Jolella - Laboratorio di Sistemi Operativi

Documentazione di Progetto per Laboratorio di Sistemi Operativi

A.A. 2018-2019

Joliardici

30 Settembre 2019

Abstract

Questo documento è un report che descrive le modalità e le scelte implementative attuate per la creazione della rete Jolella. Tale report è suddiviso in diverse sezioni: una breve descrizione dei componenti del gruppo, una introduzione descrittiva del progetto, a seguire le istruzioni per eseguire la demo del programma ed infine una discussione più approfondita sulle scelte implementative effettuate.

Sommario

1 CONTATTI

Contatti del Gruppo

- Andrea Pagliarani, Matr. 0000816920 (Referente)
  andrea.pagliarani6@studio.unibo.it
- Erind Peci, Matr. 0000766703
  erind.peci@studio.unibo.it
- Zoubeir Warrache, Matr. 0000772430
  zoubeir.warrache@studio.unibo.it
- Elisa Silvegni, Matr. 0000817268
  elisa.silvegni@studio.unibo.it

2 DESCRIZIONE DEL PROGETTO

Il progetto assegnato prevede la creazione di un sistema distribuito per la condivisione di file decentralizzato, in una rete di entità pari tra loro. In una rete *jeer-to-jeer (**J2J**,* così chiamata per l'utilizzo del linguaggio di programmazione orientato ai servizi, ***Jolie**)*, ogni nodo ha funzione di client e di server verso gli altri nodi terminali della rete (*jeer*) e condivide con gli altri determinate risorse (nel nostro caso *file*). La prima scelta implementativa che il gruppo ha deciso di effettuare è stata quella di determinare quale tipologia di sistema distribuito sviluppare: si è optato per la tipologia **Decentralizzata Ibrida,** che prevede la presenza di un **Discovery Server**; tale server centralizza al suo interno informazioni quali la mappa del network con elenco dei file presenti a sistema, collegata ad una lista dei *Jeer* che rendono disponibile tale risorsa. Ogni *Jeer* che si unisce alla rete *Jolella* (tramite funzione *join*) comunica al server, ad ogni variazione, lo status, l'indirizzo e il buffer per la gestione dell'**unchoked status**. In tal modo funzioni quali la ricerca del file, il controllo delle richieste, vengono gestite unitamente dal Server centrale, con una notevole ottimizzazione del tempo. Il problema, tuttavia, del sistema implementato è sicuramente la presenza di eventuali colli di bottiglia per le richieste inviate da/per il server e la scarsa scalabilità (single point of failure): con il crescere ed il diffondersi di tale sistema, sicuramente la soluzione migliore sarebbe quella di aumentare il numero dei server che gestiscono tali richieste.

Altri importanti punti di forza del sistema sono:

- La possibilità di partecipazione alla rete da parte di un numero elevato di Jeer: questo è possibile, sia grazie alla centralizzazione di alcune funzioni (che di fatto centralizzano e risolvono il problema di mappatura della rete), sia alle scelte implementativa integrate;
- L'interoperabilità e la possibilità di aggregazione delle risorse;
- Il dinamismo del sistema (grazie ad una gestione delle risorse sviluppata ad hoc per il server centrale).

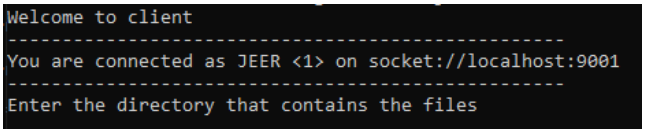
3 ISTRUZIONI PER LA DEMO

3.1 REQUISITI DI SISTEMA

- Java™ SE Runtime Environment (build 12.0.2)
- Jolie 1.8.2 – The Jolie Team

3.2 ESECUZIONE DELLA DEMO

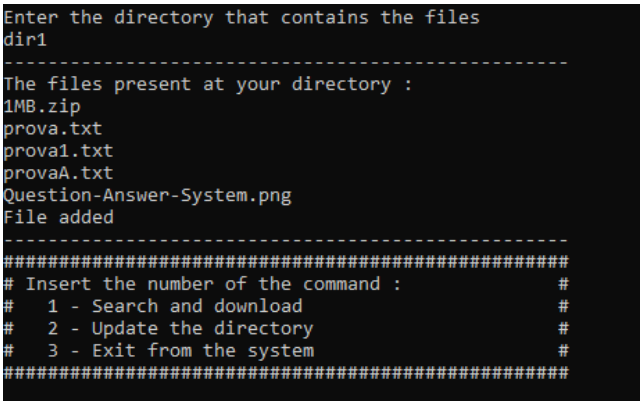
Si richiede l'apertura di n°2 shell per l'avvio del **Network Visualizer** (posizionarsi nella cartella dove presente ed inserire il comando jolie `networkVisualizzer.ol`) e del **Main Server** (posizionarsi nella cartella dove presente ed inserire il comando jolie `mainServer.ol`). Si richiede inoltre apertura di *n* ulteriori shell per il numero di client che si vogliono testare (posizionarsi nella cartella dove presente ed inserire il comando jolie `client.ol`).



