jSync

Progetto Laboratorio di Sistemi Operativi

A.A. 2015-2016

Gruppo: Angelina

Componenti:
- Pietro Tamburrini 590603 pietro.tamburrini@studio.unibo.it
- Nicolò Faedi 694919 nicolo.faedi@studio.unibo.it
- Massimo-Maria Barbato 732766 massimomaria.barbato@studio.unibo.it

Introduzione

Con jSync un utente può sincronizzare una qualsiasi cartella del suo computer con il server e viceversa.

Una particolarità dell'implementazione da noi realizzata è che quando è effettuata un'operazione di Push di una repository vengono trasferiti solo i file non presenti sul server oppure che sul server sono meno aggiornati. La stessa procedura avviene quasi specularmente quando è eseguita un'operazione di Pull.

Istruzioni per l'avvio della Demo e del Programma
Demo
I clients sono completamente automatizzati.
Es. Lanciando da terminale il file "Client1.sh" viene avviato il servizio "ClientDemo.ol" passando automaticamente come nickname il filename 'Client1'. L'esecuzione prosegue eseguendo nella CLI le istruzioni presenti nel file "Client1_Instructions" alternate da uno sleep di 4 secondi, definito dalla costante "Timer_wait" in ClientDemo.ol

Le istruzioni eseguite sono come da specifica del 'report_template' sezione ###Demo, utilizzando la cartella di riferimento cartella "RepoTest" contenente alcuni file e sottocartelle. Durante l'esecuzione delle istruzioni associate a Client2 viene creato il file "new_file.txt" e successivamente effettua la push di RepoTest.

Il server viene avviato @ socket://localhost:8000
Nelle sezioni critiche agisce uno sleep di 6 secondi per aumentare la probabilità che vi siano situazioni di concorrenza.

Avvio Demo (2 Clients - 1 Server)
1. Aggiungere tutto il contenuto della cartella 'Servizi Java' alla propria cartella (/usr/lib/jolie/javaServices)
2. Avviare da terminale 'jolie ../DEMO/Server.ol'
3. Avviare in due terminali 'jolie ../DEMO/Client1.sh' e 'jolie ../DEMO/Client2.sh'
4. Le istruzioni e gli step saranno stampati a video

Progetto
Non sono presenti sleep né sul server né sul client.

Avvio Progetto (n Clients - n Servers)
1. Aggiungere tutto il contenuto della cartella 'Servizi Java' alla propria cartella (/usr/lib/jolie/javaServices)
2. Avviare da terminale 'jolie .../PROGETTO/Server.ol' quanti server si desiderano modificando ogni volta le costanti S_NAME e S_ADDRESS in 'Server.ol'
3. Avviare da terminale quanti clients si desiderano. All'avvio verrà richiesto il proprio nickname.

Implementazione

Struttura del Progetto
Il progetto è diviso in client e server.
Sia client sia server fanno l'embedding del servizio 'FileManager.ol' e del JavaService 'example.info'.
Entrambi implementano le interfacce: **LocalInterface** e **ClientInterface** definite all'interno del file 'Interface.iol'.
In LocalInterface sono specificate le operazioni che sia client che server effettuano sulla propria memoria locale.
In ClientInterface invece sono specificate le operazioni di **comunicazione** tra client e server.

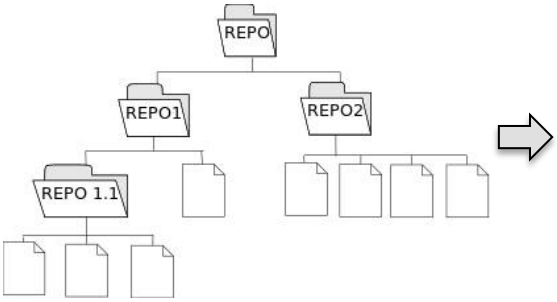
```
interface ClientInterface {
    RequestResponse:    addServer( RegServer )( bool ),
                        getServerRepoList( void )( Struttura ),
                        pushRequest( Repo )( FileRequest ),
                        pullRequest ( string )( Repo ),
                        pull ( FileRequest )( RawList )

    OneWay:             addRepository( RegRepo ),
                        push( RawList ),
                        delete( string )
}

interface LocalInterface {
    RequestResponse:    readXml( string )( Struttura ),
                        updateXml( Struttura )( void ),
                        fileToValue( Repo )( Repo ),
                        getLastModString( string )( string ),
                        setLastMod( SetVersion )( string)
}
```

Per gestire la persistenza delle repositories e dei server registrati abbiamo scelto l'impiego di file XML, appoggiandoci per la loro gestione al servizio XMLUtils
Il servizio **fileManager** è responsabile della lettura e della scrittura del file XML sia per il client sia per il server ed inoltre fa da collegamento per il servizio fileToValue facendone l'embedding.

