MyDistributedFileSystem

Federico Scozzafava (1539761)

Progetto per il corso di Programmazione di Sistema a/a 2014/2015

Sviluppo di un semplice file system distribuito multi piattaforma.

Il software implementa un protocollo di comunicazione su due canali in stile FTP (connessione di controllo e connessione dati). Ogni comando è identificato da una stringa alfanumerica ed eventuali parametri, la risposta del server è caratterizzata da un codice numerico e da una descrizione testuale del messaggio. Le varie parti del comando sono separate dal carattere '\t'.

Comandi definiti dal protocollo

Client -> Server:

- `OPEN $path $mode $block_size`
  - Apre il file remoto locato in `$path` con modalità di accesso `$mode` e dimensione blocco `$block_size`
- `READ $block_num`
  - Chiede al server l'invio del blocco `$block_num`
- `WRTE $block_num $size`
  - Informa il server che il client intende inviare il blocco `$block_num` del file precedentemente aperto, inoltre, `$size` contiene il numero di byte che il client intende inviare (necessario nel caso in cui si intenda inviare l'ultimo blocco di un file non completo)
- `CLSE`
  - Chiude il file remoto e dealloca le risorse
- `HTBT`
  - Informa il Server, come risposta a un messaggio di "*heart beating*", che il client è attivo

Server -> Client:

- `DFS_OK_CODE: 800`
  - Generico messaggio di esito positivo
- `DFS_OPEN_CODE: 801 $port $size $blocks`
  - Il server annuncia al client l'esito positivo della procedura di apertura del file remoto
  - `$port` indica il numero di porta sulla quale il server è in ascolto in attesa di stabilire una connessione dati
  - `$size` è la dimensione del file remoto
  - `$block` è il numero di blocchi del file
- `DFS_SEND_CODE: 802 $block_num`
  - Il server sta inviando il blocco `$block_num`
- `DFS_RECV_CODE: 803 $block_num`
  - Il server è pronto a ricevere il blocco `$block_num`

- `DFS_CLSE_CODE: 804`
  - Il server ha deallocato le risorse e chiuso il file remoto
- `DFS_INVD_CODE: 805 $size`
  - Il server invia al client un messaggio di invalidazione con la nuova dimensione del file nel caso in cui il file fosse stato esteso
- `DFS_HTBT_CODE: 806`
  - Il server invia un messaggio "*heart beating*" al client per verificarne lo stato
- `DFS_ERRN_CODE: 888 $errno`
  - Il server informa il client che si è verificato un errore

Descrizione di alto livello del sistema

Il server, di default, è in ascolto sulla porta `6020`, è possibile cambiare la porta in ascolto tramite file di configurazione o parametro **`-p xxxx`** da linea di comando. È inoltre possibile configurare i seguenti parametri tramite file di configurazione:

- `port`
  - Porta in ascolto
- `path`
  - Percorso base, il file system distribuito si estende dal sotto albero di questa directory
- `n_process`
  - Numero di processi/thread che il server può gestire contemporaneamente
- `threaded`
  - `1` attiva la modalità multi-thread, `0` attiva la modalità multi-processo
- `daemon`
  - `1` lancia il server in modalità daemon (solo unix)

Lato client è possibile impostare i seguenti parametri tramite file di configurazione:

- `port`
  - Porta del servizio a cui connettersi
- `block_size`
  - Dimensione del blocco

Il client apre una nuova connessione e richiede l'accesso ad un determinato file con una particolare modalità di accesso. Il server, qualora in grado di soddisfare la richiesta, apre un socket in ascolto e risponde al client chiedendo di stabilire una connessione dati. Una volta stabilite entrambi le connessioni, il client può accedere ed effettuare operazioni sul file remoto.
Il client mantiene una cache locale sia in lettura che in scrittura; prima di poter modificare un blocco è necessario scaricare la copia originale dal server, le modifiche vengono inviate solamente al momento della chiusura del file.
Il server mantiene una lista di client attualmente connessi dove ad ogni client sono associate informazioni riguardo il file aperto e la modalità di accesso. A seconda che il client detenga o meno un lock esclusivo per il file, il server invia periodicamente un messaggio di "*heart beating*" al client per verificarne l'attività, inoltre quando un client apre un file in scrittura o richiede un lock esclusivo, il server notifica (ad ogni modifica del file) tutti i client che hanno lo stesso file aperto in lettura invalidando la loro cache locale.
Il client, qualora si verifichino le condizioni sopra descritte, avvia un thread di supporto per ogni file remoto aperto responsabile della gestione dei messaggi di "*heart beating*". Prima di ogni lettura il client controlla la presenza di eventuali messaggi di invalidazione.

