FSMonitor

Cristian Di Pietrantonio - 1604142

May 3, 2016

Sistema sviluppato come progetto per l'esame di Programmazione di Sistema, A.A. 2015/2016.

Contents

1 Struttura del progetto e del codice sorgente, compilazione e compatibilità

La struttura del codice vede la classica suddivisione tra interfaccia ed implementazione: ad ogni file di implementazione nella cartella principale "FSMonitor" corrisponde un file di intestazione in "FSMonitor/include"; i file di intestazione contengono le strutture dati e la dichiarazione di ogni funzione pubblica, con annessa descrizione.
Visto che deve essere disponibile sia su Windows che su Linux, il sistema presenta alcune componenti ognuna delle quali è implementata in base al sistema operativo ma che, indipendentemente da quest'ultimo, offre la stessa interfaccia. Queste componenti sono quelle che si occupano dei seguenti aspetti:

- FileSystem (*myfile.c*);
- Thread (*thread.c*);
- Time management (*time_utilities.c*, recupero ed elaborazione dei valori che rappresentano il tempo);
- Segnali (*signal_handler.c*, per gestire i segnali provenienti da tastiera);
- Mapping (*mapping.c*);
- Mutua esclusione (*syncmapping.c* e *thread_lock.c*);

Le implementazioni per Windows e Linux si trovano rispettivamente nelle cartelle "FSMonitor/win" e "FSMonitor/linux", e le interfacce (uniche) sono in "FSMonitor/include".

Tutti gli altri file di implementazione in "FSMonitor" sono indipendenti dalla piattaforma.

1.1 Librerie platform-dependent

Viene ora presentata una rassegna delle librerie dipendenti dalla piattaforma e una breve descrizione delle funzionalità più importanti.

- **myfile.c**: include tutte le funzionalità necessarie ad ottenere le informazioni sulle cartelle ed i file di interesse. In particolare la funzione *get_directory_content* ritorna una struttura contenente una lista di file e cartelle presenti nel path richiesto, con annesse informazioni (dimensione, ultima modifica, permessi, ...). La struttura ritornata deve essere indipendente dalla piattaforma: per questo motivo, e per una maggiore facilità di elaborazione, i permessi su un file sono salvati come una stringa associata al file. Su linux, la stringa che rappresenta i permessi sarà del tipo "rwxrwxrwx", mentre su Windows è una concatenazione di "ALLOW/DENY USER rwx;", in base alle ACL associate al file.
- **thread.c**: è la libreria che permette di creare thread. La funzione *create_thread* prende come input un puntatore a funzione, che è la funzione che il thread deve eseguire; per rendere l'interfaccia indipendente dalla piattaforma è stato necessario, su Windows, utilizzare una funzione wrapper da passare alla *CreateThread* di Win32 che eseguisse la vera funzione che vogliamo sia eseguita. CreateThread accetta un tipo di funzione con una firma diversa rispetto ai pthread, utilizzati nell'implementazione Linux. Altre funzioni presenti sono quella per sospendere il thread per un certo numero di secondi e quella per terminarne l'esecuzione.

- **time_utilities.c**: permette di gestire i timestamp associati ai file e contiene le funzioni per calcolare i secondi trascorsi tra l'avvio del server e la modifica di un file;
- **signal_handler.c**: permette di gestire il segnale CTRL+C proveniente da tastiera. Catturare questo segnale, interpretato come di terminazione, permette al server di avviare le procedure di pulizia per lasciare il mapping in uno stato consistente. Per ragioni di mutua esclusione che verranno chiariti più in là, la routine di terminazione viene eseguita su un thread diverso, creato dall'handler del segnale.
- **mapping.c**: questa libreria permette di gestire un file mapping. La creazione del mapping comporta la creazione del file associato; altre funzioni permettono l'apertura e la chiusura del file mapping attraverso una semplice interfaccia.
- **syncmapping.c**: questa libreria implementa un meccanismo di mutua esclusione tra thread e processi. Su Windows sono impiegati i *named mutex*; su linux i semafori.
  Importante: è impossibile cancellare in maniera sicura un semaforo alla terminazione di un server in quanto un altro server potrebbe in quel momento effettuare un'operazione di *lock* su di esso. Per questo motivo l'*unlink* del semaforo, che su linux è persistente, è da effettuare al momento di rimozione del sistema (ho creato una funzione apposita per linux da compilare, *unlink.c*).
- **thread_lock.c**: libreria che implementa un meccanismo di mutua esclusione tra soli thread. Sia su Windows che su linux sono impiegati i mutex.

