DPLAN: Distributed PLANning

progetto finale del corso di LCS 2007/2008

autore: Federico Viscomi 412006

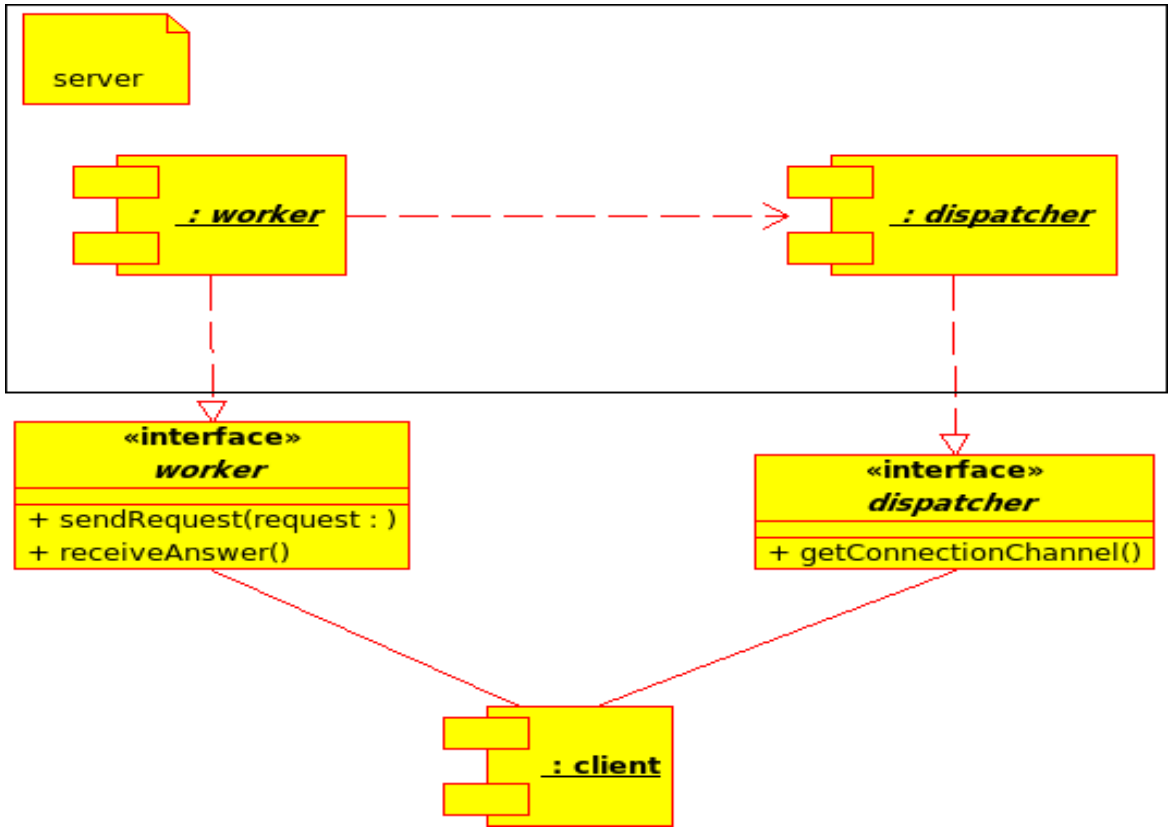
Indice

Scelte implementative e struttura del progetto

struttura del server

Il server e' diviso su vari thread:

- il thread main che: controlla gli argomenti; carica le agende in memoria; crea il canale di connessione; crea il thread dispatcher; imposta la gestione dei segnali e si mette in attesa di un SIGTERM. Quando tale segnale arriva: il thread impedisce la creazione di nuovi worker; attende la terminazione di quelli gia creati; chiude il canale di connessione; salva in memoria secondaria le agende modificate in memoria principale e infine termina il programma.
- un thread dispatcher che: in un ciclo infinito accetta connessioni dai client e crea i worker relativi. Per ogni connessione crea un nuovo thread.
- un gruppo di 0 o piu' thread worker che si occupano di servire le richieste dei client. Ogni worker riceve un messaggio, elabora le risposte le invia al client e termina.



struttura del client

Al contrario del server il client non e' multithreaded. Si comporta nel modo seguente:
fa il parsing degli argomenti; ne controlla la correttezza; prepara il messaggio di richiesta; si connette al server; invia il messaggio; riceve le risposte; chiude la connessione e termina.

