who, cm) > 0)
        return 1; // e' core, esci

    // c'e' qualcosa sul socket TCP?
    if (FD_ISSET(tcp_sock, &read_set))
      recv_server(cm);
      if (handle_cmd(0, cm) > 0)
        return 1; // e' core, esci
  }
}
```

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: basta chiamare la `recv_multi()` per il prossimo comando `core`, con gli altri comandi gestiti dietro le quinte. Vediamo ad esempio come ci si registra al server:

```
int hello() {
  // richiedi registrazione
  cmd cm = {.type = HELLO, .args = {"richiedo la mia registrazione"}};
  send_server(&cm);

  // attendi ack
  return get_ack(); // incapsula recv_multi()
}
```

Funzioni di questo tipo vengono chiamate in un loop che ottiene card, aspetta un tempo casuale, e restituisce la conferma di aver processato la card al server.

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.