

# JSync - Gruppo Polleg

## COMPONENTI

Sibani, Riccardo, 694291

Boiani, Filippo, 658443

Lo Caso, Nicola, 694703

Loreti, Ludovico, 656551

## Introduzione

Il progetto JSync permette di aggiornare, creare, eliminare e inserire, documenti e file in un Server - del quale si conosce indirizzo e protocollo di comunicazione- attraverso il download e/o l'upload di Repositories.

Una Repository è una cartella contenente file e sotto-directory che possono essere interamente ottenuti e modificati richiamando delle operazioni sul nome della Repository stessa.

Uno o più client possono eseguire le operazioni sulle cartelle contemporaneamente, rispettando i principi di concorrenza studiati nel corso di Sistemi Operativi.

L'utente (Client) può interagire con il programma attraverso una interfaccia a linea di comando (CLI).

## Consegna

La cartella del progetto si divide in Client e Server.

Per avviare il server è necessario:

- inserire da linea di comando il percorso del file server.ol
- digitare il seguente codice:

```
jolie server.ol
```

In questo caso verrà arrivato all'indirizzo di default "socket://localhost:8001".

Se si volesse modificare l'indirizzo è necessario:

```
jolie -C SERVER_LOCATION=\"socket://localhost:numeroPorta\" server.ol
```

Per eseguire il Client è necessario:

- inserire da linea di comando il percorso del file client.ol
- digitare il seguente codice:

```
jolie client.ol
```

**Per eseguire un secondo client è necessario copiare e incollare la cartella Client e digitare i comandi illustrati sopra.**

## Demo

Una volta avviato il Server, questo è già pronto a ricevere richieste e si mette in attesa. Ora si possono eseguire due (o più) client con la procedura esposta nel paragrafo precedente.

- Se si vuole avere una lista dei comandi disponibili basta digitare il comando:

```
J-Sync@Client$ help
```

- Se si vogliono avere informazioni su i vari comandi basta digitare senza parametri il comando:

```
J-Sync@Client$ "NOME_COMANDO"
```

- Per aggiungere un Server bisogna digitare la seguente lista di comandi:

```
J-Sync@Client$ server add
```

```
> Inserire il nome del nuovo server: NomeServer
```

```
> Inserire l'indirizzo del nuovo server: indirizzo:porta
```

```
> Nome Server: NomeServer - Porta: localhost:8001; Continuare? y  
> Server aggiunto correttamente sulla porta
```

in rosso sono evidenziate le informazioni da digitare.

Ripetere la stessa operazione con il secondo Client (e con tutti gli altri se ne sono stati avviati più di due).

- Per aggiungere una Repo:

```
J-Sync@Client$ repo add  
> Inserire il nome del server legato al repository: nomeServer  
> Inserire il nome del nuovo repository: nomeRepository  
> Inserire il Path in cui è la repository: /Users/nomeUser/Downloads/  
> Nome Server: server1 - Repository: REPOEsempio - LocalPath: /Users/  
nomeUser/Downloads/; Continuare? y
```

L'aggiunta della Repository copia i file in una cartella del programma "servers/nomeServer/nomeRepository" e, la carica sul server.

- Per effettuare la pull il secondo client deve eseguire i seguenti comandi:

```
J-Sync@Client$ pull
> Inserire il nome del server legato al repository: nomeServer
> Inserire il nome del nuovo repository: nomeRepo
> Nome Server: nomeServer - Repository: nomeRepo; Continuare? y
```

-Per effettuare la push il client2 deve eseguire i seguenti comandi:

```
J-Sync@Client$ push
> Inserire il nome del server legato al repository: nomServer
> Inserire il nome del nuovo repository: nomeRepo
> Nome Server: nomeServer - Repository: nomeRepo; Continuare? y
Aggiornamento proveniente dal server: 2
Versione del client ora: 2
```

## Implementazione

### STRUTTURA DEL PROGETTO:

#### CLIENT

Il client ha una cartella *data* che contiene i file:

- app.json
- servers.json
- client\_utils.ol

E' stato creato il file **servers.json** per tenere aggiornati e memorizzati i nomi e gli indirizzi associati ai Servers registrati in modo da non doverli riscrivere ogni volta che si accende il programma.

