

# JSync - Esame di Laboratorio Sistemi Operativi

## A.A. 2014-2015

*>Build software better, together.*

### LOBSTER

Componenti:

- **Gianmarco Spinaci** 0000691241
- **Michele Lorubio** 0000693868
- **Chiara Babina** 0000693799
- **Valentina Tosto** 0000692741

### Introduzione

**JSync-Lobster** è un servizio di condivisione di repositories tra Clients.

Grazie ad una semplice ed intuitiva **UI** in modalità command line (Cli), l'utente può interagire con i Servers registrati, che mantengono le informazioni sulle repositories.

Con pochi comandi l'utente può gestire i diversi Servers e richiedere o creare una repository su di essi, tenendo conto delle varie eccezioni che potrebbero sollevarsi durante l'esecuzione. L'utilità di tale servizio è data attraverso una pratica gestione del problema readers-writers, permettendo a tutti gli utenti di avere la possibilità di gestire le repositories sui Servers in comune, rispettando la concorrenza. La **UX** è agevolata anche grazie all'aggiunta di un comando per visualizzare tutte le funzionalità offerte da JSync.

In seguito spiegheremo nel dettaglio la struttura completa del nostro progetto.

## Consegna

Il progetto deve essere eseguito lanciando i servizi **cli.ol** ed i **server.ol** a disposizione.

Successivamente si scrive un comando in input sulla **UI**, il quale sarà inviato al **clientUtilities.ol**, e si aspetterà una sua risposta.

## Demo

Questa demo esplicativa è in riferimento alla cartella Demo.zip, contenuta nella root del progetto. Di seguito la struttura

Client side	Server side
_WebDesign	4000_localhost
_Desert0.1	5000_localhost

Esistono già 3 Repositories salvate sui Servers

- 4000\_localhost
  - Design\_Assets
  - List\_Desert
- 5000\_localhost
  - SerieA\_player

Elenco di comandi Demo di JSync-Lobster:

**1)** Lanciare entrambi i Servers e Clients

**2)** In entrambi i Clients sono presenti già alcuni Servers registrati

```
list servers
// ritorna la lista dei Servers registrati (contenuti nel file config.xml)
```

**3)** Dal Client **\_Desert0.1** eseguire il comando

```
addServer 5000_localhost socket://localhost:5000
//aggiunge un ulteriore Server registrato
```

A questo punto il Client **\_Desert0.1** avrà entrambi i Servers registrati.

4) Si possono visualizzare le Repositories registrate localmente con il comando

```
list reg_repos
```

oppure tutte le Repositories disponibili sui Servers registrati tramite il comando

```
list new_repos
```

se uno o più Servers non esistono, o non sono raggiungibili, si avrà un responso negativo.

5) Sempre dal Client **\_Desert0.1** eseguire l'operazione Pull della Repository **SerieA\_Player**

```
pull 5000_localhost SerieA_Player
```

6) **\_Desert0.1** a questo punto può modificare la Repository ed effettuare una Push

```
push 5000_localhost SerieA_Player
```

7) In seguito dal Client **\_WebDesign** si può eseguire, a sua volta, una Pull di **SerieA\_Player**

```
pull 5000_localhost SerieA_Player
```

e notificare che la versione è stata effettivamente incrementata, con l'aggiunta degli ulteriori files modificati.

8) Prove sulla gestione della concorrenza possono essere eseguite semplicemente con il comando

```
push 5000_localhost SerieA_Player
```

in entrambi i Clients, contemporaneamente.

9) Ulteriormente si possono eseguire comandi di eliminazione di un Server e di una Repository

```
removeServer 4000_localhost
```

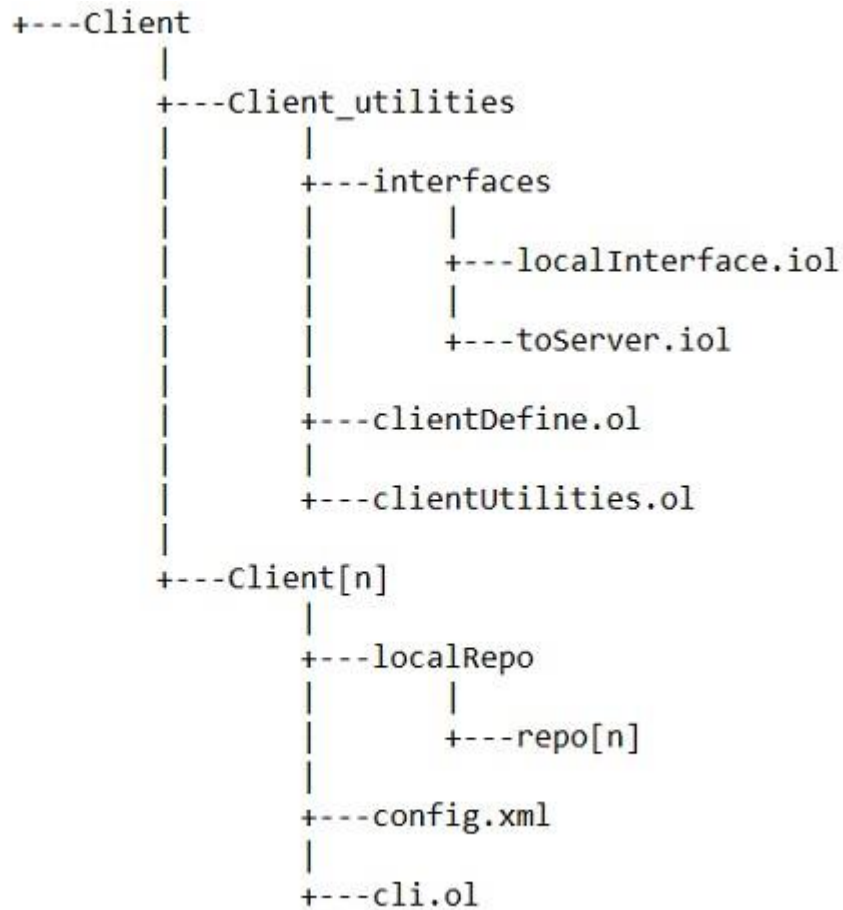
```
delete 5000_localhost SerieA_Player
```

