Laboratorio Sistemi Operativi A.A. 2019-2020

GRUPPO WHATSUP

COMPONENTI DEL PROGETTO

Referente = Linari Luca 900644 - email = luca.linari2@studio.unibo.it

Zecchin Dario 882583

Rossi Alessandro 874901

Di Matteo Carlo 890216

INDICE

1 -> DESCRIZIONE DEL PROGETTO

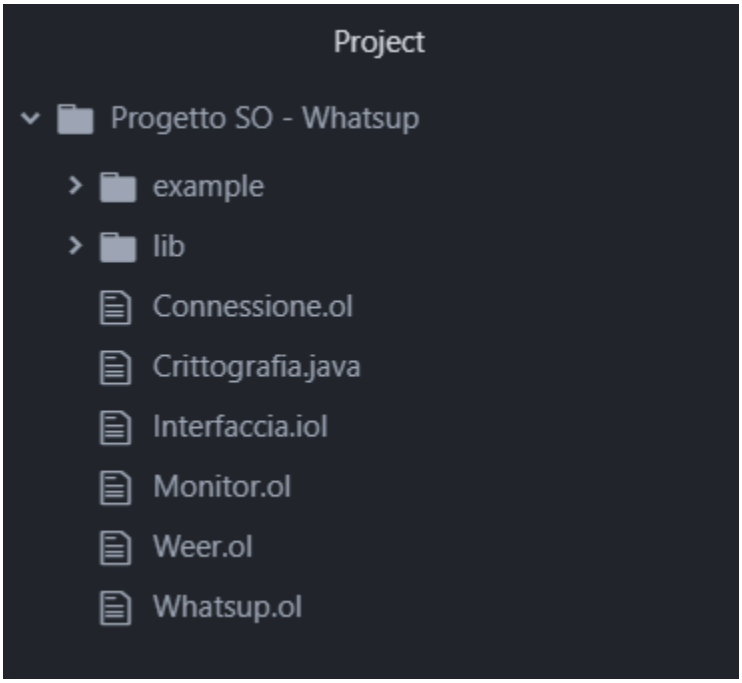
Whatsup è un'applicazione di messaggistica che sfrutta un servizio di ricerca centralizzato per la registrazione alla Rete e un sistema **Peer to Peer** per la comunicazione tra utenti.

In primo luogo, un utente per poter utilizzare il servizio deve registrare il proprio Peer, usando il file **Weer.ol**, a cui corrisponde un account presso **Whatsup.ol**, questo file svolge la funzione di un servizio di ricerca centralizzato, al fine di avere a disposizione tutti i Peer registrati alla rete.

Successivamente l'utente accede al menù principale dell'applicazione mediante le funzionalità di **Connessione.ol**, a questo punto ha la facoltà di scegliere una chat privata o pubblica, o abbandonare la rete, qualora scegliesse di iniziare una chat le conversazioni private e pubbliche verrano codificate per garantire l'integrità dei messaggi e la privacy degli utenti.

Una volta abbandonata la rete il Peer non potrà più usare l'applicazione fino al momento in cui si effettua nuovamente la sua registrazione.

Ogni azione svolta da un Peer all'interno dell'applicazione viene stampata a schermo tramite **Monitor.ol** che permette di potere avere una visione su quello che accade.



2 -> WHATSUP

Whatsup.ol è il file che ci permette di poter salvare in un array globale, accessibile a tutti gli utenti, ciascun Peer registrato nella Rete al momento.

La registrazione dei Peer avviene mediante un username e una porta, entrambi dati in input dal terminale quando viene lanciato il Peer, per questo è stato creato il **type Account**.

```
type Account : void{
  .username : string
  .porta : string
}
```

.username = è il nome del Peer che si vuole registrare, poichè se lo username è già presente collegato ad un altra porta l'utente è obbligato a scrivere un nuovo username, questo sistema garantisce l'univocità del nome all'interno della chat sia privata che di gruppo.