Il file **app.json** contiene la lista dei comandi e la loro descrizione.

Ogni comando possiede:

- Tipo (type) che può essere affermativo, negativo, di chiusura, di help, list ecc...
- Un id per identificarlo nel programma
- I vari nomi con cui l'utente può richiamare quel comando (appearance).
- gli argomenti (se presenti) per quel comando
- I parametri (se presenti)
- La descrizione del comando

I comandi possono avere anche dei sotto comandi, che vengono specificati negli argomenti.

Per esempio se un utente volesse fare un list dei server registrati gli basterebbe digitare:

```
J-Sync@Client$ list servers
> Nome Server: Server1   Indirizzo: socket://192.168.1.18:8001
> Nome Server: Server2   Indirizzo: socket://88.40.179.65:8001
> Nome Server: Server3   Indirizzo: socket://192.168.1.18:8333
```

Si potrebbe usare list anche con parametri diversi, per esempio: `list newrepos` oppure `list regrepos`. Il comando `server` funziona allo stesso modo.

Si può fare: `server add` e `server remove`.

Mentre per effettuare una push bisogna seguire la sequenza di comandi:

```
J-Sync@Client$ push
> Inserire il nome del server legato al repository: nomServer
> Inserire il nome del nuovo repository: nomeRepo
> Nome Server: nomeServer - Repository: nomeRepo; Continuare? y
```

Abbiamo deciso di implementare il comando di push e pull in questo modo per permettere all'utente di controllare meglio ciò che digita.

In generale abbiamo deciso di implementare i comandi con un file .json perché in questo modo possono essere ampliati in futuro e perché possono essere cambiati anche mentre il programma è in esecuzione (nel caso del file server.json).

Il file **client\_utils.ol** contiene due procedure che vengono incluse nel Client: loadJson e arrayControl. La prima ha lo scopo di caricare il file app.json, mentre la seconda ha lo scopo di controllare all'interno del file app.json se il comando digitato esiste ed è corretto.

#### EXTERNAL E LOCAL NEL CLIENT

Il client può delegare i propri doveri a due servizi: atServer (servizio esterno) e Local (servizio interno); ovvero, ogni servizio che non implica la connessione con un database verrà implementato in Local.ol mentre se ci interfacciamo con un server, l'operazione verrà eseguita da atServer.ol.

Questa scelta è stata fatta in quanto in un futuro aggiornamento o modifica del programma, maneggiare i servizi risulterà più veloce ed immediato. Pensiamo, ad esempio, che si cambino le interfacce per collegarsi al server: in questo caso, l'unico file che andrò a modificare ed aggiornare sarà atServer.ol, lasciando intatto Local.ol.

#### PERCHÉ MANCA UN FILE REPO.JSON

Abbiamo deciso di esplorare la cartella "servers" piuttosto che mantenere un file .json di cartelle registrate perché in questo modo l'utente sa cosa c'è realmente nelle cartelle all'interno del programma. Infatti, nel caso l'utente cancellasse una repository in modo "manuale", il file json non se ne accorgerebbe e sarebbe pertanto possibile effettuarne una push. Questa scelta è stata presa data la struttura aperta del programma (come da specifiche) in cui non si passa sempre per il programma per aggiornare una cartella e quindi, non si può delegare ad un file .json la responsabilità di avere sempre in memoria le repositories che ci sono in locale.

#### GETJSON

Nel progetto sia Clients che Servers fanno l'embedding del file GetJson.ol i cui servizi vengono messi a disposizione attraverso l'interfaccia json\_interface.ol che riceve in input il percorso di un file .json e ritorna l'albero con le variabili contenute nel file. Si è voluto implementare separatamente perché è un servizio che potrebbe essere riutilizzato anche in seguito per altre applicazioni.

#### READFILE

Sia Clients che Servers fanno anche l'embedding del file ReadFile.ol.