Struttura e implementazione del sistema

Il sistema si divide in 3 parti:

1. `myDFSServer`: contiene il software necessario all'avvio del server
2. `myDFSClient`: la libreria tramite che mette a disposizione i metodi di comunicazione del protocollo lato client
3. `myDFSCommons`: libreria di appoggio che contiene tutte le funzioni di utilità e le macro in comune tra i progetti client e server

Note generali di implementazione

Per la sincronizzazione sono stati usati esclusivamente semafori: con nome lato server, anonimi lato client. Per motivi di portabilità, i semafori anonimi (header `<semaphore.h>`), non implementati in OS X, sotto unix sono realizzati tramite semafori con nome casuale e univoco automaticamente scollegati (unlinked) al momento della creazione.
Le macro adibite alle modalità di accesso sono definite nell'header `<mydfs_commons.h>`; per garantire portabilità su diversi sistemi sono state definite specifiche macro e relativi metodi di conversione da macro di sistema `O_XXXX` a macro con prefisso `MDFS_O`.
La tokenizzazione delle stringhe è effettuata con il metodo rientrante `strtok_r()`, puttosto che con la versione classica `strtok()`, per supportare operazioni in modalità multi-thread.
Le conversioni da intero a stringa sono state effettuate impiegando il metodo `strtol()/strtoul()`, più sicuro del metodo deprecato `atoi()`.
Le operazioni di connessione, lettura e scrittura su `socket` sono non bloccanti: l'header `<poll_helper.h>`, fornito dalla libreria `myDFSCommons`, fornisce metodi di utilità per la gestione delle chiamate alla primitiva `poll()`, preferita alla primitiva `select()` in quanto in grado di supportare valori di file descriptor maggiori di `1024`. L'obiettivo di queste scelte è quello di evitare che il server effettui attività di polling e ,al tempo stesso, sfruttare gli intervalli di timeout per effettuare operazioni di cleaning up.

myDFSServer

Le strutture principali lato server sono:

- `mydfs`: struttura locale al singolo processo/thread, mantiene i dati relativi al client preso in carico
- `shared_struct`: mantiene i dati relativi ad un client nella lista dei file aperti condivisa tra i processi/thread

Il server installa due signal handler tramite la primitiva `sigaction()` con opzione `SA_RESTART` per automatizzare, ove possibile, il riavvio delle system call interrotte da segnali.

- `invalidate_handler`: ogni processo/thread è responsabile dell'invio di messaggi di invalidazione verso il proprio client, nonostante ciò le invalidazioni sono attivate da altri client che aprono/chiudono file in scrittura o richiedono un lock esclusivo. Attraverso la struttura `shared_struct`, un processo può notificare l'invalidazione al processo incaricato della gestione dell'altro client inviando un segnale(nella mo dalità multi processo si invia un segnale al processo, nel caso di modalità multi thread si invia un segnale al thread utilizzando la opportuna primitiva).
- `sigchld_handler`: questo handler è responsabile di catturare il segnale `SIGINT` e `SIGCHLD`. Nel primo caso al fine di liberare le risorse in maniera opportuna e terminare il

processo padre, nel secondo caso di catturare i processi figli terminati (nella modalità multi processo) evitando di lasciare processi nello stato zombie inattivi nel sistema

Il Server apre il socket per l'ascolto sulla porta predefinita e inizializza i semafori e il segmento di memoria condivisa necessario a mantenere la lista dei client attualmente connessi e le risorse da essi allocate.

Semafori attivi lato server:

- `backlog_semaphore`: inizializzato al valore massimo di connessioni gestibile in parallelo (file configurazione), il semaforo blocca il processo padre al raggiungimento del limite prefissato
- `main_semaphore`: semaforo binario che regola l'attivazione di processi/thread figli. Necessario per assicurarsi che il processo padre aspetti la completa inizializzazione del processo/thread figlio (la copia dei parametri dal processo padre e creazione delle strutture condivise)
- `shared_semaphore`: semaforo binario che regola l'accesso alla memoria e strutture condivise
- Un semaforo per ogni processo/thread generato è salvato nella struttura condivisa di ogni client ai fini di creare una barriera al momento della creazione del processo e durante le fasi di invio invalidazioni (per sicurezza i segnali vengano comunque momentaneamente disabilitati prima di entrare nella sezione critica durante una richiesta da parte del client)

La memoria condivisa è così organizzata:

La prima sezione è riservata a un array di interi di dimensione pari al numero massimo di processi gestibili in parallelo, per ogni processo `i`, `array[i]` conterrà il valore `1` se il client è attivo in lettura, il valore `2` se il client è attivo in scrittura/lock esclusivo, `0` altrimenti. Seguono nel segmento segmenti riservati ad altrettante strutture `shared_struct` da considerarsi valide solo nel caso in cui la corrispettiva cella nell'array contenga un valore maggiore di zero. L'accesso alla struttura condivisa è regolata dal semaforo `shared_semaphore` e da opportuni metodi.

Dopo aver accettato la connessione da parte di un client, a seconda delle impostazioni, il server crea un nuovo processo con la primitiva `fork()` o un nuovo thread. In questo caso il thread è subito scollegato dal processo principale con la chiamata al metodo `pthread_detach()` dal momento che non vogliamo attendere la sua terminazione.

Tutti i file descriptor necessari sono automaticamente ereditati dai processi figli. Il server attende, come descritto in precedenza, la inizializzazione del processo figlio e si rimane in attesa di nuove connessioni.