.porta = indica la socket con cui il Peer viene registrato alla rete, inoltre la porta viene sovrascritta alla costante WRITER_LOCATION.

```
//metodo che aggiunge il peer al network
[joinNetwork(account)(result){
  ok = true
  //ciclo che controlla che nel network non ci sia nessun peer con quello username
  for(i=0, i<#global.array.username,i++) {
    if(account.username==global.array.username[i]) {
    ok = false
    }
  }
  //se non c'è, il peer viene registrato al network
  if(ok){
    result = true
    global.array.username[global.counter] = account.username;
    global.array.porta[global.counter] = account.porta;
    global.counter++
    //se è già presente la registrazione viene interrotta
  } else {
    result = false
    Porta_Peer.location = account.porta;
    monitorMessage = "L'username " + account.username + " è già registrato alla rete pertanto la registrazione verrà interrotta";
    displayMonitor@Monitor_Output(monitorMessage);
    //il peer viene terminato
    killPeer@Porta_Peer()
  }
}]
```

Per poter aggiungere il type Account all'array globale di Whatsup.ol, il Peer lancia il metodo **joinNetwork** che controlla tramite un ciclo tutti gli elementi all'interno dell'array, e se l'username non è utilizzato viene automaticamente aggiunto al global array.

```
//metodo che permette ad un peer di lasciare il network
[leaveNetwork(portNumber)]{
  //scorro l'array globale dove sono segnati tutti i peer registrati alla rete
  for(i = 0, i < #global.array.username, i++){
    indexPortNumber = string(global.array.porta[i])
    //trovo la porta del peer che vuole lasciare la rete
    if(indexPortNumber == portNumber){
      peer = "Il Peer " + global.array.username[i] + " situato nella porta " + indexPortNumber + " si è DISCONESSO";
      displayMonitor@Monitor_Output(peer)
      //elimino dall'array il peer
      undef(global.array.porta[i]);
      undef(global.array.username[i]);
      //decremento il counter in modo da aggiungere il prossimo peer nella posizione lasciata libera da questo peer
      global.counter--
    }
  }
}
```

Viceversa per rimuovere un Peer dalla rete, quest'ultimo dovrà lanciare il metodo **leaveNetwork**.

Questo metodo inoltre gestisce il problema dell'indice dell'array che rimane vuota quando un utente si disconette, decrementando l'indice globale e permettendo al Peer successivo di registrarsi nella posizione lasciata libera.

3 -> CONNESSIONE

Connessione.ol è il file che contiene il menù, inizialmente il metodo menù doveva essere definito e lanciato direttamente in Weer.ol, lo script del Peer, per evitare un ulteriore file.

Questo causava un problema: quando il menù partiva, il Peer,in attesa di un comando di input, non riusciva a ricevere altre richieste.

Non volendo realizzare un comando per mettere il Peer in uno stato di attesa di una comunicazione, abbiamo optato per separare l'instaurazione della connessione in un file in embedding così da poter gestire le richieste, anche quando il Weer è nel menu, mentre il problema dell'input l'abbiamo risolto utilizzando la Swing_ui.

3.1 -> Connessione - Gestione delle richieste

Le diverse richieste vengono gestite nel file **Connessione.ol**, le variabili :

-global.occupato

-global.occupatoConnessione

Indicano se l'account sia occupato in una chat o in una connessione, nel caso vengano inoltrate richieste quando il valore è uguale a **"true"**, queste non vengono considerate.

In caso invece fossero settati a **"false"**, dopo un certo lasso di tempo se la richiesta non viene accettata, la richiesta diventa **"scaduta"**.

Il controllo è garantito con il confronto del token tra il Weer richiedente e il Weer che risponde alla richiesta.

Questo token rappresenta la validità della connessione, nel caso fossero diversi tra i due Weer **la richiesta di connessione** viene scartata.

Infine la variabile **global.attesaRispostaChat** comunica al Weer stesso lo stato della propria richiesta.

```
  define __embSend {
/*In questo metodo viene richiamato il file Connessione.ol
tramite in embeeding.
Questo file contiene il Menù dell'applicazione e gestisce la Connessione
tra i vari weer */

      with( emb ) {
        .filepath ="Connessione.ol";
        .type = "Jolie"
      };

      loadEmbeddedService@Runtime( emb )( embMenu.location);
      esegui@embMenu(global.account.porta)
}
```

Ogni operazione del menù dovrà essere eseguita nella Swing dove è possibile scegliere tra tre comandi:

1. Inizia Chat Privata
2. Inizia Chat Di Gruppo
3. Abbandona

Per accettare una nuova comunicazione privata o pubblica da un Weer si dovrà premere il tasto "0", che rappresenta una sorta di quarto comando.

4 -> CHAT PRIVATA

Definiamo due Weer, uno che richiede di iniziare una chat, che lo chiameremo Weer1 e l'altro che dovrà accettare la comunicazione, e lo chiameremo Weer2.

-Il Weer1 controlla che il Weer2 non sia occupato in un'altra conversazione attraverso il servizio **getOccupato()**

-In caso fosse libero, avvia il servizio **mandaMessaggio()()**, questo avvisa il Weer2 che il Weer1 vuole iniziare una comunicazione con lui.