**N.B.** Creazione/Lettura e visita di cartelle non funzionano con cartelle che hanno spazi nel nome!!

# Implementazione

## Struttura del Progetto

### Client

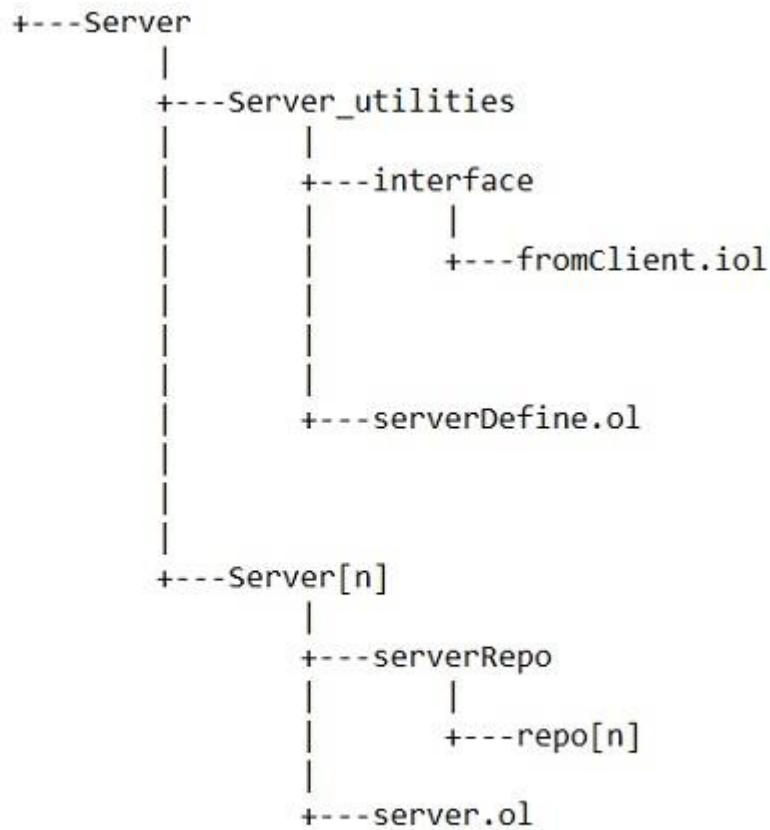


La struttura del Client è composta da:

- Una cartella **client\_utilities** che contiene:
  - La cartella **interfaces** con al proprio interno l'interfaccia locale tra la Cli ed il clientUtilities (localInterface.iol) e l'interfaccia con il Server (toServer.iol)
  - Il servizio **clientDefine.ol** contenente i vari define richiamati in clientUtilities.ol
  - Il servizio **clientUtilities.ol**, il quale ha la funzione di intermediario tra il Client ed il Server, con le operazioni relative ai comandi ricevuti dalla Cli

- Una (o più) cartelle **Client[n]** con all'interno:
  - Il servizio **cli.ol**, che ha la funzione di UI, dove si inseriscono i diversi comandi possibili
  - La cartella **localRepo** (non presente inizialmente), contenente tutte le repositories aggiunte
  - Il file **config.xml**, con la lista dei Servers registrati in ogni Client

## Server



La struttura del Server è composta da:

- Una cartella **server\_utilities** che contiene:
  - La cartella **interface** con all'interno l'interfaccia tra il clientUtilities ed il Server (fromClient.iol)
  - Il servizio **serverDefine.ol** contenente i vari define richiamati in server.ol

- Uno (o più) cartelle **Server[n]** con all'interno:
    - Il servizio **server.ol**, con le operazioni relative ai diversi comandi provenienti dal clientUtilities.ol
    - La cartella **serverRepo** (non presente inizialmente), contenente tutte le repositories aggiunte
- 

Il progetto è diviso in **più Cli** (che corrispondono ai diversi Client) e **Servers**, mentre il servizio **"clientUtilities.ol"** ha la funzione di gestire i diversi comandi tra i Clients e i Servers.

La Cli è collegata con il clientUtilities attraverso una pratica di embedding, per permettere la comunicazione senza aver bisogno di un indirizzo.

Inoltre sono utilizzati dei servizi chiamati **"clientDefine.ol"** (dalla parte del Client) e **"serverDefine.ol"** (dalla parte del Server) per gestire diversi metodi, tra i quali la lettura e la scrittura del file xml, il registro dei Servers, la visita ricorsiva di una repository, la creazione di cartelle e l'operazione modulo per l'incremento dei readers / writers.

In particolare, dopo aver inserito un comando in console, è splittato nella Cli, per analizzare ogni singola stringa e differenziare le varie funzionalità; successivamente si invia il comando nel clientUtilities, eseguendo l'operazione relativa.

Di seguito elenchiamo i comandi, descrivendoli nello specifico:

**N.B.:** Ogni funzionalità è inclusa in uno scope per gestire le eccezioni che si presentano, tra le quali l'inserimento di dati non corretti, la connessione assente con il Server ed un file non trovato.