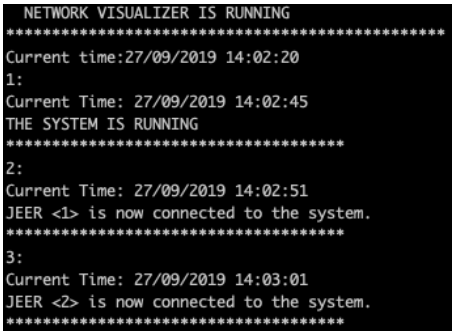
Il client deve poi selezionare la cartella che contiene i suoi file. Utilizzare uno dei seguenti valori:

- `dir1`
- `dir2`
- `dir3`
- `dir4`

All'inserimento della directory, il client avrà la possibilità di vedere i *file* al suo interno ed il sistema invierà al server la lista dei suoi *file* disponibili da aggiungere all'elenco dei *file* presenti nel network. Il client avrà poi la possibilità di decidere, inserendo il numero corrispondente, quale azione intraprendere:



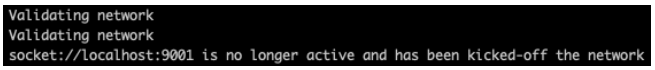
Il *Network Visualizer* si occupa di monitorare lo status della rete ed i Jeer che si uniscono, si tratta di un servizio di log implementato ai fini della dimostrazione.



Il *Main Server* (Supernodo Centrale) si occupa di gestire tutto le informazioni della rete, comprendendo attività di *look-up* delle risorse e indirizzamento delle richieste.

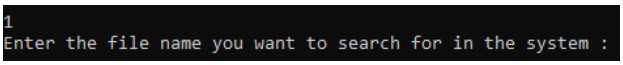
Il servizio, inoltre, monitora lo status dei Jeer (*Validating network*) attraverso un algoritmo di second chance, che permette di gestire i "*bad behaviour*" dei Jeer che lasciano la rete senza comunicarlo o che non espongono il proprio servizio sulla rete per altri motivi tecnici (*Jeer fault*).

Nel caso un Jeer non comunichi il suo status di attività (operazione eseguita ogni 15 secondi, contro i 30 dell'algoritmo), il server si occuperà di impostarlo come non attivo, non restituendolo nell'elenco dei Jeer che condividono le risorse.



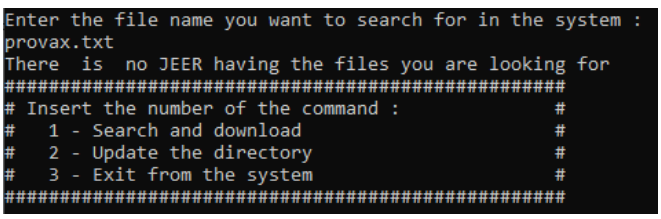
3.2.1 Ricerca e Download di un nuovo File

Il client che fosse interessato al download di nuovo file, non presente nella sua cartella, deve digitare "1" ed inserire il nome del file completo di estensione (es. `prova.txt`):

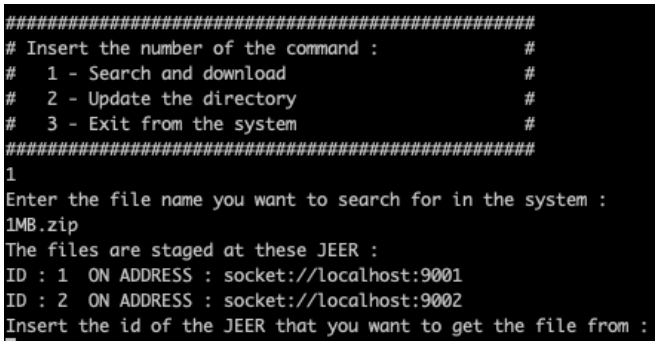


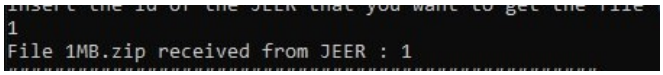
La richiesta viene inviata al Server che controlla se, per il file richiesto, ci sono Jeer attivi.

