Documentazione Progetto LAB 2

Biagio Montelisciani (676722)

2025-09-08

Contents

1 Introduzione

La seguente è la documentazione del progetto di gestione emergenze realizzato in C11 con programmazione concorrente da Biagio Montelisciani (matricola 676722) per l'esame di Laboratorio 2 AA 2024-2025 appello di luglio 2025.

1.1 Overview sull'architettura del sistema

Il sistema è strutturato in Server e Client. Come da specifiche, il Client invia messaggi su una message queue Posix; il Server li analizza e processa in modo concorrente. Il tutto è trascritto su un file di log. Per andare avanti in modo ordinato, il Server imposta un clock che fa un tick ogni unità di tempo (1 secondo di default, modificabile) e scandisce i momenti in cui fare l'update dello stato del sistema; intanto dei thread "Workers" processano le emergenze parallelamente. La scelta progettuale più grande è stata quella di gestire il modo in cui i rescuers si muovono nel mondo attraverso "passi" fatti ad ogni aggiornamento del sistema, e non semplicemente aspettando il tempo i arrivo e teletrasportandoli sul loro target; più informazioni al riguardo in seguito. Ci sono tre header file che permettono di modificare il comportamento del programma a piacimento: `config_server.h`, `config_log.h`, `config_client.h`.

1.2 Overview su compilazione ed esecuzione

`make` fa compilare server e client. `make run` fa compilare server e client ed eseguire il binario del server. In un'altra finestra di un terminale si può eseguire il client e il sistema inizierà a processare le emergenze richieste. `make clean` fa la pulizia degli oggetti e i binari.

2 Sistema di Logging

Per il logging ho deciso di implementare una piccola libreria atomica riutilizzabile che sfrutta threads.h (C11) e mq POSIX. Essa distingue due ruoli di processi che loggano: processo Server e processo Client. Quando un processo chiama `log_init(...)` specifica il suo ruolo e passa una struttura di configurazione. Nel server si fa partire un thread logger che riceve i messaggi di log su una message queue POSIX e, se richiesto, li scrive sul file di log specificato. Nel Client ci si limita ad inviare i messaggi sulla queue stessa.

Entrambi i processi chiamano `log_event(...)` per loggare. `log_event(...)` è una funzione che prende le informazioni essenziali per creare il messaggio (id, tipo di evento, stringa di messaggio) la stringa può essere un formato arricchibile con informazioni utili. Funziona come una `printf(format, ...)`. Se è specificato nella config passata dal processo, log event lo stampa anche a schermo, altrimenti invia soltanto. Un'altra funzione usata è `log_error_and_exit(...)` per gli errori fatali che fanno terminare i programmi. Essa prende anche una funzione studiata per far chiudere il processo in modo pulito e ordinato. In

`config_log.h` sono specificate le caratteristiche di ogni evento loggabile: il suo nome, se fa terminare il processo che lo ha loggato, se deve essere loggato (es: gli eventi di tipo `DEBUG` possono essere disattivati).

Per quanto riguarda la formattazione degli eventi, essa funziona nel modo seguente: - il processo passa nella config la stringa della sintassi preferita - il sistema di logging la usa per scriverci: - `timestamp` (stringa) calcolato da `log_event` - `id` dell'evento (stringa): è un numero oppure `N/A` - `event_type` (stringa) - `thread_name`(stringa): il thread che ha loggato l'evento - `messagge`(stringa): il messaggio formattato - è responsabilità del processo fornire una stringa di sintassi che supporta tale formato, ad esempio `"[%s] [%s] [%s] [(%s) %s]"` lo supporta. Per capire quale thread scrivere la piccola libreria di logging offre la funzione `log_register_this_thread(<nome>);` che salva il nome insieme al thread id di chi lo chiama.

