Report di Progetto Finale del Laboratorio 2

`Sistema di Gestione delle Emergenze con Multithreading e Message Queue in C11`

Prove in itinere superate - Progetto versione ridotta

Yuchen Cao
Matricola: 616906

1 Architettura del sistema

Il sistema progettato simula un centro di gestione delle emergenze con comunicazione asincrona e gestione concorrente. Un client esterno invia richieste di emergenza tramite una coda di messaggi POSIX, mentre un server multithreaded riceve, valida e gestisce tali emergenze distribuendo i soccorritori digitali (twin) disponibili rispettando le priorità e tempo limite.

1.1 Componenti principali

Lato server

- **main.c**: Questo modulo costituisce il punto di ingresso del sistema. Inizializza tutti i componenti principali, legge i file di configurazione, apre la coda di messaggi e crea un thread per ogni emergenza ricevuta.
- **logger.c**: Questo modulo gestisce il sistema di logging concorrente su file condiviso `emergency.log`, utilizzando un mutex per garantire l'accesso sicuro da thread multipli.
- **parse_env.c**, **parse_rescuers.c**, **parse_emergency_types.c**: Questi moduli si occupano del parsing dei rispettivi file di configurazione, costruendo le strutture dati `env_config_t`, `rescuer_data_t` ed `emergency_data_t` necessarie per l'intero sistema.
- **emergency.c**: Questo modulo gestisce l'intera vita di un'istanza di emergenza: dal parsing e validazione di un messaggio, fino alla creazione dell'oggetto e alla sua eventuale deallocazione.
- **intent.c**: Questo modulo implementa l'*Intent Table*, una struttura condivisa che coordina la competizione tra thread per l'accesso alle risorse di soccorso, tramite meccanismi di mutua esclusione e controllo delle priorità.
- **worker_thread.c**: Questo modulo definisce la logica d'esecuzione associata a ciascuna emergenza, includendo il controllo delle condizioni di `TIMEOUT`, l'assegnazione dei twin, la simulazione dell'intervento e il coordinamento tra thread mediante mutex e condition variable.

Lato client

- **client.c**: Questo modulo implementa un client che invia richieste di emergenza alla coda di messaggi `"/emergenze616906"`. Supporta sia l'invio singolo (via comand line), sia l'invio in modalità `-f` (da file), includendo la possibilità di introdurre un `ritardo` temporale tra le richieste per simulare scenari realistici.

1.2 Flusso d'esecuzione del sistema

Il flusso di esecuzione del sistema è strutturato in due fasi principali: l'inizializzazione e ricezione delle emergenze e la gestione dell'emergenza nei thread.

Fase 1: Inizializzazione e ricezione delle emergenze

All'avvio del lato server, la funzione `main()` esegue le seguenti operazioni:

- Inizializza il sistema di logging tramite il modulo `logger.c`, che consente scritture sincronizzate sul file di log `emergency.log`.
- Esegue il parsing dei tre file di configurazione: `env.conf`, `rescuers.conf` e `emergency_types.conf`, tramite i moduli `parse_env.c`, `parse_rescuers.c` e `parse_emergency_types.c`.
- Inizializza *Intent table* (`intent_table_t`) tramite il modulo `intent.c`, che funge da algoritmo di coordinamento tra thread concorrenti per l'accesso ai twin digitali.
- Entra in un ciclo di ascolto continuo su una coda di messaggi `"/emergenze616906"`, ricevendo stringhe di richiesta inviate dal lato client `client.c`, le richieste vengono parsed e trasformate in istanze `emergency_withID_t` (incapsula un `emergency_t` insieme a un ID numerico univoco).
- Per ogni emergenza ricevuta, viene creato un thread usando la funzione `worker_thread()` definito nel modulo `worker_thread.c`, che gestisce l'emergenza fino al suo completamento (fino allo stato `COMPLETED` o `TIMEOUT`).

Fase 2: Gestione dell'emergenza nei thread

Ogni thread associato ad una singola emergenza tenta di ottenere le risorse necessarie (twin digitali) tramite un algoritmo coordinamento, descritto nella sezione 2.2.