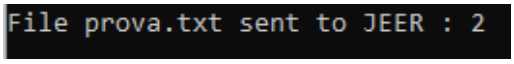
- Il server non trova Jeer attivi che possiedono il file richiesto: restituisce un messaggio al client comunicandogli che il file richiesto non è presente ed inserisce nuovamente il tab delle azioni disponibili per il client:

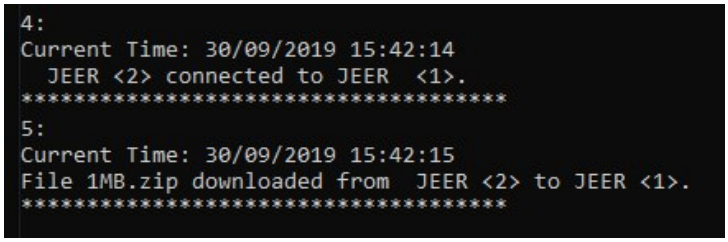


- Il server trova Jeer attivi che possiedono il file richiesto: elenca al client fino a n°5 Jeer che hanno il file disponibile e buffer maggiore. Il client inserisce il numero di ID del Jeer da quale vuole effettuare il download ed inizia il download del file nella sua directory:





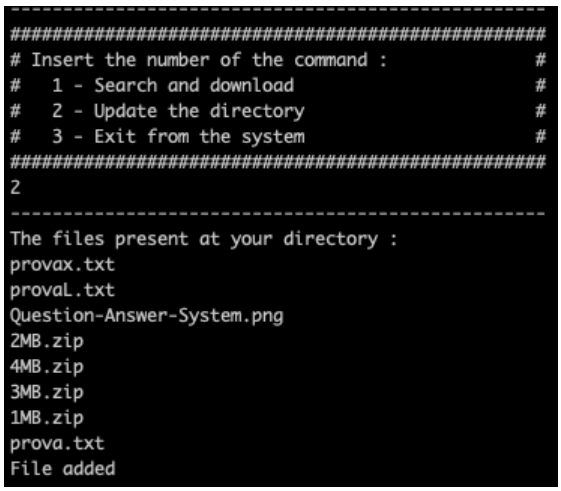




3.2.2 Update della Directory

Ai Jeer viene lasciata la possibilità di aggiornare manualmente la directory condivisa, susseguentemente l'aggiunta di un nuovo file pronto per lo sharing: per richiamare l'opzione è sufficiente dare "2" come input (come suggerito da menu):

- Viene nuovamente scansionata la directory di default inizializzata all'avvio;
- Gli elementi trovati vengono inseriti in un albero congiunti con le informazioni del Jeer;
- L'albero viene inviato al nodo centrale, che lo itera inserendo le nuove occorrenze ed aggiornando l'elenco dei Jeer associati, avendo cura di non duplicare le occorrenze ridondanti.



3.2.3 Uscita dal sistema

Ultimo comando disponibile nel menu dei Jeer è quello per la corretta gestione dell'uscita dalla rete: per richiamarlo è sufficiente inserire come input "3" e il Jeer stopperà il servizio, dopo aver comunicato al server la sua uscita dalla rete.

```
socket://localhost:9002 leaved the network
```

Allo stesso modo comunicherà l'uscita al *Network Visualizer*.

```
4:
Current Time: 27/09/2019 14:16:31
JEER <2> left the system.
**********************************
```

4 DISCUSSIONE SULLE STRATEGIE DI IMPLEMENTAZIONE

4.1 STRUTTURA DEL PROGETTO

4.1.1 Dinamicità e sistema distribuito

La scelta di una rete ibrida ha posto di fronte a varie criticità: la presenza di un nodo centralizzato, limita certamente la scalabilità del sistema, a causa della presenza di un *single point of failure*, rispetto a una rete decentralizzata pura, però permette di gestire in maniera più efficiente il *look-up* delle risorse;, è importante che venga garantito un accesso al servizio in maniera concorrente per poter gestire le richieste ed impedire che i Jeer si ritrovino in una coda con attesa indefinita avendo comunque la possibilità di modificare e analizzare le risorse condivide, evitando dannose *race condition.*

4.1.2 Principali problemi riscontrati, alternative considerate e soluzioni scelte

Come già detto precedentemente, era necessario implementare un servizio con una elevata concorrenza, sia a 360° per quanto riguarda il server, sia per alcune funzioni del client. Per affrontare questa problematica, Jolie ci è venuto incontro grazie alla possibilità di creare dei servizi con esecuzione concorrente.

A causa della scarsa documentazione, per molte funzioni è stato necessario eseguire svariati test: in particolare per avere garanzie di comprensione sulla gestione delle *input guarded non-deterministic choice* in una situazione di accesso concorrente al servizio da parte di diversi client e del servizio stesso in caso di esecuzione a *runtime*. Inoltre, l'*embedding* del servizio Java ci ha costretti a decompilare il sorgente di Jolie per avere una panoramica completa dei metodi delle classi *Value* e *ValueVector* (che altrimenti sarebbero rimaste inutilizzabili per i nostri scopi).