1.2 Librerie platform-indipendent

Le librerie platform-dependent sono impiegate per realizzare altre librerie che sono indipendenti dalla piattaforma e che implementano la logica dell'applicazione. Tali librerie saranno presentate nelle sezioni seguenti; ora sono elencate alcune librerie di supporto create con il C standard (e quindi indipendente dalla piattaforma).

- **linked_list.c**: una semplice implementazione della struttura dati linkedList. Utilizzata per rappresentare una lista di notifiche associate ad un client.
- **settings_parser.c**: è un semplice parser utilizzato per leggere il file delle impostazioni per il server e per il client. La grammatica associata è la seguente:
  SettingList → Setting SettingList
  Setting → Word > Word **newline** | ϵ
  Word → [a-zA-Z0-9.:]
- **params_parser.c**: presenta una funzione che permette di parsare la command line per ottenere una lista di opzioni (identificate con il trattino iniziale "-") o di input generico. L'implementazione presentata permette di specificare una lista di opzioni (chiamate anche parametri nel codice) che richiedono obbligatoriamente un valore associato (che deve seguirlo immediatamente nel vettore *argv*); se nessun valore associato è identificato la funzione torna errore.
- **networking.c**: questa libreria implementa le funzionalità necessarie a creare ed utilizzare i socket per la comunicazione tra diversi host tramite una rete. È l'unico caso dove l'istruzione di preprocessing IFDEF è impiegata all'interno di funzioni per specificare del

codice dipendente dalla piattaforma. Le funzioni citate sono *load_sockets_library()* e la controparte per la rimozione della libreria dinamica. Queste funzioni su Windows richiamano WSAStartup() e WSACleanup mentre su linux ritornano subito (la stessa tecnica è utilizzata nel file di header per importare librerie specifiche). Il resto dell'implementazione è indipendente dalla piattaforma.
L'implementazione è stata pensata per sfruttare IPv6 quando possibile, mantenendo la compatibilità con IPv4 creando socket dual stack.

1.3 Compilazione e compatibilità

Il sitema è compatibile con il kernel 3.x di linux e superiore (se non sbaglio è quella la versione che supporta pienamente i pthread) e con Windows NT6 (Windows Vista e superiori, causa i socket dualstack).

La compilazione è effettuata impiegando il Makefile presente nella cartella principale, attraverso lo strumento *make* su linux e *nmake* su Windows. I targhet presenti nel Makefile sono server-linux, client-linux, server-win e client-win. L'output saranno i file server(.exe) e client(.exe). I compilatori specificati nel file sono *gcc* per linux e *cl* per Windows, ma possono essere facilmente sostituiti (sono stati parametrizzati).

2 Gestione della memoria

Per poter creare sul mapping strutture dati elaborate come alberi e linked list è stato necessario implementare un (semplice) sistema per gestire l'allocazione e deallocazione di blocchi di memoria all'interno dell'address space dedicato al mapping (mem_management.c). Non potendo memorizzare metadati al di fuori del mapping (visto che altri processi dovrebbero accedere a tali metadati), la strategia adottata è la più semplice possibile ed è di seguito descritta. Un generico blocco di memoria elegibile per l'allocazione è caratterizzato da un "blocco di intestazione" in cui sono memorizzati i seguenti metadati: dimensione del blocco (esclusi i byte dei metadati), un valore booleano *isFree* che indica se tale blocco è libero, ed infine *previous*, l'offset che porta al blocco che precede quello in considerazione. L'offset di un blocco x è l'indice del primo byte di x (il primo byte dei metadati) rispetto all'inizio dell'address space. All'interno del mapping, infatti, rimandi a zone di memoria sono effettuati tramite offset e non indirizzi; ciò permette di mantenere integrità ed una corretta intepretazione dei dati tra processi diversi.

