Relazione Programmazione di sistema

Federico Ponzi

Introduzione

Per la realizzazione del progetto, si è scelto di optare per due librerie distinte: una per sistemi Windows, e una per sistemi *nix.

Di seguito sono elencati i vari moduli nell' ordine secondo cui sono stati progettati, in modo da capire meglio quali e perchè sono state fatte certe scelte.

Server: OPEN

Prima di tutto abbiamo avuto bisogno di creare un semplice programma che scambiasse dati usando un'architettura Client/Server usando le Socket.
Per come è definita questa architettura, abbiamo necessariamente avuto bisogno di iniziare dal Server (senza la quale è ovviamente impossibile testare un client).
Il server viene avviato in ascolto su una porta P. Una volta connesso, un client invierà il comando:

```
OPEN nomefile modo
```

Il modo può essere uno fra quelli stabiliti dalla open di *nix.
A questo punto, il server istanzierá la struttura dati `OpenedFile`, una tabella di file aperti in un dato istante, che si trova su una zona di memoria condivisa fra processi.

Una volta creata l'associazione, il server passa a creare una connessione secondaria con il client. Questa verrà usata per mandare messaggi di invalidazione, o utilizzata per l'heartbeating in caso di open in modalità scrittura.

Server: Close

Una volta stabilità le connessioni, si è reso necessario un modo per chiuderle.
Implementare la CLOSE a questo punto è stato abbastanza facile: liberata la memoria occupata, vengono chiuse le connessioni una volta ricevuto il messaggio:

```
CLOS
```

Dal client.

Client: Open e Close

Il client ha una struttura dati `MyDFS` (come richiesto dalle specifiche) e implementa la open e la Close usando i prototipi stabiliti dalle specifiche.

La open avvia la connessione al server, crea una socket server su una porta casuale (connessione di controllo) che invia al server, e stabilite le connessioni ritorna la struttura dati appena costruita.
A questo punto è stato possibile completare la open con i suoi rispettivi codici di errore.
La Close, come la speculare sul server, libera la memoria occupata e chiude le connessioni.

Client: Read e Cache

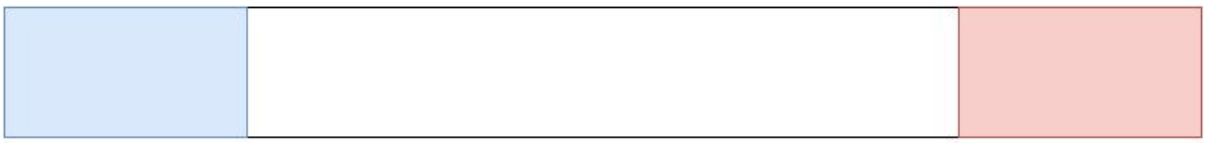
Per implementare la Read, si è resa necessaria la realizzazione del sistema di caching.
Tenendo a mente la richiesta di portabilità, si è optato per la soluzione più semplice: creare un file temporaneo, dove effettuare tutte le operazioni di lettura e scrittura.
Al momento della open, viene creato un file temporaneo, aperto, e il puntatore associato viene salvato all'interno di `MyDFSId`.
Abbiamo avuto bisogno anche di tenere traccia delle operazioni di lettura (e, successivamente, di scrittura) effettuate.
Prima di una richiesta di read, mydfs_read chiede alla cache di controllare la lista di operazioni di lettura effettuate per controllare se la richiesta dell'utente, basandosi sulla attuale posizione del puntatore al file e la dimensione della lettura, avrà successo.
In caso l'utente richieda di leggere parte del file non presente in cache (`MISS`) la cache ritorna la posizione (e la dimensione) del primo buco. La read penserà a richiedere il pezzo mancante al server:

```
REA 0
```

La dimensione da leggere è sempre la stessa, definita dalle specifiche,quindi non abbiamo bisogno di specificarla.

Il server dovrà semplicemente leggere il file e mandarlo al client partendo dalla posizione richiesta.
`Mydfs_read` terrà traccia della operazione salvandola nella lista di operazioni di lettura, è continuerá a riempire gli eventuali buchi nel file finché non sarà possibile soddisfare la richiesta dell'utente.
Alla fine, verrà effettuata una semplice read al file (che sposterà anche la posizione all'interno del FILE associato a `MyDFSId`).

Esempio:



La parte azzurra e' stata scritta a seguito di una operazione di mydfs_read, mentre la parte rosa dopo una operazione di mydfs_write.
Possiamo avere una situazione del genere all' interno del file, dopo che l' utente ha richiesto la lettura della parte iniziale del file ed ha scritto partendo dalla file del file.

Client: Write

Per la write è stato necessario modificare leggermente la cache: le letture, da 64k, **non** devono infatti sovrascrivere le parti eventualmente scritte dall'utente.
La write ha effetto nel server al momento della `Close`:

```
CLO n
```

Indica che l'utente ha effettuato n operazioni di scrittura.
Il client itera nella lista di operazioni di scrittura, leggendo e inviando al server il contenuto del file nella cache, che provvederà poi a scriverlo nel file.

Server: Invalidazione

Quando il client che ha effettuato l'apertura in write ha concluso le sue modifiche, il client che ha aperto in read lo stesso file deve essere notificato della cosa: la lista delle sue read viene svuotata, e per la mydfs_read/cache il file risulterà vuoto (e passerà quindi a riempire i buchi).
In modalità multiprocesso i server non condividono i socket descriptor, per questo il processo incaricato di servire il client/lettore deve essere notificato e deve mandare il messaggio di invalidazione.

Su *nix questa funzione è stata delegata ai segnali. Su windows sono state usate le pipe.

Server: Heartbeating

Quando viene aperto un file in scrittura, il server avvia un thread secondario che ogni n secondi (definiti nel file di configurazione) invia un messaggio al client e si aspetta una risposta. In caso non ci sia, questo thread provvederà ad interromepre la comunicazione e rimuovere l' associazione all' interno della tabella.

Client: Heartbeating

Per quanto riguarda il client, viene creato anche quì un thread secondario per la comunicazione sulla socket di controllo. Questo thread, viene spawnato in ogni caso: viene infatti incaricato anche della gestione dei messaggi di invalidazione.