La gestione della directory doveva essere eseguita utilizzando necessariamente una variabile condivisa tra i processi (**`global`**), la cui modifica veniva effettuata in particolari condizioni e poteva generare difformità nelle risposte ai client lettori. Si è optato per utilizzare la variante vista a lezione del *Paradigma lettore-scrittore* che non è però immune da *starvation*: abbiamo ipotizzato anche soluzioni alternative, ma abbiamo deciso di non implementarle.

Considerando che il servizio avrebbe dovuto girare in locale, inizialmente si pensava di dare allo user la facoltà di scegliere la porta, ma per un motivo di efficienza, abbiamo optato per una soluzione automatizzata gestita con in supporto del Nodo Centrale.

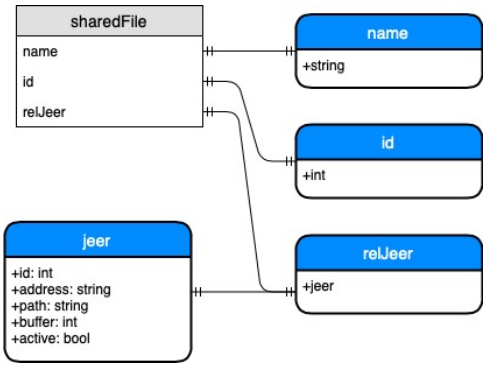
Inizialmente avevamo implementato la funzione di write nel Jeer, ma essendo bloccante ed essendo un'operazione che non influenzava la rete, abbiamo deciso di spostarla nel `node service` di modo che potesse essere eseguita concorrentemente.

4.2 Descrizione delle Feature

4.2.1 Struttura mainServer.ol

Alla luce delle considerazioni precedenti, per le funzioni del Nodo Centrale, abbiamo optato per un servizio ad elevata concorrenza, in cui un'unica funzione (`serverFunction`) gestisse le richieste esterne in maniera squisitamente non-deterministica.

Per la mappatura del network, è stato ideata una struttura ad albero con il nome dei file come radice e le variabili dei Jeer collegati come foglie: come da immagine, **sharedFile** conterrà le variabili del file condiviso, mentre **relJeer** sarà un array dei Jeer che possiedono una copia del file.



Come anticipato in precedenza, il server si occupa di gestire l'inserimento dei nuovi file condivisi dai Jeer (che gli vengono comunicati tramite apposita funzione di richiesta), avendo cura di non creare occorrenze duplicate sia nella struttura dei file, che nell'elenco delle risorse collegate.

```
define addFile { //aggiunge un file alla rete evitando duplicazioni
    for ( z = 0 , z < #req.sharedFile, z++ ) {
      s_i=0;

      exist = false;
      exJeer = false;

      while ( s_i < #global.base.sharedFile && !exist ) {
        //verifico se esistono file corrispondenti
        if ( global.base.sharedFile[s_i].name == req.sharedFile[z].name ) {
          s_j=0;

          //se esiste verifico se è già associato al jeer
          while( s_j < #global.base.sharedFile[s_i].relJeer && !exJeer ) {
            if( global.base.sharedFile[s_i].relJeer[s_j].address == req.sharedFile[z].relJeer[0].address ) {
              global.base.sharedFile[s_i].relJeer[s_j].buffer = req.sharedFile[z].relJeer[0].buffer;
              global.base.sharedFile[s_i].relJeer[s_j].path = req.sharedFile[z].relJeer[0].path;
              global.base.sharedFile[s_i].relJeer[s_j].active = true;
              exJeer = true
            };
            s_j++
          };
          //se non ha trovato il jeer lo aggiungo
          if( !exJeer ) {
            global.base.sharedFile[s_i].relJeer[#global.base.sharedFile[s_i].relJeer] << req.sharedFile[z].relJeer[0]
          };
          exist = true
        };
          s_i++
        };
      //se il file non esiste nella rete creo un nuovo nodo
      if(!exist){
        global.base.sharedFile[#global.base.sharedFile] << req.sharedFile[z]
      }
    }
}
```

La struttura ottenuta, priva di ridondanze ha vantaggi importanti: viene infatti implementato un algoritmo di ordinamento, che velocizza in maniera discreta le funzioni di ricerca (omesso nel presente manuale perché non mandatorio per il progetto).