**N.B. (2):** L'esecuzione del procedimento è eseguito solo se i risultati splittati nella Cli corrispondono alla lunghezza richiesta del comando (es: list(1) servers(2) -> il risultato dello split deve essere di dimensione 2).

## **List Servers**

Inizialmente si controlla che i dati siano inseriti correttamente, in seguito si procede alla lettura del file xml (richiamato dal servizio **clientDefine**): se la lista dei Servers non è vuota, per ognuno di essi si stampano le relative informazioni (nome ed indirizzo), altrimenti sarà visualizzato un messaggio di avviso che non sono presenti Servers registrati.

## List reg repos

Esegue l'operazione list della libreria "**File**" di Jolie, ritorna l'elenco totale delle repositories, all'interno della cartella "localRepo". Se non ci sono repositories ritorna un messaggio di errore, in caso positivo ritorna la lista di tutte le repositories locali.

## Add Server

Quando si aggiunge un Server si richiama il metodo per la lettura del file xml e si effettua prima un controllo se il nome o indirizzo del Server sono già presenti (confrontando il nome o l'indirizzo scritto in input con quelli presenti sulla lista), in tal caso viene stampato un messaggio, altrimenti inseriamo il nome ed indirizzo del Server desiderato nel file xml, richiamando il metodo writeFile, presente sempre nel servizio **clientDefine**.

## Remove Server

Dopo la lettura del file xml, Se il nome inserito in input corrisponde ad uno dei nomi registrati sulla lista, allora viene eliminato (con il comando undef) e si richiama la scrittura del file xml per apportare le modifiche effettuate.

**N.B.** se durante la ricerca trova il serverName richiesto dall'utente, allora dopo averlo eliminato, e riscritto il file di configurazione salta un'eccezione, che viene catturata e chiude l'intera operazione.

Ottimizza la ricerca, differenziando caso ottimo[ $O(1)$ ] da caso pessimo[ $O(n)$ ].

## Add Repository

Comando per il quale è necessario l'intervento del Server, perchè serve per aggiungere una repository sia sul Client che sul Server.

### **Client:**

1. Riceve dalla Cli il nome del Server a cui si deve connettere, e si effettua il binding, attraverso il metodo registro (nel servizio **clientDefine**), scorrendo la lista dei Servers per individuare l'indirizzo del Server a cui collegarsi e modificando la Location della porta di comunicazione.
2. Con l'operazione addRepository si controlla sul Server se il nome della repository da inserire è già esistente, in questo caso ritorna un errore. Invece se non sono presenti errori prosegue analizzando il percorso della directory locale inserito in input.
3. Si richiama la visita delle cartelle (nel servizio **clientDefine**) e per ogni file trovato si legge il suo percorso assoluto (readFile) per ottenere così tutto il contenuto del file; in seguito si invia al Server il percorso relativo di ogni singolo file, provvedendo ad inserirlo nella repository appena creata. **N.B.** E' necessaria la distinzione tra percorso assoluto e percorso relativo. Il percorso assoluto è il percorso locale della cartella da cui attingere i file. Si avranno tanti percorsi relativi quanti sono i file all'interno della cartella
4. Successivamente si richiama il metodo writeFilePath (nel servizio **clientDefine**) per creare tutte le cartelle del percorso relativo del file, nel caso in cui non esistessero.

5. Infine nella repository locale creata, si inserisce un file di versione, incrementato ogni volta che si esegue una push.

#### **Server:**

1. Riceve dal Client il nome della repository da ricercare, se già esiste la cartella corrispondente allora ritorna un messaggio di errore, altrimenti si crea e si aggiunge il file di versione con contenuto uguale a 0.
2. Con un'ulteriore operazione riceve il percorso relativo di ogni singolo file, attraverso la visita della directory locale, che viene splittato, per creare la cartella a cui appartiene il file, ed infine scritto nella repository richiesta, con il comando writeFile.

#### **List new repos**

Comando per far stampare tutte le repositories dei Servers registrati.

#### **Client:**

1. Se la lista dei Servers nel file xml non è vuota si collega ad ognuno di essi, e cattura un'eccezione nel caso in cui uno o più Servers non esistano o non siano raggiungibili.

**N.B.** la richiesta della lista delle repositories incluse nel Server, è implementata all'interno di uno scope, se anche un solo Server non è raggiungibile il corrispondente Fault non interromperà l'intera operazione, ma passerà al Server successivo.

#### **Server:**

1. Riceve la richiesta dal Client di inviare tutti i nomi delle repositories presenti nella directory "serverRepo". Richiama il metodo listFile di **file.iol** per ottenere l'elenco, se sono disponibili repositories, ed inviarlo al Client, altrimenti sarà stampato un messaggio di avviso relativo al Server con nessuna repository.

#### **Delete**

Comando per eliminare una repository sia sul Server che sul Client.

#### **Client:**

1. Si legge il file xml e si richiama il metodo registro (nel servizio **clientDefine**), per ricavare l'indirizzo necessario a comunicare con il Server richiesto.
2. Si invia la richiesta di eliminazione della repository al Server ed in caso di cancellazione avvenuta (oppure se la repository già era stata cancellata in precedenza), si esegue la stessa operazione nel Client, deleteDir di **"file.iol"**, eliminando l'intera cartella richiesta dall'utente.



## Server:

1. Riceve il messaggio dal Client con il nome della repository da eliminare, scorre la lista di tutte le repositories presenti in "serverRepo" e se il nome corrisponde a quello ricevuto, elimina la cartella e ritorna il messaggio di operazione effettuata.

## Push

Comando per inviare l'aggiornamento di una repository del Client sul Server, controllando i files di versione di entrambi.