- Se l'assegnazione non ha successo entro il tempo limite definito dalla priorità dell'emergenza, l'emergenza passa allo stato `TIMEOUT`. Il thread termina qui.
- Se l'assegnazione ha successo, i twin coinvolti passano allo stato `EN_ROUTE_TO_SCENE`, e l'emergenza passa allo stato `ASSIGNED`.
- Il thread crea $n + 1$ nuovi thread, dove n è il numero di twin digitali assegnati:
  - n thread che eseguono `run_twin_task()`, ognuno dei quali simula il comportamento di un soccorritore, specificamente dallo stato `EN_ROUTE_TO_SCENE` a quello `ON_SCENE`, a quello `RETURNING_TO_BASE`, a quello `IDLE`.
  - 1 thread che esegue `run_emergency_task()`, incaricato di aggiornare lo stato dell'emergenza da `ASSIGNED` a `IN_PROGRESS`, e successivamente a `COMPLETED`.
- Il thread prima di terminare attende le terminazioni delle esecuzioni dei $n + 1$ thread secondari.
- Le transizioni di stato di tutti i twin e delle emergenze vengono registrate tramite il sistema di logging.

2 Scelte progettuali

2.1 Strutture dati utilizzate e motivazioni

Oltre alle strutture dati fornite dalla specifica del progetto, ho definito le seguenti strutture dati aggiuntive.

Tracciamento con ID

Per facilitare la tracciabilità delle emergenze all'interno del sistema e rispettare la specifica di scrivere log riguardano un'emergenza, sono state introdotte due strutture che estendono le entità base con un campo `id`. Le due strutture, `emergency_request_withID_t` e `emergency_withID_t`, condividono lo stesso ID per rappresentare coerentemente la stessa emergenza in fasi diverse del ciclo di vita.

```
typedef struct {
    int                 id;
    emergency_request_t  req;
} emergency_request_withID_t;

typedef struct {
    int           id;
    emergency_t  emergency;
} emergency_withID_t;
```

Intent table

Intent table è una tabella condivisa utilizzata per coordinare la competizione tra i thread concorrenti che tentano di ottenere risorse di soccorso. Le strutture coinvolte sono le seguenti:

```
typedef struct {
    int     id;
    int     priority;
    time_t  timestamp;
    int     twin_ids[MAX_TWINS];
    int     twin_count;
} intent_t;

typedef struct {
    intent_t  *items[MAX_INTENT_ENTRIES];
    int       size;
    mtx_t     mutex;
} intent_table_t;
```

Dopo la creazione di un'emergenza, il `main` assegna tale emergenza a un thread dedicato. Questo thread calcola le risorse potenzialmente utilizzabili e genera un'istanza `intent_t`, dove:

- `id`, `priority` e `timestamp` derivano direttamente dalla struttura `emergency_withID_t`;
- `twin_ids` è un array di `int` che rappresentano gli ID di **tutti** i twin (quindi anche quelli attualmente in uno stato non `IDLE`):
  - del tipo (`rescuer_type_t`) richiesto dall'emergenza;
  - raggiungibili entro il *deadline* calcolato in base alla priorità.

L'istanza `intent_t` viene quindi inserita nella tabella globale `intent_table_t` *Intent table*, che raccoglie tutte le intenzioni di risorse attualmente attive. Per garantire la coerenza in ambienti concorrenti, tutte le operazioni di lettura/scrittura sulla *Intent table* sono protette da un `mutex`.

Questa *Intent table* viene poi utilizzata per determinare, tramite l'algoritmo coordinamento descritto in sezione 2.2, se un'emergenza possa procedere con l'assegnazione delle risorse o se esistono conflitti di priorità con altre emergenze ancora in stato `WAITING`. L'intent viene aggiornato periodicamente (attualmente ogni 1 secondo) per riflettere cambiamenti nella raggiungibilità delle risorse.