3 Client

Ho cercato di rendere il client il più semplice possibile, per lasciare al server la possibilità di fare il parsing esaustivo delle richieste di emergenza. Come richiesto nelle specifiche, il client gestisce sia una input mode di un'emergenza da linea di comando, sia da file. In entrambi i casi gestisce le singole emergenze nello stesso modo: fa un **minimo parsing** per trovare il **delay** in secondi da attendere prima di inviarle, attende, le invia.

Per aprire ed usare la message queue con il Server, usa una struttura di shared memory che viene creata e condivisa dal server e contiene il nome della coda (emergenze676722)

4 Server

Il server è il processo principale, in sostanza: - alloca le strutture necessarie per gestire le emergenze nella struttura globale `server_context_t* ctx` - fa il parsing dei file di configurazione - riceve le richieste di emergenza dal processo client - le analizza e processa in modo concorrente, loggando ciò che accade - quando riceve il segnale di stop dal client o da terminale con `ctrl+c` o individua un errore, avvia il cleanup ordinato e si ferma

4.1 Architettura del Server

Il server context `ctx` è una struttura globale che contiene lo stato del sistema. È definita in `structs.h` e popolata nel `main.c`. Viene usata dai threads del server per manipolare le strutture che essa contiene.

Come sono organizzate le strutture: - ogni **rescuer type** contiene l'array dei suoi **gemelli digitali** e sono tutti protetti da un unico mutex. - ogni **emergency type** punta ai *rescuer types* che le servono - le emergenze ricevute vivono

in una coda `pq_t emergency_queue`, più informazioni al riguardo in seguito. - le emergenze attualmente in lavorazione vivono nell'array `active_emergency` protetto da mutex. - le emergenze completate e cancellate vengono immagazzinate in altre due code `pq_t` e non toccate più eccetto per comunicare le statistiche finali.

4.2 Concorrenza

Il server è, come da richiesta progettuale, scritto in modo *concorrente*. Il sistema si aggiorna ad ogni *tick* (di default: 1 secondo) e aggiorna tutte le strutture mentre le emergenze vengono processate contemporaneamente. Aumentare o diminuire il tick permette di simulare la velocità del sistema, se il tick viene messo a 0.5 secondi, ad esempio, il sistema funziona allo stesso modo ma le cose avvengono al doppio della velocità. A seguire una descrizione di come ho reso le strutture thread safe con le primitive di sincronizzazione e il ruolo di ogni thread.

4.3 Come ho reso le strutture thread safe

Delle varie componenti da mantenere *thread safe* ho cercato di isolarne il più possibile dietro a funzioni che fanno il lavoro per me. **Le code di emergenze** hanno tutte dei mutex per essere gestite parallelamente, ma sono nascosti dietro alle funzioni che uso per manipolarle. **Il sistema di logging** usa una message queue POSIX che è thread safe e dei mutex interni per gestire i nomi dei thread da tracciare e il flushing dei file, insieme a delle variabili atomiche per aggiornare dei counter. I mutex che devo attivamente manipolare sono il mutex che protegge l'**array di emergenze** `IN_PROGRESS` e il mutex che protegge i **rescuers**. Essi vengono bloccati durante l'update dalla funzione `lock_system()` che garantisce il loro blocco (e sblocco con `unlock_system()`) ordinati e permette di evitare deadlock. Vengono anche bloccati durante la ricerca di rescuer per un'emergenza da parte di un Thread Worker, in caso l'emergenza debba fare preemption.

4.4 Come ho diviso il ruolo dei thread

Il lavoro è diviso tra tre thread singoli e una thread pool: - **thread receiver**: riceve richieste di emergenza - **thread clock**: gira ogni tick - **thread updater**: aggiorna lo stato del sistema ad ogni tick - **thread workers** (più di uno): processano le emergenze e fanno preemption

4.5 thread receiver

il thread receiver è il più semplice: - riceve una richiesta dal client attraverso la coda di messaggi `emergenze676722` - la analizza e la scarta se non è corretta - la trasforma da stringa a struttura `emergency_t *e` - la mette nella coda con `pq_push(e)` Se la richiesta è un messaggio speciale di stop, si ferma. Questo

serve al server durante il cleanup per garantire che non si blocchi a ricevere messaggi.