Conseguentemente alle funzioni di ordinamento, esistono alcune funzioni “speciali” che permettono di ridurre il buffer del Jeer che inizia la condivisione del file e di aumentare lo stesso nel momento in cui termina l’operazione: suddette operazioni sono protette, causa potenziale *race condition* legata all’esecuzione concorrente del Nodo Centrale, da operazioni su semafori che verranno meglio descritte successivamente.

Per la funzione di ricerca, abbiamo deciso di optare per un servizio embeddato in Java, in quanto più efficiente nella gestione della struttura dati, che permette di restituire i primi 5 Jeer attivi con buffer > 0 nella lista dei Jeer collegati al file.

```
public class Search extends JavaService{

  public Value javaEmbedFind(Value srcVl){
    //utilizzo variabili locali per evitare race condition
    ValueVector ordVect = ValueVector.create();
    Value response = Value.create();
    Value thisOne = Value.create();
    Value unChockOpt = Value.create();
    ValueVector jeerTree = ValueVector.create();
    String nextOne;
    int buff = 5;

    //file da cercare
    String toFind = srcVl.getFirstChild("src").strValue();

    ordVect.deepCopy(srcVl.getChildren("sharedFile"));
    Iterator<Value> vectIterat = ordVect.iterator();

    //itero l'albero dei file per trovare quello corrispondente al valore di ricerca
    while(vectIterat.hasNext()){

      thisOne.erase();
      thisOne.deepCopy(vectIterat.next());
      nextOne = thisOne.getFirstChild("name").strValue();


      if(nextOne.equals(toFind)){
        jeerTree.deepCopy(thisOne.getChildren("relJeer"));
        Iterator<Value> jeerIterat = jeerTree.iterator();

        //prelevo massimo 5 jeer da proporre al client che ha effettuato la ricerca
        while(jeerIterat.hasNext() && buff > 0){
          unChockOpt.erase();
          unChockOpt.deepCopy(jeerIterat.next());

          if(unChockOpt.getFirstChild("active").boolValue() && unChockOpt.getFirstChild("buffer").intValue()>0){
            response.getNewChild("jeer").deepCopy(unChockOpt);
            buff--;
          }
        }
        break; //interrompo la ricerca una volta trovato
      }
    }
    return response;
  }
```

È importante notare che l'uso di variabili locali al metodo non si tratta di una scelta casuale: l'accesso al servizio avviene in maniera del tutto concorrente dai processi lettori; utilizzare variabili globali per tutti i processi, avrebbe portato alla possibilità di generare una *race condition* (e di conseguenza risultati sbagliati), che è stata così evitata.

È importante notare, a questo punto, che si è optato per l'utilizzo di semafori, proprio per permettere l'esecuzione concorrente dei processi: si tratta di un problema tipico lettore/scrittore, che abbiamo gestito con una variante basilare dei semafori che non è immune da *starvation*. Esistono soluzioni più complesse che risolvono il problema, tuttavia, considerando lo scopo dimostrativo del progetto ed il fatto che queste soluzioni non erano incluse nel programma di esame, si è optato per mantenere l'algoritmo a un livello basilare.

```
}else if ( req=="find" ) { //operazione di ricerca file nella rete
      //wait lettore che verifica di essere il primo e incrementa il numero lettori
      acquire@SemaphoreUtils(global.rd_file)(stat);
      //se è il primo lettore acquisisce il semaforo di scrittura
      if( global.num_file_lettori == 0 ) {
        acquire@SemaphoreUtils(global.wr_file)(stat)
      };

      global.num_file_lettori++;
      //signal lettore
      release@SemaphoreUtils(global.rd_file)(stat);

      println@Console( "Searching for: "+ req.toSearch )();

      javaBase.src = req.toSearch;
      javaBase.sharedFile << global.base.sharedFile;
      //servizio Java di ricerca
      javaEmbedFind@javaPort(javaBase)(found);

      msgFromSrv = "founded";

      msgFromSrv.jeers.jeer << found.jeer;
      //wait lettore
      acquire@SemaphoreUtils(global.rd_file)(stat);
      global.num_file_lettori--;
      //se ultimo lettore signal write
      if( global.num_file_lettori == 0) {
        release@SemaphoreUtils(global.wr_file)(stat)
      };
      //signal lettore
      release@SemaphoreUtils(global.rd_file)(stat)
```

Nel richiamare il servizio di ricerca del file, viene utilizzata una variabile locale al posto di una globale (sempre per evitare *race condition*): **javaBase** contiene una deep copy della directory ed una foglia con la stringa relativa al file da cercare, queste informazioni vengono poi passate al servizio embeddato, che restituisce la lista dei Jeer disponibili (max 5) unchocked.