FileToValue è un servizio che data una cartella da visitare, restituisce una struttura ad albero che rappresenta le sottocartelle e i file in essa contenuti.
La creazione dell'albero avviene mediante una visita ricorsiva delle varie sottocartelle. Quando, durante l'analisi, viene trovato un file, alla sua rappresentazione nella struttura viene associata la versione.




```
Value = Repo : java.lang.String
        .relativePath[0] = Repo : java.lang.String
        .repo[0] = Repo1 : java.lang.String
                .file[0] = Repo1File1.txt : java.lang.String
                        .relativePath[0] = Repo/Repo1/Repo1File1.txt : java.lang.String
                        .version[0] = 1435766191000 : java.lang.Long
                .relativePath[0] = Repo/Repo1 : java.lang.String
                .repo[0] = Repo1.1 : java.lang.String
                        .file[0] = Repo1.1File1.txt : java.lang.String
                                .relativePath[0] = Repo/Repo1/Repo1.1/Repo1.1File1.txt : java.lang.String
                                .version[0] = 1435766145000 : java.lang.Long
                        .file[1] = Repo1.1File2.txt : java.lang.String
                                .relativePath[0] = Repo/Repo1/Repo1.1/Repo1.1File2.txt : java.lang.String
                                .version[0] = 1435766153000 : java.lang.Long
                        .file[2] = Repo1.1File3.txt : java.lang.String
                                .relativePath[0] = Repo/Repo1/Repo1.1/Repo1.1File3.txt : java.lang.String
                                .version[0] = 1435766160000 : java.lang.Long
                        .relativePath[0] = Repo/Repo1/Repo1.1 : java.lang.String
        .repo[1] = Repo2 : java.lang.String
                .file[0] = Repo2File1.txt : java.lang.String
                        .relativePath[0] = Repo/Repo2/Repo2File1.txt : java.lang.String
                        .version[0] = 1435766201000 : java.lang.Long
                .file[1] = Repo2File2.txt : java.lang.String
                        .relativePath[0] = Repo/Repo2/Repo2File2.txt : java.lang.String
                        .version[0] = 1435766207000 : java.lang.Long
                .file[2] = Repo2File3.txt : java.lang.String
                        .relativePath[0] = Repo/Repo2/Repo2File3.txt : java.lang.String
                        .version[0] = 1435766215000 : java.lang.Long
                .file[3] = Repo2File4.txt : java.lang.String
                        .relativePath[0] = Repo/Repo2/Repo2File4.txt : java.lang.String
                        .version[0] = 1435766221000 : java.lang.Long
                .relativePath[0] = Repo/Repo2 : java.lang.String
```

Abbiamo deciso di far fare da tramite a FileManager per evitare ad altri servizi di avere ulteriori outputPort locali per contattare FileToValue.

Per definire la versione di un file, impieghiamo un **JavaService** che si appoggia alle funzioni *lastModified* e *setLastModified* della classe java.io.file.

VERSIONING

Per definire la versione di un file abbiamo deciso di impiegare l'ultima modifica del file riportata nei suoi metadati, perché ci sembrava più preciso e sicuro e ci permette di avere la versione di ogni file.

Facendo ciò abbiamo riscontrato un grosso problema: quando il file viene trasformato in Raw e scritto, ciò viene considerato una modifica (creazione del file) e quindi il parametro lastModified viene aggiornato.

Per ovviare a questo problema abbiamo usato il metodo setLastModified di java.io.file. Una volta che il file viene trasferito e scritto, il suo lastModified viene reso coerente con il lastModified del file trasferito.

```
clientVersion.path = push_rawList.file[i].filename;
clientVersion.version = push_rawList.file[i].version;
undef(push_rawList.file[i].version);

writeFile@File(push_rawList.file[i])();
setLastMod@JavaService(clientVersion)(r)
```

Di contro questa soluzione presenta un problema di logica. L'utente ha una copia di un file non aggiornata, ma che ha modificato recentemente. Quando effettua un'operazione di Push della repository contenente il file sovrascrive il file del server facendo perdere tutte le modifiche precedenti alla sua.