## Client:

1. Si legge il file xml e si richiama il metodo registro (nel servizio **clientDefine**), per prelevare l'indirizzo del Server nel quale inviare la push.
2. Si invia la richiesta di incremento della variabile globale dei writers.
3. Se la variabile è stata incrementata, si procede alla spedizione del file di versione, per controllare se è maggiore o uguale di quello sul Server (solo in tal caso si può eseguire la push) oppure se esiste la repository da inviare (altrimenti deve essere creata).
4. Si esegue la lettura di tutti gli altri files (ignorando quello di versione), si modifica la repository globale da "localRepo" a "serverRepo" e si inviano uno alla volta sul Server, sovrascrivendoli su quelli già presenti o creandoli se non esistono.
5. Il file di versione è gestito attraverso una richiesta al Server, che invierà la sua versione da sovrascrivere a quella locale, dopo che già era stata incrementata.
6. Infine si invia la richiesta di decremento della variabile globale dei writers, a operazione conclusa.

**N.B.** la richiesta di incremento/decremento della variabile globale Writers/Readers è ciò che scandisce la concorrenza

## Server:

1. Riceve la richiesta di incremento della variabile globale dei writers (all'interno di un costrutto synchronized per rendere atomica l'operazione) solo nel caso in cui il numero dei readers sia uguale a 0, altrimenti la push non può essere eseguita.
2. Se la variabile dei writers è stata incrementata, riceve il file di versione dal Client e controlla inizialmente se esiste già la repository richiesta, con all'interno il file di versione: se esiste allora confronta le due versioni, solo se quella del Client è maggiore o uguale di quella del Server allora si incrementa (anche questa operazione all'interno di un costrutto synchronized); altrimenti se la repository non esiste, si crea e si inserisce all'interno il file di versione, inviato dal Client
3. Riceve uno alla volta i files dal Client, per sovrascriverli ai suoi o crearli se non sono presenti.
4. Riceve la richiesta dal Client di inviargli il file di versione incrementato.
5. Infine decrementa la variabile globale dei writers, inclusa in un costrutto synchronized.

## Pull

Comando per scaricare una repository specifica dal Server e sovrascriverla alla propria locale oppure crearla se non è presente.

### **Client:**

1. Si legge il file xml e si richiama il metodo registro (nel servizio **clientDefine**), per prelevare l'indirizzo del Server richiesto.
2. Si invia la richiesta di incremento della variabile globale dei readers.
3. Se la variabile dei readers è stata incrementata, si invia il nome della repository desiderata al Server ricevendo la struttura di tutte le cartelle e sottocartelle.
4. Dopo che si ha a disposizione la struttura, si richiedono i files, uno alla volta, al Server, che provvederà ad inviarli.
5. Quando arriva un file, si sostituisce il nome della repository globale da "serverRepo" a "localRepo" e si creano le cartelle per i files che ne necessitano, in caso non esistano.
6. Infine si invia la richiesta di decremento della variabile globale dei readers, a operazione conclusa.

### **Server:**

1. Riceve la richiesta di incremento della variabile globale dei readers (all'interno di un costrutto synchronized per rendere l'operazione atomica) sono nel caso in cui il numero dei writers sia uguale a 0, altrimenti la pull non può essere eseguita.
  2. Se la variabile dei readers è stata incrementata, riceve il nome della repository da inviare; se esiste ritorna un messaggio di successo insieme alla struttura delle cartelle contenute nella propria repository.
  3. Riceve una alla volta la richiesta di ogni file contenuto nella repository in questione, e lo spedisce al Client.
  4. Infine decrementa la variabile globale dei readers, inclusa in un costrutto synchronized.
- 

## **Gestione del problema reader-writer**

Per la gestione della concorrenza tra readers e writers abbiamo avuto diverse alternative valide: l'uso dei semafori nella libreria di Jolie; la gestione dei controlli tramite il file di versione; l'utilizzo di una coda o un merging dei files (GitHub style).

Alla fine abbiamo optato per due implementazioni diverse:

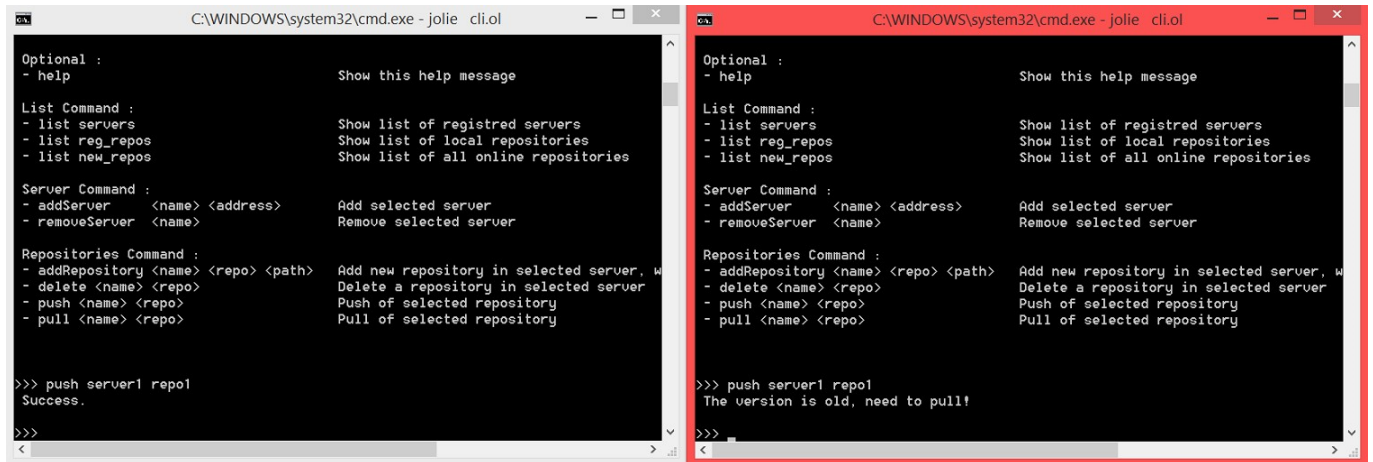
### • **Push-push**

Due (o più) writers sono gestiti attraverso il controllo di versione, se il Client1 prova ad effettuare una push mentre il Client2 sta già eseguendo la sua sulla stessa repository, allora il Client1 dovrà prima aggiornare la sua versione, con una pull, e solo successivamente può caricare i suoi files. Poichè la