4.2.1.1 Gestione concorrenza

Sono stati implementati n°3 semafori binari (`wr_file, wr_stat, rd_file`) e due variabili che contano il numero dei lettori. N.B: i processi che andavano incontro ad un paradigma lettore-scrittore erano due:

1. Relativo alla modifica dell'albero della directory;
2. Relativo alla modifica dell'albero degli status (modificato dall'algoritmo di *second-chance,* che eventualmente potrebbe modificare anche l'albero della directory);


```
}else if ( req=="find" ) { //operazione di ricerca file nella rete
      //wait lettore che verifica di essere il primo e incrementa il numero lettori
      acquire@SemaphoreUtils(global.rd_file)(stat);
      //se è il primo lettore acquisisce il semaforo di scrittura
      if( global.num_file_lettori == 0 ) {
        acquire@SemaphoreUtils(global.wr_file)(stat)
      };

      global.num_file_lettori++;
      //signal lettore
      release@SemaphoreUtils(global.rd_file)(stat);

      println@Console( "Searching for: "+ req.toSearch )();

      javaBase.src = req.toSearch;
      javaBase.sharedFile << global.base.sharedFile;
      //servizio Java di ricerca
      javaEmbedFind@javaPort(javaBase)(found);

      msgFromSrv = "founded";

      msgFromSrv.jeers.jeer << found.jeer;
      //wait lettore
      acquire@SemaphoreUtils(global.rd_file)(stat);
      global.num_file_lettori--;
      //se ultimo lettore signal write
      if( global.num_file_lettori == 0) {
        release@SemaphoreUtils(global.wr_file)(stat)
      };
      //signal lettore
      release@SemaphoreUtils(global.rd_file)(stat)
```

```
if( req == "add" ){ //operazione di aggiunta file alla rete

      //wait su semaforo scrittore
      acquire@SemaphoreUtils(global.wr_file)(stat);

      addFile;
      algoCall;
      //signal su semaforo scrittore
      release@SemaphoreUtils(global.wr_file)(stat);

      msgFromSrv = "File added"
```

Nelle immagini precedenti vengono rappresentati i due processi principali coinvolti nell'interrogazione dell'albero della directory, mentre sia il processo di second chance, che quello di servizio, utilizzato dal client per dichiararsi attivo, accedono a 2 sezioni critiche (una relativa all'albero dello status dei Jeer, ma possono, nel caso in cui debbano disattivare un Jeer, accedere anche alla sezione critica relativa alla modifica dell'albero della directory): per risolvere il problema abbiamo dunque utilizzato un semaforo di scrittura aggiuntivo relativo allo status dei Jeer che viene acquisito a inizio processo, prima di avviare l'algoritmo conseguente.

4.2.2 Struttura client.ol

Nel Client sono state implementate 3 output port:

```
outputPort INNODE {
Protocol: http
Interfaces: ClientClient
}

// porta statica   per comunicare con il  nodo centrale
outputPort MAINSRV {
  Location: "socket://localhost:8000"
  Protocol: http
  Interfaces: JolellaMain
}

//porta per comunicare con il monitor
outputPort TOMONITOR {
  Location:"socket://localhost:8051"
  Protocol:http
  Interfaces:MonitorInterface
}
```

- INNODE: per comunicare con altri Jeers o per stampare messaggi del Jeer attuale
- MAINSRV: per la comunicazione con il Server Centrale
- TOMONITOR: per stampare informazioni di log (join e movimenti) dei Jeers sul Monitor.

Il file client contiene due blocchi principali di codice: init e main.

Nel blocco init esistono 4 passi per la costruzione di un Jeer:

1. Dynamic binding

Il dynamic binding ci permette di creare sistemi dinamici, dove le informazioni di "binding" (locations e protocolli) possono cambiare durante l'esecuzione: il Dynamic binding è ottenuto trattando le porte di input/output come variabili.

In questa prima fase viene inviato un messaggio al server per connettersi al sistema con una stringa "join" (lato server sono stati implementati una serie di comandi, richiamati tramite la stessa funzione `serverFunction`): il server risponde con la location generata automaticamente nella procedura `sendJeerForCon` (sezione critica protetta da semaforo).