Il processo figlio appena creato attende la copia delle informazioni nella struttura condivisa da parte del processo genitore, le copia nella struttura locale ed entra nel loop di elaborazione.

Il lock esclusivo è gestito a livello server grazie ad un campo nella struttura condivisa che indica se un client ha acquisito il lock sul determinato file.

myDFSClient

La struttura principale della libreria è la struct `MyDFSId`, questa contiene i riferimenti a tutte le risorse (cache, semafori, socket) necessarie al client.

Il client, come il server, opera in maniera non bloccante, nel caso in cui ci sia necessario che il client risponda a messaggi di "*heart beating*", il client avvia un thread per la gestione di questi ultimi. Il thread non è scollegato dal processo primario perché al momento della chiusura della connessione vogliamo assicurarci che questo termini. L'accesso al socket di controllo è regolato tramite un semaforo anonimo.

Ad ogni lettura del socket di controllo, che sia da parte del thread secondario o meno, il client gestisce tutte le richieste di beating e invalidazione (anche se ce ne sono molte accodate); il metodo che si occupa della ricezione dei messaggi è implementato con un ciclo while onde evitare di incorrere in problemi di stack-overflow.

Il meccanismo di caching:

La cache è implementata tramite un `file mapping anonimo` (il motivo di questa scelta è quello di evitare di creare file nel file system) indicizzato a blocchi, lo stato di ogni blocco è definito in un array di interi ausiliario, nel quale ogni cella è valorizzata con `0` se il blocco non è presente in cache, `1` se il blocco è presente in cache, `2` se il blocco è presente in cache ed è stato modificato.

Qualora il file venisse esteso, e sia necessario ridimensionare il file mapping della cache, per motivi di portabilità, non viene usata la più comoda primitiva `mremap()` (non implementata in OS X), bensì il ogni volta viene creato un nuovo file mapping di dimensione raddoppiata, copiato il contenuto dell'originale e sostituito.

Il thread dedicato alla gestione dei messaggi di "*heart beating*" è attivo ad intervalli di `1` secondo, durante i quali quest'ultimo prova ad acquisire il lock del semaforo condiviso con il processo principale, tramite la primitiva `sem_trywait()`, per permette al thread principale una comunicazione più fluida con il server. Una volta acquisito il lock si effettua una `peek()` sul socket di controllo per verificare la presenza di messaggi di "*heart beating*", solo in quel caso il thread gestisce la richiesta.

myDFSCommons

Di seguito la descrizione delle funionalità fornite dagli header di questa libreria:

- `configuration_mangaer.h`
  - Definisce metodi per il caricamento ed il parsing dei file di configurazione server e client
- `mydfs_commons.h`
  - Definisce macro per le modalità di accesso ai file e funzioni di utilità in comune tra client e server (funzioni socket, conversione macro, funzione close() con controllo dell'errore integrato)
- `poll_helper.h`
  - Definisce metodi wrapper con controllo dell'errore integrato intorno alla primitiva `poll()`
- `read_utilities.h`
  - Definisce metodi per la gestione dell'`I/O` non bloccante su socket e file (readline non bloccante e così via)
- `sem_utilites.h`
  - Definisce metodi con controllo dell'errore integrato per la gestione e la creazione di semafori con nome ed anonimi

WIN_32 API porting

Panoramica generale

Il porting su piattaforma Windows è stato eseguito utilizzando il pacchetto **`MINGW`**, sfruttando, ove possibile, il sottosistema `POSIX` presente in Windows (alcune delle primitive open, read, write sono rimaste invariate). Le primitive riguardanti i semafori sono state rimpiazzate con le relative

primitive WIN32. La primitiva poll() è stata rimpiazzata con select() e la memoria condivisa (non portabile) è stata rimpiazzata con un file mapping con nome non mappato a file. Processi e thread figli sono creati in modalità SUSPENDED, evitando così di dover ricorrere a barriere di semafori durante l'inizializzazione.

Ereditarietà

Il passaggio dei descrittori SOCKET e HANDLE necessari ai processi figli avviene tramite le strutture condivise e il meccanismo di ereditarietà fornito dalli API WIN_32,nessun meccanismo di duplicazione HANDLE / re indirizzamento standard HANLE è risultato necessario.

Porting dei segnali

Sotto Windows i segnali sono stati portati con il seguente metodo:
Ogni processo/thread associato ad un client crea un thread dedicato alla gestione dei "*segnali emulati*". Il thread crea un evento con nome (il nome è legato all'id del processo chiamante) e si mette in attesa con la primitiva WAITFORSINGLEOBJECT(). Quando si ha il trigger dell'evento il thread si attiva, richiede il lock del semaforo specifico del processo, invia il messaggio di invalidazione al client e resetta l'evento. Il processo che vuole inviare un segnale invoca il metodo kill() definito in <mydfs_commons.h> fornendo come parametro il l'id assegnato al processo da segnalare (0-n_process), il metodo apre l'evento, invoca una SetEvent() e chiude l'evento. Alla fine del ciclo di vita del processo, si segnala l'evento indicando al thread di terminare e quindi si attende il thread, l'evento è definitivamente distrutto.