scrittura del file di versione è una sezione critica, abbiamo deciso di utilizzare un costrutto synchronized per racchiudere questa parte, in modo tale che l'istruzione sia eseguita in maniera atomica.

(Siamo consapevoli che questa scelta porta ad una perdita di dati da parte del Client1 - problema trattato nel dettaglio nella sezione dei "Problemi riscontrati").

E' da tenere presente che due push di due repositories diverse sono permesse, perché si tratta di diverse informazioni condivise.



```
C:\WINDOWS\system32\cmd.exe - jolie cli.ol
Optional :
- help                Show this help message

List Command :
- list servers         Show list of registred servers
- list reg_repos       Show list of local repositories
- list new_repos       Show list of all online repositories

Server Command :
- addServer <name> <address> Add selected server
- removeServer <name>       Remove selected server

Repositories Command :
- addRepository <name> <repo> <path> Add new repository in selected server, w
- delete <name> <repo>           Delete a repository in selected server
- push <name> <repo>            Push of selected repository
- pull <name> <repo>            Pull of selected repository

>>> push server1 repo1
Success.
>>>
```

```
C:\WINDOWS\system32\cmd.exe - jolie cli.ol
Optional :
- help                Show this help message

List Command :
- list servers         Show list of registred servers
- list reg_repos       Show list of local repositories
- list new_repos       Show list of all online repositories

Server Command :
- addServer <name> <address> Add selected server
- removeServer <name>       Remove selected server

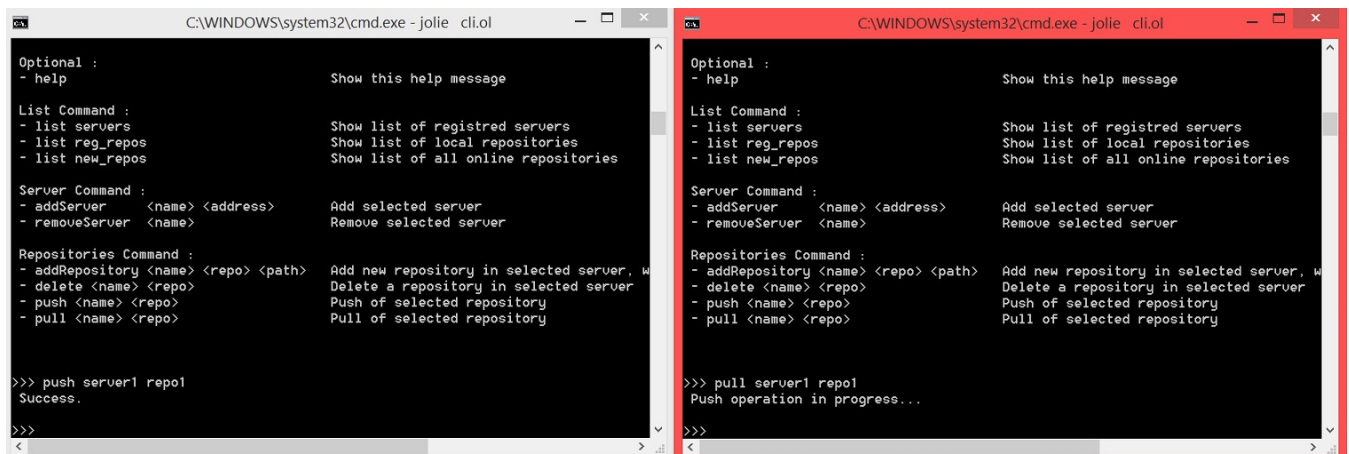
Repositories Command :
- addRepository <name> <repo> <path> Add new repository in selected server, w
- delete <name> <repo>           Delete a repository in selected server
- push <name> <repo>            Push of selected repository
- pull <name> <repo>            Pull of selected repository

>>> push server1 repo1
The version is old, need to pull!
>>>
```

## • Push-pull / pull-push

Per gestire il problema writer-reader e viceversa, abbiamo utilizzato dei contatori (reader e writer) globali e atomici nel Server, che saranno condivisi da tutte le operazioni del Server richieste dai Clients. Nello specifico si possono presentare due casi:

- Il Client1 esegue una **push**: il writerCount sarà incrementato e se contemporaneamente il Client2 esegue una pull, controlla se il writerCount è uguale a 1, in questo caso sarà bloccato con un messaggio di avviso e solo in seguito, quando la push sarà completata, potrà richiamare la pull.



```
C:\WINDOWS\system32\cmd.exe - jolie cli.ol
Optional :
- help                Show this help message

List Command :
- list servers         Show list of registred servers
- list reg_repos       Show list of local repositories
- list new_repos       Show list of all online repositories

Server Command :
- addServer <name> <address> Add selected server
- removeServer <name>       Remove selected server

Repositories Command :
- addRepository <name> <repo> <path> Add new repository in selected server, w
- delete <name> <repo>           Delete a repository in selected server
- push <name> <repo>            Push of selected repository
- pull <name> <repo>            Pull of selected repository

>>> push server1 repo1
Success.
>>>
```

```
C:\WINDOWS\system32\cmd.exe - jolie cli.ol
Optional :
- help                Show this help message

List Command :
- list servers         Show list of registred servers
- list reg_repos       Show list of local repositories
- list new_repos       Show list of all online repositories

Server Command :
- addServer <name> <address> Add selected server
- removeServer <name>       Remove selected server

Repositories Command :
- addRepository <name> <repo> <path> Add new repository in selected server, w
- delete <name> <repo>           Delete a repository in selected server
- push <name> <repo>            Push of selected repository
- pull <name> <repo>            Pull of selected repository

>>> pull server1 repo1
Push operation in progress...
>>>
```