Dopo aver creato il mapping, viene inizializzata la gestione dello stesso tramite una funzione apposita. Viene creato il primo unico blocco di memoria allocabile, grande quanto la dimensione del mapping (meno i metadati del blocco). Successivamente, quando si vuole allocare memoria, la funzione *pmm_malloc* "salta" di blocco in blocco fino ad arrivare al primo blocco di memoria libero e con sufficiente memoria (se disponibile, altrimenti torna errore). A quel punto una prima parte del blocco è allocata come richiesto mentre la seconda, in caso sia sufficientemente grande, è impiegata per creare un nuovo blocco libero (altirmenti viene allocato l'intero blocco trovato). La funzione di allocazione, quindi, non è molto efficiente visto che col crescere delle allocazioni deve effetturare più salti; è tuttavia l'unica soluzione ragionevolmente facile da implementare.

Quando un blocco di memoria deve essere liberato, sono considerati 4 blocchi: il blocco precedente x_{i-1}, il blocco da liberare x_i, il blocco successivo x_{i+1} ed il blocco successivo al successivo x_{i+2} (se esistono). Infatti se il blocco precedente (se esiste), o successivo (se esiste), od entrambi sono liberi, questi sono accorpati insieme al corrente in un unico blocco libero; se il

successivo è libero, al blocco x_{i+2} viene aggiornato il puntatore al blocco precedente (che prima puntava ad x_{i+1}) con il valore x_{i-1} se anche il blocco precedente era libero, x_i altrimenti. Se il blocco x_{i+1} non è libero, il suo puntatore al blocco precedente è aggiornato a x_{i-1} se tale blocco è libero, altrimenti rimane invariato (x_i). Questi casi sono tutti gestiti nel codice. I blocchi sono ridimensionati e i metadati aggiornati in accordo a questi cambiamenti. Notare che la *pmm_free* ha tempo costante di esecuzione.

3 Architettura generale

Viene ora presentata l'architettura generale del client e del server che formano il sistema. Il client è un programma molto semplice che riceve un comando da command line e lo trasmette tramite TCP. Qualora gli sia chiesto, esso rimane in ascolto di aggiornamenti creando un nuovo thread sul quale viene aperto un server UDP. Il server si compone delle seguenti parti:

- **Server TCP**: è la componente che si occupa di restare in ascolto per ricevere e gestire le richieste in arrivo dai client. Quando un client contatta il server, viene creato un thread per gestire la richiesta, apportare eventuali modifiche alle strutture dati, ed infine rispondere al client con un codice di ritorno. Questa componente è eseguita su un thread apposito.
- **Server Monitor**: è la componente che si occupa di inviare periodicamente aggiornamenti ai client registrati tramite UDP. Questa componente è eseguita su un thread apposito.
- **Daemon**: questa componente, eseguita anch'essa su un thread differente, si occupa di aggiornare le strutture dati interne al mapping, tra cui la rappresentazione del ramo del filesystem osservato. In altre parole controlla se si sono verificate modifiche al filesystem dall'ultima volta che è stato controllato, ed in tal caso aggiorna le *code di notifica* dei server interessati dalla modifica. Per mantenere la consistenza dei dati e per ragioni di ottimizzazione dell'implementazione, un solo daemon per host è attivo. Il primo server ad essere avviato attiva il daemon, mentre i successivi si registrano per ricevere le notifiche nelle loro code di notifica dal daemon già attivo. Il daemon opera sul mapping, quindi è in grado di parlare con tutti i processi server, sebbene appartenga solamente ad uno di questi (il daemon è un thread). Qualora il processo che possiede il daemon terminasse, il processo che per prima si accorgerà di ciò attiverà un nuovo daemon.

Il punto di entrata è il serverMonitor (server_monitor.c): in un primo momento viene creato (se esiste, aperto) il meccanismo di lock per accedere al mapping. Se il mapping non esiste allora il corrente processo è il primo ad essere stato avviato, il mapping viene popolato con le strutture dati necessarie, ed il daemon viene avviato. Successivamente viene avviato su un nuovo thread il server TCP ed il server monitor entra nel suo loop principale nel quale "dorme" x secondi per poi risvegliarsi e controllare se il daemon ha generato nuove notifiche. Se vi sono nuove notifiche, queste sono inviate ai client opportuni (ossia quelli registrati ad un path interessato dalla notifica). Il daemon esegue anch'esso un loop infinito dove scansiona ogni $x - 1$ secondi il filesystem. Il tempo di riposo del daemon è $x - 1$ in modo tale da aggiornare il mapping un po' più spesso di quanto il server monitor controlli per le notifiche, cosicché ad ogni controllo quest'ultimo abbia sempre gli ultimi aggiornamenti. Inoltre, per mandare il risveglio del server monitor fuori fase con quello del daemon ed evitare il più possibile situazioni nelle quali uno dei due è costretto ad aspettare l'altro, il server monitor attende 2 secondi prima di entrare nel loop principale.