Grazie a questa funzione, il Nodo Centrale, decide la location dell'ouput port INNODE (univoca e progressiva ad ogni operazione di join) che ci permette di comunicare verso il `node service`, trattando ogni Jeer come servizio a sé stante (nonostante sia eseguito in locale): i Jeer impostano la location dell'input port generata (porta INNODE del `node service`) quando si chiamano i metodi dell'interfaccia ClientClient.

```
msgFromJeer = "join";
msgFromJeer.jeerAddress = PART_ADDRESS;
serverFunction@MAINSRV(msgFromJeer)(response);
```

```
define sendJeerForCon{ //per gestire il servizio in locale si assegna una porta progressiva

  global.port++;
  global.jeerAddress[global.port-9001] = req.jeerAddress + global.port;
  global.jeerAddress[global.port-9001].secondChance = true;
  global.jeerAddress[global.port-9001].deactivated = false;
  msgFromSrv = "You are connected ";
  msgFromSrv.jeerAddress = global.jeerAddress[global.port-9001]
}
```

2. Per estrarre l'id del Jeer, essendo quest'ultimo progressivo, viene utilizzata un'operazione di substring dall'ultima cifra della porta.

```
loc = response.jeerAddress;
println@Console( loc )();
str = loc;
str.begin = 22;
str.end = 23;
substring@StringUtils(str)(idFromLoc);
```

3. Viene richiesto all'utente di inserire il nome della cartella con la quale il JEER si presenta al network, mettendo a disposizione degli altri JEER i propri files (verrà poi data, successivamente, la possibilità di aggiungere quelli che verranno scaricati dal network scegliendo l'operazione "2 -Update the directory"); in questo passo abbiamo usato due operazione: `exists` per verificare se il nome della cartella esista o meno (nel caso di inesistenza il sistema stampa un messaggio "Directory does not exist." e richiede di ripetere l'input); la seconda operazione si chiama `isDirectory` e serve per verificare se il nome inserito sia di una cartella o di un file (in caso di errore verrà stampato un messaggio "You have inserted a file name." e richiederà nuovamente di ripetere l'input).

```
verifyDir = false;
directory = "";

while( verifyDir  == false) {
  println@Console( "Enter the directory that contains the files \n")();
  in(dir);

  exists@File( dir )( resps );

  if( resps ) {

    isDirectory@File( dir )( isDir );

    if( isDir ) {
      directory = dir;
      verifyDir = true

    }else{
      println@Console( "You have inserted a file name ." )()
    }


  }else{
    println@Console( "Directory does not exist.")()
  }
};
```

4. Dynamic embedding:

   Utilizzando il dynamic embedding, viene permesso al client di invocare le operazioni embeddate nel servizio `node service`. Come illustrato nel codice, usiamo Location per impostare l'input port del node service.

```
/* dynamic embedding of node_service.ol */
with( emb ) {
      .filepath = "-C LOCATION=\"" + loc + "\" node_service.ol";
      .type = "Jolie"
};
loadEmbeddedService@Runtime( emb )()
```

Nel blocco main del client, verrà chiamata per prima l'operazione `userCall` tramite INNODE: questa avvia un ciclo manda una richiesta al server ogni 15 secondi per confermare il proprio status attivo.

Una volta salvato il Jeer, abbiamo usato il commando curl, come consigliato nella documentazione del progetto su Github, per inizializzare la directory di ogni Jeer.

```
//inizializza la directory di ogni jeer con un file scaricato.
execRequest = "curl";
 with( execRequest ){
.args[0] = "-L";
.args[1] = "http://ipv4.download.thinkbroadband.com:81/" + idFromLoc + "MB.zip";
.args[2] = "-O";
.workingDirectory = directory;
.stdOutConsoleEnable = true
};
exec@Exec( execRequest )( execResponse );
```

Appena si scarica il file zip, verrà stampata la lista dei file trovati nella cartella del Jeer: il sistema salva questi file in una lista per mandarla al server insieme ai dati del jeer che sono: id, address, path, buffer, active, e una stringa di commando "add" che specifica al server quale procedura si deve eseguire.

```
println@Console("-----------------------------------------------------")();
println@Console("The files present at your directory :")();
cartella.directory = jeer.path ;
list@File( cartella)( rispostaProva);

for( j = 0, j < #rispostaProva.result, j++ ) {
  println@Console( rispostaProva.result[j] )();
  msgFromJeer.sharedFile[j].id = global.idFile++;
  msgFromJeer.sharedFile[j].name = rispostaProva.result[j];
  msgFromJeer.sharedFile[j].relJeer << jeer
};
msgFromJeer = "add";
serverFunction@MAINSRV(msgFromJeer)(response);
println@Console( response )();
```

Dopo il salvataggio dei file di un Jeer, il sistema stampa la lista dei comandi disponibili:

- “Search and download”: ricerca di un file nella rete.
- “Update the directory”: aggiornamento dei file di un Jeer.
- “Exit from the system”: operazione di leave dalla rete e uscita dal sistema.

```
println@Console("#####################################################" )();
println@Console( "# Insert the number of the command :              #" )();
println@Console( "#   1 - Search and download                       #" )();
println@Console( "#   2 - Update the directory                      #" )();
println@Console( "#   3 - Exit from the system                      #" )();
println@Console("#####################################################" )();
```

Scegliendo tra i commandi, lo user può cercare un file nel sistema: ricerca che viene eseguita richiamando nel Server Centrale l’algoritmo di lookup in Java, tramite comando “find”.