Simulazione temporale

Per ogni emergenza che ha ricevuto una assegnazione valida di risorse, vengono creati $n + 1$ thread:

- n *twin thread*, uno per ogni twin digitale assegnato, simula la sequenza di cambiamenti stati `EN_ROUTE_TO_SCENE` → `ON_SCENE` → `RETURNING_TO_BASE` → `IDLE`.
- 1 *emergency thread*, responsabile dell'avanzamento dello stato dell'emergenza `ASSIGNED` → `IN_PROGRESS` → `COMPLETED`.

Questi $n+1$ thread condividono una struttura di sincronizzazione comune, definita come segue:

```
typedef struct {
    int     arrived;
    int     returned;
    mtx_t   mutex;
    cnd_t   all_arrived;
    cnd_t   all_returned;
} emergency_sync_t;
```

Ogni thread riceve un argomento strutturato che contiene i riferimenti utili:

```
typedef struct {
    rescuer_digital_twin_t  *twin;
    emergency_withID_t      *e;
    emergency_sync_t        *sync;
} twin_arg_t;

typedef struct {
    emergency_withID_t  *e;
    emergency_sync_t    *sync;
} em_arg_t;
```

Dopo l'assegnazione delle risorse, ogni twin coinvolto risulta nello stato `EN_ROUTE_TO_SCENE` e l'emergenza è nello stato `ASSIGNED`. Di seguito viene descritto l'intero flusso di simulazione in cui i twin eseguono l'intervento e rientrano alla base. Tutte le letture e scritture sulla struttura condivisa `emergency_sync_t` sono protette da un `mutex`, al fine di garantire la coerenza tra i thread concorrenti.

1. Ogni twin thread arriva e cambia stato a `ON_SCENE`, incrementa di 1 il campo `arrived` e attende su condition variable `all_arrived`, a meno che non sia l'ultimo.
2. L'ultimo twin thread che arriva effettua una `cnd_broadcast()` sulla condizione `all_arrived`, svegliando gli altri $n - 1$ twin thread e il emergency thread.
3. Il emergency thread, al risveglio, imposta lo stato a `IN_PROGRESS`, tutti i twin ora iniziano a lavorare.
4. Al termine del lavoro, ciascun `twin thread` cambia stato a `RETURNING_TO_BASE`, incrementa di 1 il campo `returned`, e l'ultimo twin thread effettua una `cnd_signal()` su `all_returned`.
5. Il emergency thread, in attesa su `all_returned`, imposta lo stato finale a `COMPLETED`.

[*Attenzione*] I twin thread non attendono su `all_returned`; essi continuano autonomamente a simulare il ritorno al base. Pertanto, è possibile che uno o più twin siano già diventati `IDLE` mentre altri sono ancora in stato `ON_SCENE`, in base alla distanza o al `time_to_manage`.

2.2 Algoritmo coordinamento basato su priorità

Uno Pseudocodice dell'algoritmo è riportato di seguito:

```
int worker_thread(void *arg) {
    int replace_intent_counter = 0;
    int first_time = 1;
    while(true) {
        // Step 1: controlla se ci sono abbastanza numero di twin
        // raggiungibili entro il tempo limite
        if(!check_reachability(emergency, rescuer_data)) {
            / ... /
            return 0;
        }
        // Step 2: controlla se il tempo deadline e' scaduto
        if(!check_deadline(emergency)) {
            / ... /
            return 0;
        }
        // Step 3: alla prima volta si registra un intent, dalla
        // seconda in poi si aggiorna l'intent ogni 1 secondo
        if(first_time || replace_intent_counter >= 200) {
            refresh_intent(intent_table, emergency, rescuer_data,
                first_time);
            / ... /
            first_time = 0;
            replace_intent_counter = 0;
        }
        // Step 4: determina se l'emergenza corrente puo' entrare
        // nella fase di assegnazione, riprovare dopo 5ms altrimenti
        if(!can_proceed(intent_table, emergency->id)) {
            thrd_sleep(5ms);
            replace_intent_counter++;
            continue;
        }
        // Step 5: tenta di assegnare le risorse, in caso fallito
        // riprovare dopo 5ms
        rescuer_digital_twin_t *assigned_twins[MAX_TWINS];
        if(assign_rescuers_to_emergency(emergency, rescuer_data,
            assigned_twins, twin_locks)) {
            // elimina l'intent se ha successo
            unregister_intent(itable, e->id);
            // Step 6: modella il comportamento temporale dei twin
            // assegnati e dell'emergenza
            handle_emergency(emergency, assigned_twins);
            / ... /
            return 0;
        }
        thrd_sleep(5ms);
        replace_intent_counter++;
    }
}
```