Il seguente file contiene due servizi: scan ed explore.

L'unico servizio richiamabile dall'esterno è scan. Riceve dall'esterno il percorso iniziale della repository, il nome della repository e il percorso finale della repo; questo servizio chiama a sua volta explore() che si occupa di visitare l'albero delle sotto-directory e dei file ricorsivamente. Al richiedente vengono ritornati i path delle directory i path dei file e il contenuto dei file stessi.

## DYNAMIC ADDRESSING

Inizialmente avevamo pensato di fare un file `init.json` con le configurazioni del server, tuttavia, siccome non è possibile fare il dynamic binding delle `inputPort` (dobbiamo settare la `Location` e il `Protocol`) abbiamo deciso di non utilizzare questa funzionalità (anche se la procedura `verifyConfig` è rimasta nel codice del server).

## REPOS

Il Server mantiene una cartella “`repos`” che contiene tutte le repo presenti sul Server stesso. Ogni repo ha un file con estensione `.version` che contiene il numero della versione. Solo al Server è permesso modificare il numero di versione. Di conseguenza, un client ha la version aggiornata se il numero è lo stesso contenuto nel file `version` del Server. Una volta effettuata una `push`, il Server aumenta il numero contenuto di `.version` e lo invia al Client che ha richiesto la `push`.

## CONCORRENZA

Per gestire la concorrenza il Server utilizza i semafori messi a disposizione dal servizio `semaphore_utils.ol`.

Ogni repo ha 3 semafori associati e una variabile globale, questi semafori vengono rilasciati quando viene creata con il servizio `create`.

Quando il server viene avviato, durante l’inizializzazione, controlla le repository presenti nella cartella “`repos`” e fa la `release` dei semafori per ogni repo.

## PERCHÉ NON USARE SYNCHRONIZED

Abbiamo deciso di utilizzare i semafori piuttosto che il `synchronized` poiché volevamo permettere a più client di fare `push` e `pull` per repo diverse contemporaneamente, bloccando il Client solo nel caso in cui fosse già in atto un’operazione sulla repo specifica richiesta. Di conseguenza se due Client fanno la `push` di due repository con nomi diversi contemporaneamente il Server non blocca nessuna delle due richieste. Il `synchronized` avrebbe messo in coda i due client sul metodo “`push`”, con il nostro metodo ciò non avviene.

## PUSH

Il problema dell’implementazione di *pull* e *push* può essere ricondotto al problema più generico del Reader-Writer. Le due operazioni, `push` e `pull`, corrispondono rispettivamente alla scrittura e alla lettura di una Repo. Il problema viene anche detto secondo problema del reader writer siccome da la precedenza agli scrittori.

Un Client che fa una richiesta di `push` per una determinata Repo sta chiedendo sostanzialmente al Server di scrivere (o meglio, sovrascrivere) quella Repo.

Abbiamo deciso di implementare la `push` in modo da notificare gli eventuali futuri lettori dell’arrivo di uno scrittore, in modo tale da mettere in coda i nuovi lettori arrivati. (Sotto è illustrato l’algoritmo).

La sequenza di operazioni che vengono eseguite per fare la `push` è la seguente:

- Inizialmente bisogna fare il controllo dell’esistenza della Repo in quanto i semafori sono gestiti a livello di `RepoName` (quando una cartella viene creata vengono rilasciati i lock per lavorare con essa). In questo modo è possibile fare più `push` di Repositories diverse contemporaneamente su uno stesso Server.
- Se la repository esiste è possibile fare la `P( PUSH_REPO_NAME)` che mi dà il diritto di modificare la variabile globale `lock( REPO_NAME)` che non è altro che un array dove ogni variabile ha lo stesso nome della repo e come valore un booleano.