- Il Client1 esegue una **pull**: il readerCount sarà incrementato e se il Client2 esegue una push dovrà controllare se il readerCount è maggiore o uguale a 1, nel caso i readers siano più di uno, il Client2 dovrà aspettare a tempo indeterminato (problema di starvation). Solo quando il readerCount è uguale a 0, il Client2 potrà effettuare la push.

The image shows two terminal windows side-by-side, both titled "C:\WINDOWS\system32\cmd.exe - jolie cli.ol". The left window displays the help message for the Jolie CLI, which includes sections for Optional commands (help), List Command (list servers, list reg\_repos, list new\_repos), Server Command (addServer, removeServer), and Repositories Command (addRepository, delete, push, pull). Below the help message, the command "pull server1 repo1" is entered, resulting in the output "Success, pull request done." The right window shows the same help message, but the command "push server1 repo1" is entered, resulting in the output "Pull operation in progress...".

## • Pull-pull

Due (o più) readers invece sono permessi, quindi non abbiamo inserito **nessun controllo**, perchè tutti possono scaricare contemporaneamente il contenuto della stessa repository.

The image shows two terminal windows side-by-side, both titled "C:\WINDOWS\system32\cmd.exe - jolie cli.ol". The left window displays the help message for the Jolie CLI, which includes sections for Optional commands (help), List Command (list servers, list reg\_repos, list new\_repos), Server Command (addServer, removeServer), and Repositories Command (addRepository, delete, push, pull). Below the help message, the command "pull server1 repo1" is entered, resulting in the output "Success, pull request done." The right window shows the same help message, but the command "pull server1 repo1" is entered, resulting in the output "Success, pull request done."

## Define utilizzati

Nei servizi `clientDefine.ol` e `serverDefine.ol` abbiamo incluso dei define, richiamati frequentemente nei diversi comandi:

### Registro - clientDefine

Utilizzato per settare la location (indirizzo) ad un Server richiesto.

```
18 /*
19  * Setta la location in base al nome ed indirizzo del Server.
20  * Per ogni Server presente nella lista, se il nome è uguale
21  * a quello scritto in input, allora setta la location del
22  * Server tramite il suo indirizzo prelevato dalla lista
23  */
24 */
25 define registro
26 {
27     name -> configList.server[k].name;
28     address -> configList.server[k].address;
29
30     for(k = 0, k < #configList.server, k++) {
31
32         if( name == serverName ) {
33
34             ServerConnection.location = address
35         }
36     }
37 }
```

### Lettura / scrittura del file xml – clientDefine

Per la lettura e scrittura del file xml, che contiene la lista dei Servers registrati dai diversi utenti.

```
40 /*
41  * Lettura del file xml che contiene la lista dei Servers
42  */
43 define readFile
44 {
45     scope( fileXml )
46     {
47         // Pulizia del file configList dei Servers e della variabile file
48         undef( configList );
49         undef( file );
50
51         // Se il file xml non esiste, setta la variabile come vuota
52         install( FileNotFound => configList.vuoto = true );
53
54         // Parametri per la lettura del file
55         file.filename = "config.xml";
56         file.format = "binary";
57
58         // Lettura file xml di configurazione
59         readFile@File( file )( configFile );
60
61         // Salva il file di configurazione nella variabile configList
62         xmlToValue@XmlUtils( configFile )( configList )
63     }
64 }
65 }
66
67
68 /*
69  * Scrittura del file xml per inserire la lista dei Servers
70  */
71 define writeFile
72 {
73
74     stringXml.rootNodeName = "configlist";
75     stringXml.root << configList;
76
77     // Trasformazione della variabile in una stringa in formato xml
78     valueToXml@XmlUtils( stringXml )( fileXml );
79
80     // Parametri della scrittura file
81     file.content = fileXml;
82     file.filename = "config.xml";
83
84     // Creazione del file xml partendo dalla stringa nello stesso formato
85     writeFile@File( file );
86
87     // Pulizia della configList dei Servers e della variabile file
88     undef( configList );
89     undef( file )
90 }
```

## Creazione di cartelle - clientDefine

Per ogni cartella all'interno del percorso di un singolo file, si richiede la sua creazione. Se è già esistente, non sarà creata. **e.g.** nel caso di una pull di una Repository che contiene più cartelle annidate, se non esiste il percorso esatto (formato dalle stesse cartelle del Server) si incorrerà in un errore)

```
93  /*
94  * Creazione delle cartelle, durante l'invio dei files,
95  * nel caso non siano presenti
96  */
97  define writeFilePath
98  {
99      scope( EmptyDirectory )
100      {
101          install( IOException => nullProcess );
102
103          toSplit = toSend.filename;
104          toSplit.regex = "/";
105
106          split@StringUtils( toSplit )( splitResult );
107
108          // Creazione di ogni cartella contenuta nel percorso
109          // (tranne per il file)
110          for(j=0, j<#splitResult.result-1, j++){
111              dir += splitResult.result[j] + "/";
112              mkdir@File( dir )()
113          };
114          writeFile@File( toSend )();
115          undef( dir )
116      }
117  }
```

## Visita delle cartelle - clientDefine & serverDefine

Per la visita ricorsiva delle cartelle.