CLIENT E SERVER

Il file 'client.ol' contiene sia la CLI sia tutta la logica del client.
Abbiamo implementato tutto in un solo file perché file separati non ci permettevano di richiedere ulteriori input da tastiera all'utente.
Come suggerito sul gruppo, l'impiego di differenti variabili per registrare differenti input da tastiera funziona egregiamente quando la registrazione al servizio registerForInput avviene nello stesso main, ma quando la registrazione è eseguita in un servizio e l'input da tastiera è registrato in un servizio richiamato la soluzione delle variabili differenti non funziona.

Gli elenchi delle repository e dei server registrati dal client sono mantenuti nella struttura global.root, rispettivamente nei nodi *global.root.repo* per le repository e *global.root.server* per i server.
Ogni elemento di *global.root.server* è di tipo *regServer*. Tale tipo di elemento contiene il nome assegnato localmente al server (parametro *.name*) e l'indirizzo per raggiungerlo (parametro *.adress*).
Ogni elemento di *global.root.repo* è di tipo regRepo. Tale tipo contiene il nome (parametro *.name*) e il path assoluto (parametro *.adress*) della repository nel sistema del client. Inoltre per associare

una repository al server di riferimento, il tipo *regRepo* contiene il nome del server (parametro . *.serverName*) e l'indirizzo (*.serverAdress*).

```
type Struttura: void {
    .server[0,*]: RegServer
    .repo[0,*]: RegRepo
}
```

```
type RegServer: void {
    .name: string
    .address: string
}
```

```
type RegRepo: void {
    .name: string
    .path: string
    .serverName: string
    .serverAddress: string

    .sDB?: SemaphoreRequest
    .sMutex?: SemaphoreRequest
    .readerCount?: int
}
```

All'**avvio del client** è richiesto all'utente il proprio nickname e, tramite il servizio fileManager, dalle informazioni contenute nel file XML associato a tale nickname, ne vengono caricati in *global.root* i server e le repository registrati.
Successivamente il client attende l'input dell'utente.
Il **define** di eseguiComando permette di rispondere con l'operazione richiesta dall'utente.

Anche all'**avvio del server** viene caricata tramite fileManager da XML la lista di *regRepo* e salvata in *global.root.repo*.
Durante l'inizializzazione dei semafori del server, richiamata nell'init, sono create 2 variabili di tipo *semaphoreRequest*:
- *sRootRequest* semaforo sulla lista di repo registrate
- *sRootMutex* semaforo sulla variabile *global.rootReaderCount*

Inoltre ogni variabile *regRepo* del server contiene anche una variabile *readerCount* che conserva il numero di lettori su quella repository e 2 variabili di tipo *semaphoreRequest*:
- *sDB* semaforo sulla repository
- *sMutex* semaforo sulla variabile *readerCount*

AddRepository

Un utente può aggiungere una qualsiasi cartella del proprio computer inserendone nei parametri del comando il path
- assoluto quando la repository che si vuol aggiungere non è contenuta in quella di esecuzione del programma
- relativo quando la repository che si vuol aggiungere è contenuta nella cartella di esecuzione del programma

Se la repository specificata non è fisicamente presente su server o su client viene creata una cartella vuota per contenere i file scambiati durante le operazioni di Push e di Pull.
La *regRepo* associata viene aggiunta alle strutture (*global.root.repo*) del client e del server.

Push

Per l’operazione di Push abbiamo deciso di implementare un'ottimizzazione che permette di usare meno banda a discapito del costo computazionale.