Dopo circa **10000 millisecondi**, se il Weer2 non accetta la comunicazione la richiesta scade.

```
mandaMessaggio@Weer(messaggiorichiesta)(accetta)
} else
  menu
  if(accetta){
  Weer.location = portaWeerarrivata;
  attesa@Weer(varPorta)
  } else{
    println@Console("L'altro Weer non vuole parlarti")();
    Weer.location = portaWeerarrivata;
    setTokenConn@Weer(" ");
    Weer.location = varPorta;
    weerScaduto@Weer(portaWeerarrivata);
    menu
```

In caso contrario, viene generato un token da assegnare ai due Weer, nel momento in cui il Weer2 accetta la connessione, viene controllato il token tra i due Weer per assicurare la corretta comunicazione.

```
[mandaMessaggio(portaWeerRichiedente)(response){
  global.attesaRispostaChat = 0;
  // questo blocca tutte le connessione entranti quando il menu è in attesa di una risposta a un altro weer
  if( global.occupatoConnessione == false){
    global.accountChat.porta = portaWeerRichiedente.myporta;
    global.occupatoConnessione = true;
    global.newChat.token = portaWeerRichiedente.token;
    Porta_output.location = portaWeerRichiedente.myporta
    println@Console("Premi 0 nel TERMINALE SWING per accettare la connessione DAL WEER " + portaWeerRichiedente.myporta)();
    // passato questo lasso di tempo la sessione sara' scaduta
    sleep@Time(10000)();
    if(global.attesaRispostaChat==0 || global.attesaRispostaChat==1){
      // nel caso nel terminale non si prema 0 per accettare questa Connessione vorrà dire che la richiesta è stata rifiutata
      response = false
    } else{
      response = true
    }
  } else{
    response = false
  }
  global.occupatoConnessione = false
}]
```

Se il controllo del token è positivo, vengono avviati i metodi della chat.

Abbiamo dato priorità a realizzare una chat in cui la comunicazione avvenga in modo veloce e intuitivo, stampando dal terminale i messaggi ricevuti e inviando ciò che venga scritto dal terminale.

Abbiamo voluto realizzare lo stesso metodo della chat per entrambi i Weer.

La condizione di uscita dalla conversazione viene offerta digitando **"exit"** o nel caso in cui l'altro Weer cambi il proprio stato.

La persistenza viene garantita salvando ogni messaggio mandato/ricevuto in un proprio file di testo, nominato come la concatenazione degli username dei due Weer.

Il corretto uso dell'input viene garantito attraverso l'uso di questi costrutti:

```
  token2 = new
  csets.sessionToken = token2
subscribeSessionListener@Console( { token = token2 } )();
 synchronized( inputSession ) {
   in( testo )
 }
unsubscribeSessionListener@Console( { token = token2 } )();
```

Quando viene digitato un messaggio viene inviato attraverso il servizio **scriviSulFIle()**: che si occupa di visualizzare il messaggio e riportarlo nel file del Weer ricevente.

Quando un Weer decide di uscire dalla chat il secondo Weer viene notificato attraverso il metodo **getMessaggioAltroPeer()**.

5 -> CHAT DI GRUPPO

Per iniziare invece una chat di gruppo, dopo aver chiesto il nome della chat, viene richiesto il numero di partecipanti e dopo aver instaurato (con la stessa idea della chat privata), con successo, la connessione con ogni membro della chat, la comunicazione può partire.

Viene usato un albero e attraverso la deep copy viene aggiornata la "lista" di Weer che appartengono al gruppo.

```
token   = new;
richiesta.portaPeer = global.account.porta;
richiesta.token = token
richiesta.nomeChat = global.chatName
// viene passato su richiesta l'albero degli account che compongono il gruppo
richiesta << global.chatGruppo
// per ogni account viene avviato il metodo
for( i = 0 ,i <  #global.chatGruppo.account, i++) {
  Porta_output.location = global.chatGruppo.account[i].porta
  chatDiGruppo@Porta_output(richiesta)
  // metodo chat a ogni peer
}
// infine anche al Weer stesso
Porta_output.location = WRITER_LOCATION;
chatDiGruppo@Porta_output(richiesta)
```

Anche i metodi della **chatDiGruppo** sono **"simmetrici"** tra i diversi Weer. Il funzionamento è quindi pressochè identico alla chat privata.