La visita funziona quanto segue: partendo da un percorso assoluto, si utilizza il comando list dell'interfaccia **string\_utils.iol** per ottenere tutte le sottocartelle e i files in esso contenuti. Con un ciclo for si salvano le sottocartelle e i files in una variabile diversa e per ognuno si applica nuovamente il comando list: se ciò che ritorna è un elemento vuoto, allora significa che si sta esaminando un file o una cartella vuota.

Nel caso in cui il nome contenga un ".", il percorso viene salvato in una variabile finale perchè si tratta di un file; in caso contrario il percorso non viene salvato.

Poichè la struttura che memorizza i percorsi assoluti delle cartelle, memorizza anche un attributo booleano mark, che indica se la cartella è già stata visitata o meno, con un ciclo while si cerca la prima cartella con tale attributo settato a false.

Dopo ciò, si preparano le variabili per iniziare nuovamente la visita, e si richiama il define.

**N.B.** Questo metodo non funziona con cartelle che hanno spazi nel nome!!

```

124  /*
125   * Definizione della visita ricorsiva di tutte le cartelle
126   */
127  define visita
128  {
129
130      root.directory = abDirectory;
131
132      list@File( root )( subDir );
133
134      for(j = 0, j < #subDir.result, j++) {
135
136          // Salva il percorso assoluto e relativo
137          cartelle.sottocartelle[i].abNome = abDirectory + "/" + subDir.result[j];
138
139          newRoot.directory = cartelle.sottocartelle[i].abNome;
140
141          // Viene controllato se la cartella ha delle sottocartelle,
142          // se non le ha si salva tutto il percorso per arrivare in quella cartella
143          list@File( newRoot )( last );
144
145          if(#last.result == 0)
146          {
147              len = #folderStructure.file;
148
149              currentFileAbsName -> cartelle.sottocartelle[i].abNome;
150
151              currentFileAbsName.substring = ".";
152
153              contains@StringUtils( currentFileAbsName )( isContained );
154
155              if( isContained == true )
156
157                  folderStructure.file[len].absolute = currentFileAbsName
158          };
159
160          i++
161      };
162
163      i = 1;
164
165      // Finchè una sottocartella è già stata visitata, si passa alla successiva
166      while(cartelle.sottocartelle[i].mark == true && i < #cartelle.sottocartelle) {
167
168          i++
169      };
170
171      // Se non si è arrivati alla fine di tutte le sottocartelle, l'attributo mark della cartella viene
172      // settato a true, e si richiama il metodo visita
173      if( is_defined( cartelle.sottocartelle[i].abNome )) {
174
175          cartelle.sottocartelle[i].mark = true;
176
177          abDirectory = cartelle.sottocartelle[i].abNome;
178
179          i = #cartelle.sottocartelle;
180
181          visita
182      }
183  }
184

```



## Inizializzazione delle cartelle - clientDefine & serverDefine

Per inizializzare la visita delle cartelle, e per ottenere i percorsi relativi.

Il percorso assoluto iniziale viene diviso e in una variabile viene salvato l'ultimo elemento ottenuto dall'operazione split, che corrisponde all'inizio del percorso relativo.

Si richiama la visita, e la variabile ottenuta viene divisa nuovamente: quando si trova un valore uguale alla variabile che ha memorizzato il percorso relativo, allora tutti i successivi valori, che erano stati divisi, vengono salvati per formare il percorso relativo.

```
186  /*
187  * Inizializzazione della visita e chiamata ricorsiva
188  */
189  define initializeVisita
190  {
191
192      // Trovo la cartella iniziale del percorso relativo
193      abDirectory.regex = "/";
194
195      split@StringUtils( abDirectory )( resultSplit );
196
197      rlDirectory = resultSplit.result[#resultSplit.result-1];
198
199      undef( abDirectory.regex );
200
201      // Predispongo la visita
202      i = 1;
203      visita;
204
205      // Per ogni file prendo il percorso assoluto e lo splitto
206      for(i=0, i<#folderStructure.file, i++){
207
208          folderStructure.file[i].absolute.regex = "/";
209
210          split@StringUtils( folderStructure.file[i].absolute )( resultSplit );
211
212          // Per ogni elemento splittato, costruisco il suo percorso relativo
213          for(j=0, j<#resultSplit.result, j++){
214
215              if( resultSplit.result[j] == rlDirectory ){
216
217                  while( j<#resultSplit.result-1 ){
218
219                      folderStructure.file[i].relative += "/" + resultSplit.result[j+1];
220
221                      j++;
222                  }
223              }
224          };
225
226          undef( folderStructure.file[i].absolute.regex )
227      }
228  }
```



## Modulo - serverDefine

Utilizzato per individuare il corretto indice della variabile globale che corrisponde ai readers/writers. Poichè si dispone di due indici, 0 e 1, l'operazione modulo viene effettuata solo nel caso in cui, in una formula  $a \bmod b$ ,  $a$  sia maggiore di  $b$ , quindi solo quando l'indice corrisponde a 1; nel caso in cui l'indice sia minore, questo viene solo incrementato di 1. Di conseguenza, a seconda dell'indice passato si ottiene quello opposto.