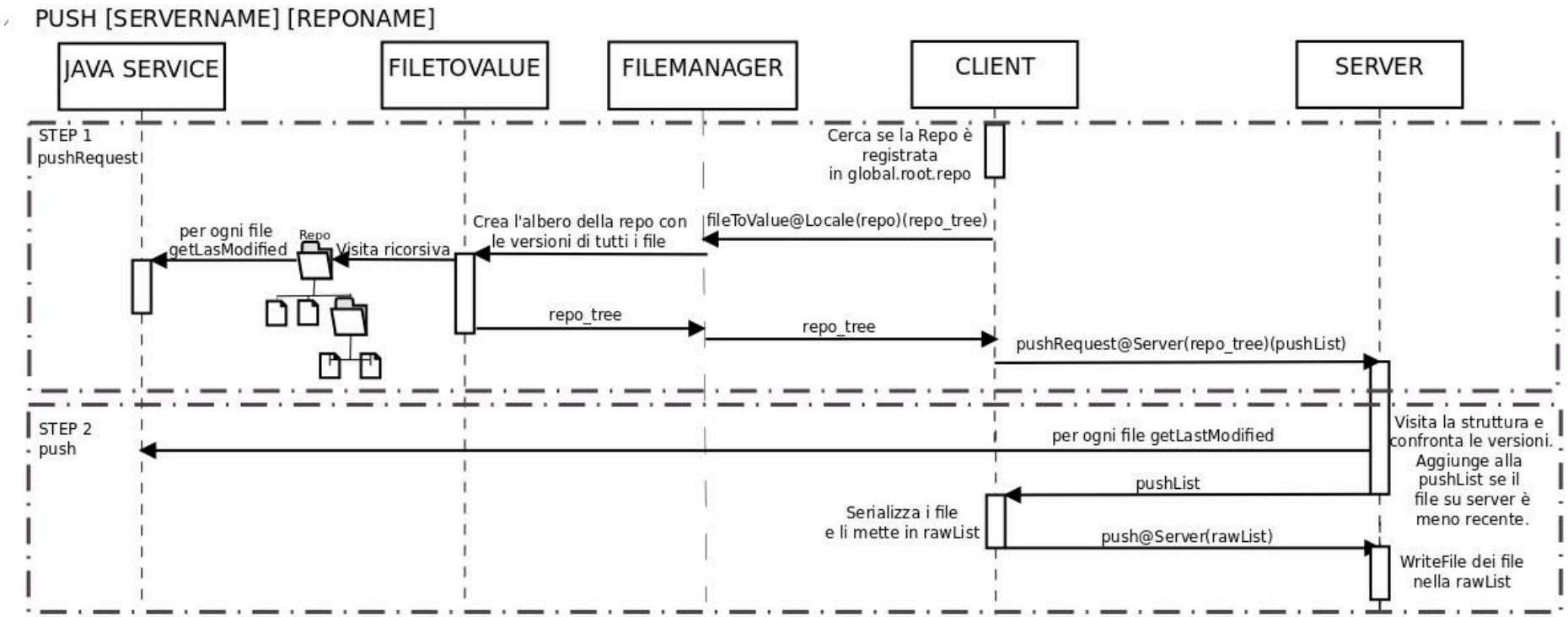
Invece di inviare tutto il contenuto di una repository inviamo solo i file che sono necessari.

La Push si compone quindi di 2 step. Nel primo il client richiede a file manager di visitare ricorsivamente la propria repository e creare la struttura ad albero con i nomi delle cartelle e dei file. Ogni file ha inoltre la data dell'ultima modifica in millisecondi in un long. Il client invia questo albero al server (*pushRequest*).

Nel secondo step il server visita la struttura ad albero inviata dal client e confronta le informazioni dei file nell'albero con i file nella propria repository. Se trova un file non presente su server oppure meno aggiornato su server ne aggiunge il nome a una lista (*fileRequest*) che invia al client.
Il client quindi serializza ogni file nella fileRequest e li invia al server (push) che li scrive nella propria repository.

Inoltre se nel secondo step il server trova qualche file che l'utente non ha aggiornato lo avvisa di effettuare una pull.

La concorrenza sulla repo è gestita con semafori e presenta il rischio di **starvation** per chi vuole fare push nel caso continuino sempre ad arrivare richieste di pull su quella repo.


PUSH [SERVERNAME] [REPONAME]
JAVA SERVICE
FILETOVALUE
FILEMANAGER
CLIENT
SERVER
STEP 1
pushRequest
Cerca se la Repo è registrata in global.root.repo
per ogni file getLasModified
Repo
Visita ricorsiva
Crea l'albero della repo con le versioni di tutti i file
fileToValue@Locale(repo)(repo_tree)
repo_tree
repo_tree
pushRequest@Server(repo_tree)(pushList)
STEP 2
push
per ogni file getLastModified
Visita la struttura e confronta le versioni. Aggiunge alla pushList se il file su server è meno recente.
pushList
Serializza i file e li mette in rawList
push@Server(rawList)
WriteFile dei file nella rawList

Pull
L'operazione di Pull è la versione speculare dell'operazione di Push effettuata dal server.
Il client invia una richiesta della struttura della repository al server specificato.
Il server restituisce la struttura della repository da lui mantenuta (*serverRepo_tree*).
Successivamente viene ricercata localmente la repository, nel caso in cui non sia presente viene richiesto all'utente se e dove vuole crearla.
A questo punto viene effettuato il confronto tra le versioni dei file di client e server, producendo così una lista di file che il server deve inviare al client (*fileRequest*), che è inviata al server, il quale provvede ad inviare la *rawList* contenente i file richiesti.

Delete
La repository è ricercata localmente per corrispondenza di nome server e nome repository. In seguito viene parallelamente inviata la richiesta di rimozione della repository al server e la rimozione locale.