- A questo punto faccio la P(SCRITTURA\_NOME\_REPO) che mi dà il diritto di sovrascrivere i file (se non sono stati cancellati tra il controllo di esistenza iniziale e l'acquisizione di PUSH\_REPO\_NAME).
- Controllo la versione, se è aggiornata rispetto al Server copio i file. Se la versione non è aggiornata mando la notifica al Client
- Infine faccio V(SCRITTURA\_NOME\_REPO)
- Setto a FALSE la variabile global lock
- Rilascio il secondo semaforo attraverso una V(PUSH\_NOME\_REPO)

```

1  PUSH( REPO )( INCREMENTO_VERSIONE ){
2      //prima di tutto controllo l'esistenza della repo
3      esiste = exists( REPO )
4      if( esiste == TRUE ){
5
6          //acquisisco il diritto di modificare la variabile lock
7          acquire( PUSH_NOME_REPO )
8
9          //devo notificare l'arrivo di un writer agli altri reader
10         // setto la variabile lock a true così i futuri reader che arrivano aspettano
11         // questa variabile lock serve a notificare ai reader che è arrivata una richiesta di push
12         global.lock.( NOME_REPO ) = TRUE
13
14         /**INIZIO SEZIONE CRITICA SCRITTURA**
15         acquire( SCRITTURA_NOME_REPO )
16             esiste = exists( REPO )
17             if( esiste == TRUE ){
18
19                 //controllio la version: se è aggiornata sovrascrivo la repo preesistente
20                 if( aggiornata ){
21                     //cambio la versione
22                     //invio la notifica di cambiamento
23                 }else {
24
25                     //non è aggiornata
26                 }
27             }else{
28                 //la repo non esiste più sul server,
29                 //i suoi semafori sono ancora esistenti tuttavia
30             };
31         release( SCRITTURA_REPO_NAME )
32         /**FINE SEZIONE CRITICA SCRITTURA**
33
34         global.lock.( REPO_NAME ) = FALSE
35         release( PUSH_NOME_REPO )
36     }
37 }
38
39 - Semafori:
40     PUSH_NOME_REPO = 1
41     SCRITTURA_NOME_REPO = 1
42 - Variabili Globali
43     lock(NOME_REPO) = FALSE

```

## PULL.

Un client che fa una richiesta di pull di una Repo ad un Server può essere visto come un reader per quella determinata Repo.

I reader possono leggere in concorrenza il contenuto della cartella: finché l'ultimo reader non ha finito di leggere, nessuno può fare la push per quella determinata Repository.

Tuttavia, appena arriva al Server una richiesta di push, questa ha la priorità su tutte le future richieste di pull che arriveranno dopo (vengono messe in attesa).

Sotto è illustrato l'algoritmo:

```

1  PULL( NOME_REPO )( FILES ){
2      //controllo se esiste il semaforo relativo alla repo altrimenti rimango in attesa all'infinito
3      if( is_defined( global.lock.( NOME_REPO ) ) ) {
4
5          ///MI BLOCCO SE C'E' QUALCUNO CHE STA SCRIVENDO
6          while( global.lock.( NOME_REPO ) ) {
7              //ASPETTO
8          };
9
10         ///***Inizio Sezione critica 1***
11         //prendo il lock per incrementare il numero di reader e prendere il diritto di lettura
12         acquire( LETTURA_NOME_REPO )
13         global.reader.(NOME_REPO)++; //incremento il numero di reader di quella repo
14
15         if( global.reader.(NOME_REPO) == 1 ){
16
17             acquire( SCRITTURA_NOME_REPO ) //Se sono il primo reader prendo il semaforo di scrittura
18             //controllo se esiste la NOME_REPO (potrebbe esistere il suo semaforo ma
19             // non la NOME_REPO corrispondente: il delete non cancella i semafori)
20             res = exists( NOME_REPO )
21         };
22         release( LETTURA_NOME_REPO );
23         ///***Fine Sezione critica 1***
24
25         //LETTURA
26         if( res == TRUE ){
27             //leggo e mando i file al client
28         }else {
29             //la NOME_REPO non esiste (anche se un tempo è esistita siccome esistono i semafori a suo nome)
30         };
31
32         ///*** Inizio Sezione critica 2***
33         acquire( LETTURA_NOME_REPO )
34         global.reader.(NOME_REPO)--;
35         if( global.reader.(NOME_REPO) == 0 ){
36             release( SCRITTURA_NOME_REPO )
37         };
38         release( LETTURA_NOME_REPO )
39         ///***Fine Sezione critica 2***
40     }else {
41         //la repo non esiste e non è mai esistita
42     }
43 }
44
45 - Semafori:
46     LETTURA_NOME_REPO = 1
47     SCRITTURA_NOME_REPO = 1
48 - Variabili Globali
49     lock(NOME_REPO) = FALSE