Ogni qualvolta un Weer abbandoni la chat viene aggiornata la "lista" dei Weer ponendo la porta del Weer uscente uguale a un valore vuoto, viene reso disponibile con il servizio **avvisaAltriWeer()**

```
[avvisaAltriWeer(portaUscente)]{
// avvisa ogni Weer nella chat di Gruppo che il Weer corrente sta uscendo
    for ( i = 0 , i < #global.chatGruppo.account, i++){
        if( global.chatGruppo.account[i].porta == portaUscente){
            global.chatGruppo.account[i].porta = " "
            println@Console("Il weer "+ portaUscente + " e' uscito dalla chat ")()
```

6 -> CRITTOGRAFIA

Sono state sviluppate due tipologie di **crittografia** diverse per la chat **privata** e per la chat **di gruppo**, anche se entrambe si basano su una **chiave pubblica** e una **privata** generate automaticamente e assegnate al Peer.

Sia la generazione delle chiavi sia i metodi che codificano sono stati sviluppati in Java per utilizzare le librerie **Cipher** e **Key**.

6.1 -> Crittografia - Chat Privata

Per la prima è stata adottato una crittografia **asimmetrica** con una chiave pubblica e una chiave privata basate sull'algoritmo **RSA**: il testo del messaggio viene codificato attraverso la chiave pubblica del destinatario che successivamente lo decodifica mediante l'uso della sua chiave privata.

```
KeyFactory keyFactor = KeyFactory.getInstance("RSA");
PublicKey pubKey = keyFactor.generatePublic(new X509EncodedKeySpec(publicKeyArray));
//codifico attraverso la libreria Cipher
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE, pubKey);
byte[] encryptedMessageArray = cipher.doFinal(messageArray);
```

```
KeyFactory keyFactor = KeyFactory.getInstance("RSA");
PrivateKey prvKey = keyFactor.generatePrivate(new PKCS8EncodedKeySpec(privateKeyArray));
//decodifico il messaggio criptato
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.DECRYPT_MODE, prvKey);
byte[] messageArray = cipher.doFinal(encryptedMessageArray);
//trasformo il messaggio decodificato in una stringa e lo metto nel value
String message = new String(messageArray, "UTF8");
```

6.2 -> Crittografia - Chat di Gruppo

La crittografia della chat di gruppo non codifica il messaggio, ma verifica l'integrità del messaggio e l'identità del mittente attraverso la firma digitale.

La firma digitale è il testo del messaggio trasformato in hash e codificato con la chiave privata del mittente, i destinatari confrontano la firma digitale ricevuta codificata con il messaggio della chat trasformato in una stringa hash, se le due corrispondono, la comunicazione è avvenuta in modo corretto.

Per la funzione hash è stata utilizzata la funzione **md5** di Jolie, fornita dall'interfaccia **messagge_digest.iol**.

```
byte[] digitalSignatureArray = digitalSignature.getBytes();
String hashMessageInput = inputValue.getFirstChild("messaggio_hash").strValue();
try{
  //assegno ad un oggetto PublicKey la chiave pubblica presa in input
  KeyFactory keyFactor = KeyFactory.getInstance("RSA");
  PublicKey pubKey = keyFactor.generatePublic(new X509EncodedKeySpec(publicKeyArray));
  //decripto la firma digitale ottenendo il messaggio hash
  Cipher cipher = Cipher.getInstance("RSA");
  cipher.init(Cipher.DECRYPT_MODE, pubKey);
  byte[] hashMessageArray = cipher.doFinal(digitalSignatureArray);
  //trasformo in stringa il messaggio hash
  String hashMessage = new String(hashMessageArray, "UTF8");
  //confronto che la stringa hash preso in input sia uguale alla stringa hash ottenuta prima
  boolean right = false;
  if(hashMessageInput.equals(hashMessage)){
      right = true;
  } else {
      right = false;
  }
```

7 -> MONITOR

Il monitor è un file che deve essere lanciato dopo **Whatsup.ol**, e ci permette di monitorare tutte le azioni che avvengono sulla rete.

Il file contiene un metodo **OneWay** che prende in input una stringa e la stampa sul terminale del monitor, per poter far ciò si è definita una inputPort e di conseguenza le sue relative outputPort che hanno una socket di riferimento fissa (**socket://localhost:9100**).