gestione dei segnali

lo standard posix prevede che un segnale asincrono inviato a un processo venga ricevuto da un thread qualsiasi tra quelli che non lo bloccano. D'altro canto in LinuxThreads ogni thread e' un processo kernel dunque i segnali sono inviati sempre a un thread qualsiasi. L' implementazione di sigwait installa un dummy signal handlers per i segnali specificati. Poiche' i signal handler sono condivisi tra tutti i thread gli altri thread non possono installare un altro signal handler perche' sovrascriverebbe quello della sigwait. Supponiamo di avere un thread in attesa su una sigwait(set, sig) , e che un segnale sig in set arrivi al processo. Per i sistemi che aderiscono allo standard posix: se tutti gli altri thread nel processo bloccano sig allora e' il thread in attesa sulla sigwait a ricevere sig; altrimenti non si ha questa garanzia. Per I sistemi LinuxThread non si ha nessuna garanzia neanche nel caso in cui tutti gli altri thread nel processo blocchino sig. fortunatamente dalla versione 2.6 del kernel viene usata la Native POSIX Thread Library (NPTL) che e' strettamente POSIX-compatibile.
La soluzione adottata e' quella adatta ad un sistema POSIX:

- SIGPIPE: il thread dispatcher e il thread di terminazione sbloccano e ignorano SIGPIPE perche' non usano chiamate di sistema per le quali questo segnale ha significato; i thread worker bloccano SIGPIPE cosi' che le chiamate di sistema che potrebbero generare il segnale ritornano con un codice d'errore(ed errno impostato a EINTR) e il worker che ha generato il segnale e' libero di decidere come gestire la causa del segnale. In questo caso si sceglie di terminare l'esecuzione del worker
- SIGTERM: tutti I thread bloccano SIGTERM e il thread di terminazione rimane in attesa su una sigwait(SIGTERM).

Per quanto riguarda i LinuxThread l'handler di SIGTERM non puo' eseguire le operazione di terminazione del server perche' non sarebbe async-signal-safe anche se si tralasciasse questo problema non si potrebbe avere una soluzione corretta: se fosse un worker a ricevere SIGTERM l'handler non

potrebbe attendere la terminazione del worker stesso; piu' precisamente tale handler potrebbe attendere la terminazione di tutti gli altri worker, terminare il dispatcher e ritornare cosi' che il worker corrente faccia il suo normale decorso. Tuttavia il thread principale rimarrebbe in esecuzione e di conseguenza il server non terminerebbe oppure questo thread potrebbe essere terminato dall'handler e in tal caso il worker che ha ricevuto il segnale non potrebbe portare a termine l'interazione col client. L'uso di una sigwait(SIGTERM, sig) non e' una soluzione corretta perche' non abbiamo garanzie che sia il thread che esegue la sigwait a ricevere il segnale. Altre soluzione che prevedono l'uso di sigwait e handler non ha senso a causa del modo in cui viene implementata sigwait. In conclusione non sembra esserci una soluzione corretta alla gestione dei segnali nelle ipotesi fatte sull'architettura del server e nel caso dei LinuxThread.

gestione agende: sincronizzazione memoria principale / file system

le operazioni sulle agende che necessitano di sincronizzazione sono:

- eliminazione/creazione: queste operazioni vengono fatte sia in memoria centrale che nel file system nello stesso momento in quanto non sono particolarmente onerose.
- aggiunta/rimozione record: le operazioni di aggiunta ed eliminazione record vengono fatte solo in memoria centrale e poi riportare sul file system al momento della terminazione del server. In questo modo diminuiscono gli accessi ai file. Questa soluzione presenta un inconveniente: supponiamo che un client richieda la scrittura di un determinato evento su un agenda. Il client riceve un risposta che indica un esito positivo. Se pero' in seguito quando il server e' in fase di terminazione si verifica un errore relativo a quella agenda il client non viene notificato.

gestione delle agende: strutture dati

le agende sono gestiste da planners_table. Quest'ultimo implementa un hash con indice calcolato sul nome dell'agenda e usa le liste concatenate per risolvere le collisioni. In particolare la tabella hash e' fatta di puntatori a strutture che contengono:

- il puntatore all'elemento successivo
- il nome dell'agenda
- il contenuto dell'agenda

- un flag che indica se l'agenda e' stata modificata o meno.

gestione agende: mutua esclusione

Piu' worker nello stesso tempo possono richiedere l'accesso in lettura e scrittura alle agende quindi abbiamo una regione critica nel server. In uno scenario ideale la mutua esclusione deve garantire che:

- se un worker ha accesso in lettura-scrittura ad un record nessun altro worker vi puo' accedere.
- se non ci sono worker che accedono in lettura-scrittura ad un record allora piu' worker possano accedere in sola lettura.
- la politica di scheduling dei worker sia ottimale e non dia luogo a starvation.