```











La sequenza di operazioni che vengono eseguite per fare la push è la seguente:

- Controllo se è definito lock(NOME\_REPO) (che viene creato e rilasciato la prima volta nel createRepo).
- Se è definito significa che esiste la Repo e si può procedere, se non esiste mando la notifica.
- Una volta controllato, si rimane bloccati nel ciclo finché il lock(NOME\_REPO) non viene settato a FALSE (dal writer).
- Uscito dal ciclo posso fare P(LETTURA\_NOME\_REPO) e modificare il contatore dei reader (se sono il primo prendo anche il permesso di scrittura e controllo l'esistenza della repo)
- Rilascio il lock di Lettura.
- Leggo la repo, se esiste, altrimenti mando la notifica al Client
- Una volta finita la lettura, posso ri-acquisire il lock di lettura, modificare il contatore dei reader (e, nel caso sia l'ultimo reader, rilasciare anche il lock di scrittura).

Il lock di scrittura, impedisce ai writers di completare la push fintanto che ci sono reader. Tuttavia, come anticipato precedentemente, le richieste di push hanno la precedenza. Per garantire la precedenza dei writer abbiamo utilizzato una variabile globale chiamata "lock" che viene settata solo dai writers stessi. Quando un reader sta per effettuare una lettura controlla prima se è settata la variabile lock: se è settata a true significa che c'è un writer in coda (o che sta già scrivendo) e i nuovi reader che arrivano dovranno aspettare fino al suo completamento.

Ci potrebbe essere il problema della starvation dei reader per una determinata repo, però con il fatto che per poter fare la push, un writer deve avere la versione aggiornata, ogni writer in coda dopo il primo non avrà più l'ultima versione e dovrà scaricarsela attraverso una pull (diventando così un reader).

Quando un Client fa la pull di una Repository da un Server questa viene salvata in una cartella locale del programma. Il percorso è il seguente:

| ▼  servers |   | oggi 14:10 |
|---|---|------------|
| ▶          | Server1                                 | oggi 14:09 |
| ▶          | Server2                                 | oggi 14:09 |
| ▼          | Server3                                 | oggi 14:10 |
| ▶         | Repo1                                   | oggi 14:10 |
| ▼        | Repo2                                   | oggi 14:09 |
|          | 01 Prayer in C (Robin Schulz Remix).mp3 | 41         |
|          | 11538116_10...3478684_o.jpg             | ieri 16:41 |
|          | Schermata 20...22.55.28.png             | ieri 16:41 |
|          | Schermata 20...16.43.04.png             | ieri 16:43 |

Come si può notare il Client mantiene una cartella diversa per ogni nome di server registrato. Ciascuna cartella nomeServer contiene tutte le repo scaricate da quel server. Tutti i server sono contenute in una cartella chiamata "servers".

Questo avviene perchè, come da specifiche, su due server diversi ci possono essere repositories diverse ma con lo stesso nome contenenti files diversi. Per evitare conflitti: è pertanto necessario creare cartelle diverse per ogni server.

Quando il Client fa il create di una Repository deve specificare il Server in cui vuole crearla, il nome della repo e il localPath. Abbiamo interpretato il localPath come percorso locale nel computer del Client da cui viene copiata la Repository. Di conseguenza il create copia il contenuto della cartella specificata nel localPath e lo invia al Server. Una volta creata la nel Server, il Client la copia nella cartella servers/nomeServer aggiungendo il file .version con il numero di versione inviato dal Server.