Dopo la ricerca allo user viene data la possibilità di scegliere da quale dei Jeer, che si trovano nella lista dei nodes che possiedono il file richiesto, effettuare il download.

```
for ( k = 0, k < #resp.jeers.jeer , k++ ) {

  if( resp.jeers.jeer[k].id == iddownload ) {

    // cambiare la location del output port con la location del jeer da cui viene scaricato il file
    INNODE.location = resp.jeers.jeer[k].address;
    sendMsgToNode@INNODE("Hello JEER " + iddownload + " -- this is JEER  "+ jeer.id + ", downloading "+filen);

    // network Visualizzer
    tosend.content = "  JEER <" + jeer.id + "> connected to JEER  <" + iddownload + ">. ";
    monitor@TOMONITOR(tosend)();

    // manda un messaggio buffer al server per chiamare il metodo che gestisce il buffer .
     msgFromJeer = "buffer" ;
     msgFromJeer.jeerAddress = resp.jeers.jeer[k].address ;
     serverFunction@MAINSRV(msgFromJeer)(response);

    // si chiama il metodo download verso node service per fare il "read" dei file
    filename = "./"+resp.jeers.jeer[k].path+"/" + filen ;
    download@INNODE(filename)(fileResponse);

    // download  del file nel path del jeer actuale.
    msgToWrite.filename = "./" + jeer.path + "/" + filen;
    msgToWrite.format = "binary";
    msgToWrite.bin = fileResponse;
    INNODE.location = loc;
    write@INNODE(msgToWrite);

    msgFromJeer = "bufferUp" ;
    msgFromJeer.jeerAddress = resp.jeers.jeer[k].address ;
    serverFunction@MAINSRV(msgFromJeer)(response);
    //messaggi tra node che sono in connessione e verso il monitor
    INNODE.location = resp.jeers.jeer[k].address;
    sendMsgToNode@INNODE("File " + filen +  " sent to JEER : " +jeer.id);
    println@Console("File " + filen + " received from JEER : " +iddownload)();
    tosend.content = "File " + filen + " downloaded from  JEER <" + jeer.id + "> to JEER <" + iddownload + ">.";
    monitor@TOMONITOR(tosend)()

  }
};

//si riimposta la location del outputport con la location del jeer acutale .
INNODE.location = loc
```

Come seconda scelta il client può aggiornare la directory caricando cosi nel sistema i nuovi files (o aggiungersi ai nodes relativi a uno già presente) oppure effettuare il leave dal sistema.

Ogni operazione viene comunicata al Network Visualizzer tramite il metodo monitor.

Lato client (Jeer), non sono state prese particolari accortezze per quanto riguarda eventuali *race condition*: come tipo di servizio si presta bene a un utilizzo sequenziale dei comandi e le uniche funzioni implementate in maniera concorrente sono quelle relative al download del file e quella sfruttata dal Jeer per comunicare il suo stato attivo al Nodo Centrale.

Il client si occupa di comunicare costantemente con il server per una questione di monitoraggio della rete, gestita dal primo con l'algoritmo di *second chance*, questa funzione viene richiamata dal client all'avvio e richiama infinite iterazioni, intervallate di 15 secondi, con cui si assicura, salvo problemi di rete, una costante comunicazione con il Nodo Centrale; questa funzione viene eseguita all'interno di un sistema concorrente, per permettere al client di comunicare con il Server Centrale senza interruzioni, indipendentemente dalle operazioni che si trova ad eseguire.

```
//manda al server ogni 15 sec un messaggio "active" per non far' disconnettere il jeer.
[userCall(loc) ]{

  for(i=0, i >=0 , i++) {
    sleep@Time(15000)();
    msgFromJeer = "active";
        msgFromJeer.jeerAddress = loc;
    serverFunction@MAINSRV(msgFromJeer)(response)
  }
}
```

È importante sottolineare, che anche la funzione di `download` dei file è stata inserita nello stesso servizio concorrente, embeddato nel servizio client all'avvio.

```
  //esegue il read dei files per poi chiamare write nel client actuale .
[download(request)(response){
  fileToRead.filename = request ;
  fileToRead.format = "base64";
  //errore da gestire
  readFile@File(fileToRead)(res);
  response = res
}]
```

Lo stesso vale per la funzione `write`.

```
[write(request)]{
    base64ToRaw@Converter(request.bin)(fileToWrite);
    writeFile@File({
        .filename = request.filename,
        .format = request.format,
        .content = fileToWrite
    })()
}
```