L'algoritmo si compone delle seguenti fasi principali:

- **Step 1**: Con `check_reachability(emergency, rescuer_data)` si controlla che se esistono abbastanza numero di twin (anche quelli non `IDLE` attualmente) raggiungibili entro il deadline. Se la risposta è no, il `worker_thread` imposta lo stato dell'emergenza a `TIMEOUT` per motivo di distanza, il thread termina e il ciclo di vita dell'emergenza finisce qui.

- **Step 2**: Con `check_deadline(emergency)` si controlla che se l'emergenza sia ancora valida (non scaduta), se la risposta è no, imposta lo stato dell'emergenza a `TIMEOUT` per motivo di carenza di risorse, il thread termina e il ciclo di vita dell'emergenza finisce qui.
- **Step 3**: Con `refresh_intent(intent_table, emergency, rescuer_data, first_time)` si crea per ogni emergenza una struttura `intent` che descrive le sue intenzioni sulle risorse da usare. Questa viene registrata in *Intent table*. L'intent viene aggiornato periodicamente per riflettere la raggiungibilità attuale dei twin.
- **Step 4**: Con `can_proceed(intent_table, emergency->id)` il thread verifica se la sua emergenza può procedere all'assegnazione. Il criterio adottato è il seguente:
  - Si controlla se l'intent associato all'emergenza corrente è in conflitto con uno degli intent presenti nella intent table. Due intent sono considerati in conflitto se i loro array `twin_ids` hanno elementi in comune.
  - Se non esiste alcun conflitto, l'emergenza può accedere immediatamente alla procedura di assegnazione delle risorse.
  - Se esistono conflitti, si confrontano le priorità: l'emergenza corrente può procedere solo se ha una priorità più alta rispetto agli altri intent in conflitto.
  - In caso di parità di priorità, viene data precedenza all'emergenza con il `timestamp` più piccolo (cioè quella arrivata prima), garantendo così un comportamento FIFO equo tra emergenze che hanno la stessa priorità. Tuttavia, per evitare **starvation** tra emergenze che hanno la stessa priorità, si introduce un periodo di tolleranza `WINDOW_PERIOD_SEC` di 2 secondi: se il `timestamp` dell'emergenza corrente è maggiore, ma la differenza è superiore a 2 secondi rispetto a quello dell'altra emergenza che ha `timestamp` minore, allora l'emergenza corrente può comunque procedere all'assegnazione.
  - Se nessuna delle condizioni precedenti è soddisfatta, l'emergenza corrente deve attendere. Il thread si sospende per 5ms tramite `thrd_sleep()` e ripete il ciclo a partire dallo **Step 1**.

Uno Pseudocodice dello **Step 4** è riportato di seguito:

```
#define WINDOW_PERIOD_SEC 2

int can_proceed(intent_table_t *table, intent_t *candidate) {
    int res = 1;
    time_t now = time(NULL);

    for(int i = 0; i < table->size; ++i) {
        const intent_t *other = table->items[i];
        if(!other || other->id == candidate->id) continue;

        if(has_conflict(candidate, other)) {
            if(other->priority > candidate->priority) {
                res = 0;
                break;
            }
            if(other->priority == candidate->priority) {
                if(other->timestamp < candidate->timestamp &&
                   candidate->timestamp - other->timestamp <
                   WINDOW_PERIOD_SEC) {
                    res = 0;
                    break;
                }
```

```
            }
        }
    }
    return res;
}
```