La soluzione adottata e' questa:

```
reader() {
        lock(readers_count);
        if (readers_count == 0)
                lock(planners);
        readers_count++;
        unlock(reader_count);
        /* read */
        lock(readers_count);
        readers_count--;
        if (readers_count == 0)
                unlock(planners);
        unlock(reader_count);
}
writer(){
        lock(planners);
        /* write */
        unlock(planners);
}
```

dove i reader() sono i worker che servono richieste di tipo MSG_EGIORNO e MSG_EMESE, i writer sono tutti gli altri.
La soluzione adottata differisce da quella ideale perche' :

- e' piu' restrittiva: la mutua esclusione e' a livello di tutto l'insieme di agende e non dei singoli record.
- si potrebbe verificare starvation di writer().

gestione dei thread worker

i thread sono gestiti da pthread_mem. La soluzione adottata usa un hash table che contiene i pthread_t dei worker. L'indice hash e' calcolato usando il valore intero channel_t che e' il canale che usa il worker per comunicare col client. Il sistema operativo garantisce che ci sia un canale di comunicazione diverso per ogni sessione attiva. Inoltre il numero di worker che possono essere in esecuzione in un dato istante non puo' essere maggiore di SOMAXCON che e' la dimensione della hash table. Dunque non si possono verificare collisioni. Si e' deciso di non usare memoria allocata dinamicamente perche' la gestione della deallocazione avrebbe complicato la gestione dei worker. Il metodo submitNewTask se il server non e' entrato nella fase di terminazione crea un nuovo thread assegnandogli un pthread_t e incrementa la variabile count che corrisponde al numero di thread creati e non ancora terminati. Il metodo waitForThreadTermination assicura che non vengano creati altri thread usando una variabile booleana che viene testata prima della creazione di thread e si mette in attesa sulla condition variable. Il metodo setThreadNotActive che deve essere chiamato da ogni worker prima di terminare, decrementa il valore di count e se questo e' 0 invoca pthread_cond_broadcast sulla condition variable per segnalare che non ci sono piu' thread in esecuzione.

gestione degli errori

il file error_check.c contiene delle funzioni per la gestione dell'errore. Tutti gli errori vengono riportati su stderr.

README

compilare il server: make dserver

compilare il client: make dplan

avviare il server:

dserver diragenda

dove diragenda e' il path della directory che contiene le agende

Il server lavora in foreground. Se diragenda esiste, la apre in lettura e scrittura e per ogni file agenda aname contenuto nella directory carica il contenuto dell'agenda aname all'interno di una lista in memoria centrale Se diragenda non esiste viene creato. All'attivazione, il server crea anche una socket AF UNIX nella directory ./tmp

./tmp/dsock

su cui i client apriranno le connessioni con il server.

avviare il client:

Il client e` un comando Unix che puo` essere invocato con diverse modalita` ed opzioni

dplan -c aname

-> crea una nuova agenda di nome ''aname''

dplan -q aname

-> rimuove una agenda di nome ''aname'' (solo se vuota)

dplan agenda -d gg-mm-aaaa -u utente#descrizione

-> inserisce in ''agenda'' un nuovo evento

-> -d specifica la data

-> -u specifica l'utente che sta effettuando la registrazione e la descrizione dell'evento

dplan agenda -g gg-mm-aaaa

-> richiede gli eventi registrati per un certo giorno su ''agenda'' e li stampa sullo stdout

dplan agenda -m mm-aaaa

-> richiede gli eventi registrati su ‘‘agenda’’ per un certo mese e li stampa sullo stdout

dplan agenda -r pattern

-> elimina tutti gli eventi di ‘‘agenda’’ che contengono ‘‘pattern’’ come sottostringa in un qualsiasi campo

dplan

-> stampa un messaggio di uso

test

Il progetto e' stato testato con i test forniti dal docente e consegnato sulla sessione:

ssh viscomi@olivia.cli.di.unipi.it

le altre macchine del cli (fujim* e fujih*) non sono state usate perche' la connessione non aveva successo.