4.6 thread clock

Il thread clock aspetta **un tick** prima di segnalare al thread updater di fare l'update del contesto del server. In questo modo il segnale di fare l'update viene inviato sicuramente una volta ogni tick. Un tick è una unità di tempo arbitraria, impostata di default ad **un secondo** ma modificabile per accelerare o decelerare la simulazione a piacere. La natura di questo progetto è *discreta* nello **spazio** (celle separate e distanza di Manhattan) e lo è anche nel **tempo**.

4.7 thread updater

Il thread updater gira ogni tick (segnalato dal thread clock) e si occupa di aggiornare lo stato dell'ambiente generale `ctx`, ovvero: - fa fare un passo ai gemelli digitali dei rescuer che si stanno muovendo (maggiori informazioni al riguardo in seguito) - aggiorna lo stato delle emergenze e dei rescuer - manda un'emergenza in timeout se sta aspettando troppo - fa passare emergenze dalla lista di priorità 0 a quella di priorità 1 se necessario - segnala i thread workers in attesa se la loro emergenza è stata cancellata, completata o altro

4.7.1 Come "camminano" i rescuers

I rescuers si spostano di cella in cella in modo discreto. Ci sono vari modi per simulare il loro spostamento, tra questi aspettare il tempo di arrivo e teletrasportarli sul posto, o far percorrere prima il percorso sull'asse x e poi sull'asse y o viceversa; io ho optato per una versione leggermente più complessa ma piacevole da vedere in caso di una eventuale stampa a schermo dei rescuer.

I miei rescuers si spostano sulla mappa approssimando una retta dal punto in cui sono a quello in cui devono arrivare. Per farlo su celle discrete ho implementato una funzione, `update_rescuer_digital_twin_position(t)` che usa una versione adattata dell'Algoritmo di Bresenham [1], implementato in `bresenham.h` e `bresenham.c` nel progetto. A seguire una breve descrizione dell'algoritmo e della modifica necessaria che gli è stata fatta e di come è integrato nel progetto.

[1] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

4.7.2 Algoritmo di Bresenham

Siano x, y le coordinate correnti e x_t, y_t le coordinate della destinazione. Introduciamo dx, dy, sx, sy, err che sono rispettivamente le distanze su X e Y tra il punto di partenza e quello di arrivo, le "direzioni" su cui andare (alto o basso, destra o sinistra) rispetto alla nostra posizione attuale, e l'errore, che riflette la differenza tra i passi da percorre su X e quelli su Y. Le seguenti regole permettono di incrementare x, y un passo alla volta per arrivare a x_t, y_t lungo il percorso più simile alla retta ideale.

$$dx = |x_t - x| \quad sx = \begin{cases} +1 & \text{se } x < x_t \\ -1 & \text{altrimenti} \end{cases}$$

$$dy = |y_t - y| \quad sy = \begin{cases} +1 & \text{se } y < y_t \\ -1 & \text{altrimenti} \end{cases}$$

$$err = dx - dy$$

$e2 \leftarrow 2 \cdot err$

if $e2 \geq -dy$ **then** $x \leftarrow x + sx$; $err \leftarrow err - dy$ **end if**

if $e2 \leq dx$ **then** $y \leftarrow y + sy$; $err \leftarrow err + dx$ **end if**

Per essere implementato nel codice, ogni struttura che lo usa deve ricordarsi il suo stato precedente $\{d_x, d_y, s_x, s_y, err, x_t, y_t\}$. I rescuer dts lo mettono nel loro campo `bresenham_trajectory_t *trajectory` e chiamano le funzioni di `bresenham.h` per cambiare traiettoria e calcolare i passi da fare.