- **Step 5**: Con `assign_rescuers_to_emergency(emergency, rescuer_data, assigned_twins, twin_locks)` il thread tenta di acquisire i lock tramite `mtx_trylock()` su tutti i twin `IDLE` e raggiungibili più vicini. In caso di successo, con `unregister_intent(intent_table, emergency->id)` il thread elimina l'intent dell'emergenza corrente dalla Intent table in modo da permettere gli altri thread a procedere. Aggiorna lo stato di ciascun twin a `EN_ROUTE_TO_SCENE` e lo stato di emergenza a `ASSIGNED`, avvia la simulazione dell'intervento tramite la funzione `handle_emergency()`. Nel caso contrario, il thread si sospende per 5ms tramite `thrd_sleep()` e ripete il ciclo a partire dallo **Step 1**.
- **Step 6**: Con `handle_emergency(emergency, assigned_twins)` il thread simula l'avanzamento dell'intervento come descritto in sezione 2.1 **Simulazione temporale**. Una volta la simulazione ha finito, il thread termina, il ciclo di vita dell'emergenza finisce qui.

Vantaggi dell'algoritmo proposto

L'algoritmo di coordinamento basato su *intent espliciti* e *conflitto di risorse* consente una gestione fine e ad alta concorrenza delle emergenze. A differenza di un semplice approccio FIFO o di un ordinamento statico basato esclusivamente sulla priorità, l'algoritmo proposto introduce una **negoziazione dinamica** delle risorse, in cui ogni emergenza dichiara esplicitamente le risorse richieste attraverso una struttura `intent`, registrata nella *Intent table* condivisa.

Il vantaggio principale è il bilanciamento tra **priorità globale** — che garantisce la precedenza alle emergenze più gravi e con timestamp più piccoli — e **progressione equa** tra emergenze con la stessa priorità. In particolare, viene introdotta una finestra di tolleranza temporale `WINDOW_PERIOD_SEC` di 2 secondi: se un'emergenza con priorità uguale ha un `timestamp` maggiore, ma la differenza temporale supera tale soglia, essa può comunque accedere alla fase di assegnazione. Questo meccanismo evita fenomeni di *starvation* e migliora l'efficienza. Inoltre, un'emergenza con priorità più bassa può comunque procedere, a condizione che non abbia conflitti attivi con altre emergenze di priorità superiore.

L'*aggiornamento periodico* dell'intent consente all'algoritmo di *adattarsi dinamicamente* all'evoluzione dello stato delle risorse. Un'emergenza inizialmente bloccata da conflitti può diventare immediatamente idonea all'assegnazione non appena le emergenze concorrenti aggiornano il proprio intent, escludendo i twin precedentemente contesi.

In sintesi, l'algoritmo proposto offre i seguenti vantaggi:

- Assegnazione prioritaria basata sulla gravità e sull'ordine di arrivo;
- Equità tra emergenze con la stessa priorità grazie al meccanismo di finestra temporale;
- Adattamento dinamico allo stato attuale delle risorse tramite il refresh dell'intent;
- Prevenzione di *deadlock* tramite ordering dei lock.

2.3 Sincronizzazione e concorrenza

Il sistema progettato è altamente concorrente: per ogni emergenza ricevuta e validata, il `main()` crea un thread dedicato che si occupa della gestione dell'emergenza. Tali thread competono tra

loro per l'accesso alle risorse di soccorso disponibili, utilizzando meccanismi di sincronizzazione per garantire coerenza e correttezza.

Una volta che un thread ottiene con successo le risorse richieste, esso avvia la simulazione dell'intervento creando a sua volta $n + 1$ nuovi thread: uno per ciascun `twin` assegnato (totale n), più un thread aggiuntivo per monitorare e gestire lo stato dell'emergenza.

Pertanto, per ogni emergenza accettata, il numero totale di thread coinvolti nel sistema può variare da un minimo di 1 (in caso di `TIMEOUT` immediato) fino a un massimo di $1+(n+1) = n+2$, dove n rappresenta il numero di twin assegnati all'emergenza.