4 Strutture dati

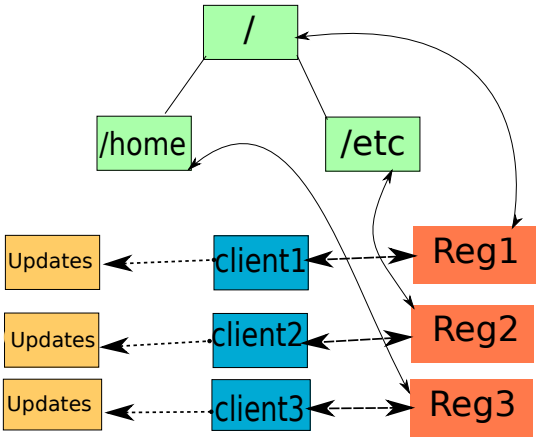
4.1 Mapping

Le strutture dati principali si trovano nel mapping. I server utilizzano il mapping come mezzo per ricevere notifiche dal daemon sui path che essi monitorano; inoltre il daemon stabilisce la frequenza di aggiornamento proprio leggendola dal mapping. Il mapping è strutturato, quindi, come un insieme di "tipi primitivi" (numero di server attivi, tempo di aggiornamento, che è il minimo tra tutti i tempi di aggiornamento richiesti dai server, id del server che possiede il daemon, ...), e due strutture dati complesse quali il *filesystemtree* e una linked list di *notifications bucket*. La prima è un albero che rappresenta i rami di filesystem monitorati dai server, aggiornato all'ultima scansione. Ogni nodo è un file o una cartella con i relativi attributi (permessi, ultima modifica, dimensione, nome). I notifications bucket sono dei "contenitori" creati dai server al momento della registrazione; ognuno di essi è identificato dall'ID del server e dal path; un server, quindi, ha tanti notifications bucket quanti sono i path che monitora. Quando il daemon rileva una modifica (confrontando il filesystem con la rappresentazione nel mapping) inserisce una notifica in ogni bucket interessato (confrontando il path del file modificato con quello associato al bucket). Al risveglio, ogni server monitor controlla i suoi notifications bucket per reperire le notifiche. L'accesso a queste strutture dati è controllato tramite un meccanismo di locking spiegato nella sezione apposita, più avanti.

4.2 ClientRegister

Il server deve poter memorizzare i client che si registrano e i path che i client vogliono monitorare. Bisogna fare attenzione a non confondere i path monitorati dal server e quelli monitorati dai client: nel codice i primi sono chiamati *server path* e i secondi *client path*. Un insieme di server path possono non essere monitorati da alcun client, ma non esiste un client path che non sia incluso in qualche server path.

I dati sui client sono memorizzati in una struttura dati che ha due rappresentazioni: una tramite albero dei client path, analogo all'albero dei server path nel mapping, ma contenente esclusivamente nodi rappresentati una registrazione (senza file e cartelle contenuti nel path, ma solo le cartelle che compongono il path) ed un'altra tramite linked list di *clientNode*, dove quest'ultima è una struttura che rappresenta un client registrato ad un qualche path. Sono due "facciate" che permettono l'accesso alle stesse *registrazioni*, come mostrato nella seguente figura. La prima rapppresentazione rende veloce il dispaccio di notifiche (ottenute dai buckets), mentre la seconda permette di aprire un solo socket per client ed inviare tutte le notifiche per quel client in una sola trasmissione.



Ogni qual volta un client invia un comando per ricevere aggiornamenti su di un path, viene aggiunto il path all'albero, se non esiste già, e viene creata una nuova registrazione e appesa all'ultimo nodo del path. Un puntatore alla stessa registrazione è aggiunto alla lista delle registrazioni presenti nella struttura del client. Se tale struttura non esiste (il client non aveva precedenti registrazioni), viene creata.