Ad ogni tick, si calcolano le coordinate che i rescuer in movimento avranno dopo un passo. Si applica l'algoritmo sopra tante volte quante le celle che un twin può percorrere `t->rescuer->speed`. **Un passo in diagonale conta come due passi**.

4.7.3 Chi dice alle emergenze che sono completate

È il thread updater che gestisce il controllo del completamento delle emergenze in modo organico: ad ogni tick, aggiornando lo stato dei rescuers, decrementa il loro tempo di permanenza sull'emergenza che stanno gestendo e, se esso diventa 0, li distacca da quella emergenza e li rimanda alla base dopo aver decrementato il numero di rescuers a lavoro sull'emergenza.

Quando sta aggiornando lo stato delle emergenze attive, se trova un'emergenza che ha 0 rescuers ancora al lavoro, la considera completata e manda il segnale al worker che la gestisce.

4.8 thread workers

I thread workers vengono creati insieme agli altri e vivono tutto il tempo di vita del server. Manipolano prevalentemente la struttura `ctx->active_emergencies` che ha il suo mutex e un array con tante celle quanti thread workers: ognuno ha il suo slot. Il ruolo di un TW è divisibile in quattro fasi: **estrazione, ricerca, attesa, rilascio**.

4.8.1 Estrazione

Il Thread fa la pop della coda di emergenze. La pop rimane in attesa finché la coda non contiene qualcosa o viene chiusa, per questo è importante chiuderla alla fine per non bloccare gli workers. Una volta presa l'emergenza più "calda", la mette nel suo slot personale dell'array di emergenze attive.

4.8.2 Ricerca

Il TW usa una funzione per cercare i rescuer che servono ad un'emergenza. Per farlo li scorre, scartando quelli che non possono essere presi, e sceglie il migliore secondo alcuni criteri di preferenza (preferisce gli IDLE a quelli occupati, preferisce quelli più vicini etc…). Scarta automaticamente quelli troppo lontani dall'emergenza che non arriverebbero in tempo per il suo timer di Timeout.

Se non li ha trovati vuol dire che non ci sono abbastanza risorse per quella emergenza, quindi essa deve andare in Timeout. Altrimenti li invio verso la scena, attuando il meccanismo di Preemption su quelli che sono occupati da altre emergenze meno importanti e metto quelle emergenze in pausa. Le emergenze messe in pausa si ricordano quali rescuers hanno perso e quali hanno ancora e provano a recuperare i rescuers persi, intanto il loro timeout scorre e se non ne trovano, vanno in Timeout anche loro.

4.8.3 Attesa & Rilascio

Una volta trovati i rescuer per l'emergenza ed inviati verso la scena, il thread si mette in attesa sulla condition variable dell'emergenza, attende che un evento significativo avvenga all'emergenza: ad esempio che essa venga messa in pausa da un'altra, in caso riparte la sua ricerca dei rescuers; oppure che venga cancellata, allora la mette nelle cancellate; oppure che sia completata. Quando l'emergenza è stata rilasciata nella coda delle cancellate o delle completate, ne estrae un'altra e il ciclo riparte.

5 Altre aggiunte

5.1 Sistema di stop pulito

Chiamare `./client -S` o premento `ctrl+C` in qualsiasi momento nella shell del server fa interrompere il server in modo ordinato ed apparire dei log finali che dicono le statistiche del sistema (emergenze completate, cancellate etc).

5.2 Visualizzazione JSON e ASCII

Per facilitare il debugging e la visualizzazione di quello che accade ho fatto in modo che durante l'esecuzione del server tre file di testo venissero popolati con informazioni utili. I file vengono creati e aggiornati automaticamente mentre il server gira e messi nella directory`/out`.

file	scopo
`simulation.json`	Posizioni rescuers ed emergenze ad ogni tick
`positions.json`	Coordinate richieste di emergenza e basi rescuers
`positions.txt`	(ASCII) Rappresentazione grafica delle posizioni