Dal punto di vista della sincronizzazione, il sistema adotta un approccio mirato in cui solo le strutture effettivamente condivise o soggette a modifiche concorrenti vengono protette tramite mutex o variabili di condizione. Di seguito si descrivono le principali considerazioni:

- I tre file di configurazione `.conf` (`env.conf`, `rescuers.conf`, `emergency_types.conf`) vengono letti una sola volta dal `main` in fase di avvio. I dati risultanti (strutture `env_config_t`, `rescuer_data_t`, `emergency_data_t`) non vengono mai modificati durante l'esecuzione e sono quindi acceduti in sola lettura, senza necessità di sincronizzazione.

- Anche le strutture `emergency_withID_t` non sono condivise tra thread. Ogni emergenza è gestita unicamente dal proprio thread principale e da $n+1$ thread secondari, dove n è il numero di twin assegnati. I n thread dei twin effettuano solo operazioni in lettura (ad esempio lettura delle coordinate), pertanto non necessitano di protezione. Le uniche scritture avvengono da parte del thread principale e del thread di simulazione (`run_emergency_task`), ma questi non operano mai in parallelo: quando i thread di simulazione sono attivi, il thread principale ha già concluso la fase di assegnazione e resta passivo. Per questo motivo, anche se l'oggetto è condiviso, non sono necessari meccanismi di mutua esclusione espliciti.

- I `digital twin` rappresentano invece una risorsa condivisa tra i vari thread. Per evitare race condition durante la fase di assegnazione, ogni twin è associato a un `mutex` dedicato. La funzione `assign_rescuers_to_emergency()` prova ad acquisire questi lock con `mtx_trylock()`, li verifica nello stato `IDLE`, e successivamente aggiorna il loro stato.
  Nella fase di simulazione, i twin assegnati non sono più soggetti a concorrenza. La funzione `assign_rescuers_to_emergency()` garantisce che ogni twin venga bloccato con `mtx_trylock()` e che il suo stato sia `IDLE` prima di essere utilizzato. Una volta completata l'assegnazione, questi twin vengono rimossi implicitamente dal pool di risorse disponibili e ciascuno è gestito da un unico thread simulativo. Per questo motivo, durante `handle_emergency()`, lo stato dei twin può essere modificato in sicurezza senza ulteriori sincronizzazioni.

- La scrittura sul file di log è anch'essa protetta da un mutex globale (`log_mutex`), in modo da evitare corruzione o interleaving tra messaggi di log generati da thread diversi. Ogni chiamata a `log_event()` esegue il `mtx_lock()` prima della scrittura e il `mtx_unlock()` subito dopo.

- Infine, la struttura `intent_table_t` Intent table viene utilizzata per coordinare l'assegnazione delle risorse tra thread concorrenti. Essa contiene un mutex globale che protegge tutte le operazioni sia di lettura che di scrittura. In particolare, l'uso del lock è fondamentale per la funzione `can_proceed()`, che verifica se esistono conflitti di risorse con altri intent già registrati. In assenza di protezione, potrebbero verificarsi condizioni di inconsistenza, ad esempio durante la modifica della tabella da parte di un thread un'altro thread la sta leggendo per effettuare il confronto.

3 Istruzioni per compilare ed eseguire

Il progetto è suddiviso in due componenti principali:

- Il **server**, gestito dal file `main.c` e da tutti i moduli associati;
- Il **client**, contenuto all'interno della cartella `Client/`.

Compilazione

Per compilare il server:

```
cd ProgettoFinale
make
```

Per compilare il client:

```
cd ProgettoFinale/Client
make
```

Esecuzione

Per avviare il server

```
cd ProgettoFinale
make run
```

Per eseguire il client in modalità singola:

```
cd ProgettoFinale/Client
./client <nome_emergenza> <coord_x> <coord_y> <delay_in_secs>
```

Esempio:

```
./client Incendio 100 200 3
```

Oppure, in modalità `-f`:

```
./client -f input.txt
```

Note

- Il server crea automaticamente la coda POSIX `"/emergenze616906"` definita in `main.c`;
- Assicurarsi che la coda sia creata **prima** di eseguire il client;
- Nella cartella `Client/` si trova un file denominato `input.txt` che può essere utilizzato come parametro di `client.c` in modalità `-f`;
- Per rimuovere i file compilati:

```
make clean        # da eseguire nelle rispettive cartelle
```