E' stato usato in questo modo per poter generalizzare il più possibile la Request-Response per l'incremento della variabile globale dei readers/writers.

```
89  /*
90   * Definizione del modulo, nel quale si passa l'id del reader o writer e si esegue
91   * il modulo per individuare l'indice dove è contenuto il numero di readers/writers,
92   * da controllare per sapere se poter eseguire l'incremento oppure bloccarsi
93   * (se il numero di readers/writers è maggiore o uguale di 1)
94   */
95  define modulo
96  {
97
98      a = operando;
99      b = 1;
100
101      if( a < b ) {
102
103          mod = a+1
104      }
105
106      else {
107
108          mod = a%b
109      }
110
111  }
```

---

## Problemi riscontrati & soluzioni adottate

### File manager

Inizialmente, per non appesantire il Client e per sfruttare nel migliore dei modi i servizi di Jolie, volevamo implementare un servizio a parte chiamato **fileManager.ol**.

Questo servizio, collegato al **clientUtilities.ol** attraverso la pratica dell'embedding, serviva per gestire la lettura e scrittura del file xml e per la visita ricorsiva delle cartelle.

Alla fine però non è stato possibile mettere in atto questa idea perchè abbiamo riscontrato dei problemi.

Lavorando su sistemi operativi diversi abbiamo notato che su macchina Linux si incorreva in errori riguardanti i threads e il programma si arrestava. Invece su macchina Windows sembrava non esserci alcun errore, abbiamo provato a fare le prove da lei consigliate, disinstallare OpenJDK e installare la versione ufficiale di Java, cioè quella di Oracle, ma il problema non si è risolto.

Non riuscendo a capire il perchè su macchina Linux il programma generasse questi errori, abbiamo cercato di capire se su macchina Windows andasse veramente tutto bene. Dopo tante prove abbiamo riscontrato il problema anche su di esso, notando che l'utilizzo della CPU era notevole; infatti appena lanciato il programma,

l'utilizzo della CPU arrivava al 100%, dopo qualche secondo il consumo si abbassava, per ritornare in pochi secondi al 100%.

Allora a quel punto abbiamo deciso di creare il servizio **clientDefine.ol**, contenente tutti i define richiamati nei comandi del servizio **clientUtilities.ol**, con la sola differenza di implementarlo senza embedding. In tal modo il programma gira perfettamente su entrambi i sistemi operativi, con un ridotto utilizzo della Cpu.

## Add Repository / Pull (gestione delle cartelle)

Abbiamo avuto dei problemi riguardo i percorsi delle cartelle, poichè non sapevamo come far visitare tutte le sottocartelle della cartella principale e non solo i files contenuti all'interno. In seguito abbiamo deciso di implementare una visita ricorsiva di tutte le sottocartelle, gestendo anche la differenza tra la lettura del file, che accetta un percorso assoluto, e la scrittura, che accetta un percorso relativo.

Inoltre nell' **Add repository**, se sono presenti cartelle vuote nella directory locale che si desidera aggiungere nel Client e nel Server, abbiamo deciso di non farle aggiungere, mentre nella **Pull** ritorna un messaggio di repository vuota, se nel Server è stato cancellato il contenuto della repository in questione.

## Invio di immagini

Nella nostra soluzione non è possibile spedire immagini, ma **solo files prettamente testuali**. Per quella funzionalità, basterebbe aggiungere una conversione in binario del contenuto del file.

## Push della stessa Repository

Durante le scelte d' implementazione della push, abbiamo riscontrato il problema della **perdita di dati** da parte di un Client. Per esempio se avvengono due push della stessa repository, ad avere una perdita di dati sarà il Client con la versione più vecchia, cioè quello che arriva successivamente.

Abbiamo pensato che comunque il Client in questione sarà notificato con un messaggio di aggiornare la versione prima di procedere con la push, quindi prima di effettuare la pull potrà salvare i dati su cui stava lavorando senza perdere nulla.

La procedura più ragionevole sarebbe stata sicuramente quella del merging dei files, però ciò non era richiesto nelle specifiche del progetto quindi abbiamo optato per tralasciare questa idea.

## .DS\_Store su MAC

Visto che abbiamo lavorato solo su macchine Windows e Linux, abbiamo voluto provare il progetto anche su un Mac.

Eseguendo le varie prove abbiamo notato un errore mai comparso nè su Windows nè su Linux. Quando si apre una cartella manualmente, come "localRepo", si genera automaticamente un file nascosto chiamato ".DS\_Store". E se in seguito si richiama un comando, ad esempio `list reg_repos`, oltre a stampare le repositories salvate localmente, stamperà anche questo file nascosto.

## Define Modulo

Il define **Modulo** è stato implementato così inizialmente, poi ci siamo resi conto che ci sarebbe potuta essere un'ottimizzazione del codice: si sarebbe potuto eliminare il confronto della variabile passata traslando le posizioni delle variabili globali da 0 e 1, a 1 e 2, in modo tale che l'operazione modulo si sarebbe potuta risolvere quanto segue (sia 'a' il valore passato corrispondente all'indice della variabile, e 'mod' il risultato ottenuto):

$$(a+1) \% 2 = \text{mod}+1$$

In questo modo, se la variabile passata fosse stata 1, allora il risultato sarebbe stato 2; se invece la variabile fosse stata 2, il risultato sarebbe stato 1.

Avevamo anche pensato di tenere come indici della variabile globale 0 e 1, mentre gli identificativi passati dalle operazioni **push** e **pull** settarli come 1 e 2: l'operazione modulo avrebbe funzionato anche senza fare nessun tipo di incremento alla variabile 'a' e alla variabile finale 'mod', ma avremmo dovuto fare comunque dei confronti per fare l'incremento e il decremento della posizione corrispondente alle due operazioni, riducendo l'ottimizzazione data dal metodo modulo.

Infine abbiamo mantenuto la scelta fatta a livello di codice per questioni di tempo sulla consegna.