5 Funzionamento

Quando un server viene avviato deve registrare il suo path di avvio nel mapping. Viene chiamata dal server monitor la funzione *first_scan* presente nel file *mapping_structure.c* che si occupa di controllare se tale ramo è già presente nel mapping; in caso negativo, il ramo viene aggiunto ed è effettuata una prima scansione. Successivi aggiornamenti della struttura sono eseguiti dal daemon attraverso la funzione *update*. Nella funzione update, per ogni cambiamento rilevato viene aggiunta una notifica nei bucket dei server interessati dal cambiamento. Quando il server monitor andrà a controllare i propri notifications bucket copierà tutte le notifiche in memoria locale (rimuovendole dal mapping). Una volta che le notifiche sono in memoria locale, queste sono "smistate" tra i client conosciuti al server. Alcune notifiche non sono di interesse ad alcun client, altre sono aggiunte alle code di notifica di ogni client, una linked list chiamata "updates", in attesa di essere trasmesse. Lo smistamento avviene percorrendo dalla radice l'albero delle registrazioni dei client, seguendo il percorso indicato dal path (assoluto) del file associato alla notifica. Una volta completato lo smistamento, per ogni client viene creato un client UDP che trasmette le notifiche nel formato richiesto.

Tutti i casi particolari, quali:

- Registrazione da parte di un client ad una cartella di cui già riceve notifiche;
- Registrazione da parte di un client ad una "superfolder" di una cartella già monitorata;
- Aggiunta di un path al server che è già incluso in altri server path;
- Merge di più subpath (client e server) in path più generali richiesti successivamente da un client;
- Cancellazione dal filesystem di una cartella monitorata da un server (e magari registrata da un client);

sono gestiti correttamente.

6 Aggiunte al protocollo

Il protocollo presenta il comando aggiuntivo ADDPNR che permette al client di aggiungere un path alla lista di *server path* monitorate e registrarsi a tale path in maniera non ricorsiva. Questa funzionalità è implementata sul server come un ADDP seguita da un INNR.

Sono stati aggiunti i seguenti codici di ritorno:

- 300: un errore interno al server durante la gestione del comando inviato dal client ha causato la terminazione del processo server.
- 201: si tenta di registrarsi ad un path già monitorato in maniera ricorsiva. Il codice è a titolo informativo, lo stato del client sul server non cambia.
- 202: la registrazione effettuata sostituisce una o più registrazioni a subfolder di quella appena richiesta (in maniera ricorsiva).

7 Sincronizzazione dei thread e processi e terminazione del server

Affiché il server possa condividere correttamente dati tra diversi thread e processi, vi è necessità di diversi meccanismi di lock. Sono stati predisposti due lock per sincronizzare l'accesso al file mapping:

- "syncmapping" è il lock (semaforo su linux, named mutex su Windows) che sincronizza i processi e i thread per quanto riguarda l'accesso effettivo al mapping. Questo lock non può essere eliminato in quanto, come già spiegato nell'introduzione, è possibile che un nuovo server sia già bloccato su di esso (e pensi di non essere il primo server avviato).
- "activateLock" (active deriva dal nome di una variabile che indicava lo stato del processo, ora non più presente) è un secondo lock che protegge l'accesso al primo lock all'interno dello stesso processo. Prima di acquisire il syncmapping lock, un thread deve acquisire prima l'ativateLock; quest'ultimo sarà rilasciato da un thread solo quando lo stesso thread ha già rilasciato il syncmapping lock.

Per spiegare la necessità dell'activateLock, bisogna considerare il modo in cui è gestita la terminazione del server. Alla ricezione del segnale CTRL+C (sigint), viene creato un nuovo thread che esegue la routine di terminazione. Questa routine si occupa di rimuovere i bucket del server dalla linked list di notifications bucket nel mapping, ed eventualmente rimuovere alcuni rami del filesystemtree. È chiaro che per fare ciò debba anch'essa acquisire i due lock sopracitati per far si che il mapping rimanga integro (per eventuali altri server, se presenti). La routine, lasciata la sezione critica, chiamerebbe EXIT(0). Tuttavia, tra le due istruzioni, un altro thread (serverMonitor o daemon) potrebbe avere il tempo di entrare (ma non lasciare) la sezione critica prima che il processo sia terminato, lasciando il lock bloccato. Per prevenire ciò, quindi, introduciamo il mutex activeLock che vive solo nel processo corrente; quando la routine lascia la sezione critica definita da syncmapping nessun altro thread dello stesso processo può accedere al mapping perché detiene il possesso di activeLock. La routine termina senza rilasciare activeLock, lasciando syncmapping in maniera corretta.

Un ulteriore mutex, threadLock, è necessario per proteggere la struttura ClientRegister, utilizzata dai thread server monitor e server TCP.