Il monitor, così come **Whatsup.ol**, deve stare sempre attivo in modo che possa mostrare costantemente ciò che succede.

```
[messaggiMonitor(str)]{
  synchronized( tokenMonitor ){
    //incremento il mio token
    global.countMonitor++;
    println@Console(global.countMonitor + " -> " + str)()
  }
}
```

8 -> ISTRUZIONI PER USARE L'APPLICAZIONE

1)Aprire un terminale e lanciare jolie Whatsup.ol

2)Aprire un altro terminale e lanciare jolie Monito1.ol

3)Iniziare a mettere dei Peer sulla rete usando il seguente comando:

SINTASSI = jolie -C "WRITER_LOCATION="socket://NUMERO_PORTA"" Weer.ol USERNAME

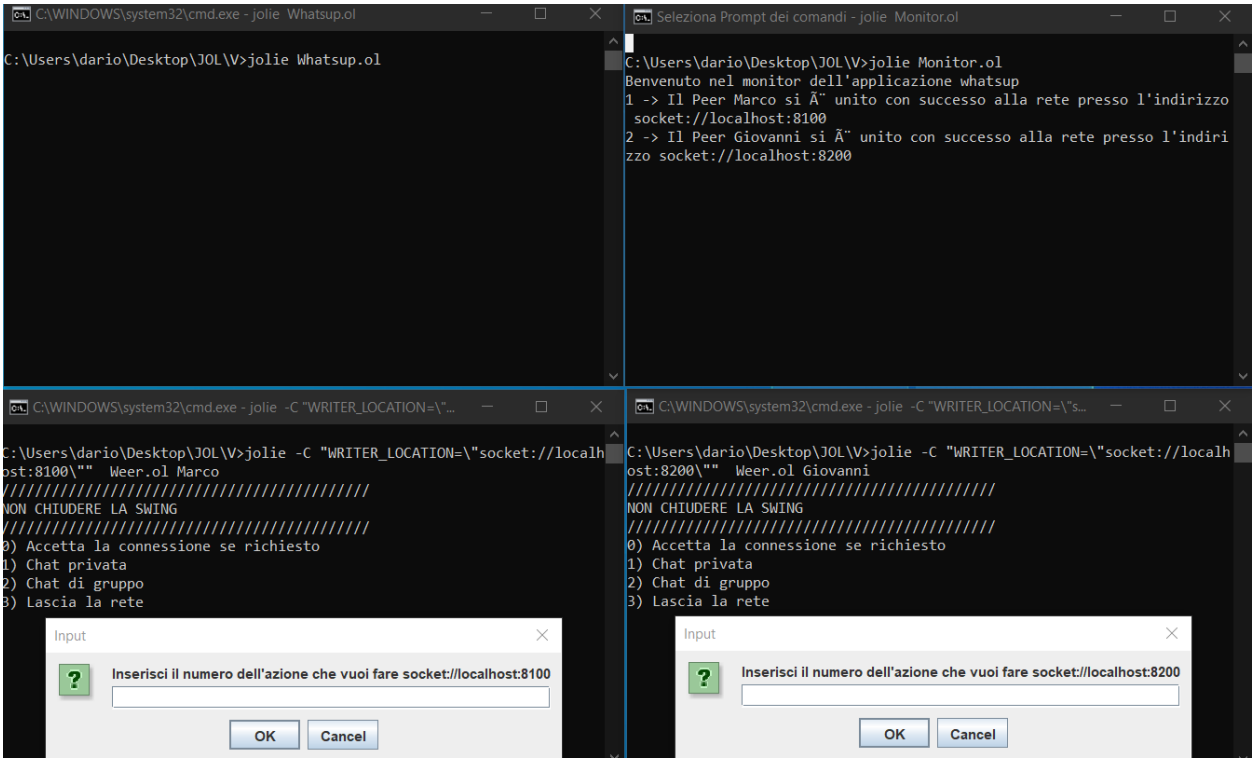
jolie -C "WRITER_LOCATION="socket://localhost:8100"" Weer.ol Marco

jolie -C "WRITER_LOCATION="socket://localhost:8300"" Weer.ol Givanni

jolie -C "WRITER_LOCATION="socket://localhost:8400"" Weer.ol Red

4)Si aprirà il menù e scegliere quale azione fare

IMPORTANTE = mai chiudere la swing/ui



9 -> STRUTTURA DEL PROGETTO

9.1 -> Divisione del Progetto tra i componenti del gruppo

All'interno del nostro gruppo abbiamo deciso di dividerci i compiti nel seguente modo:

-Linari Luca e Zecchin Dario hanno implmentato Whatsup.ol e quindi la gestione della registrazione e della disconessione del Peer, inoltre hanno sviluppato la crittografia in Java.

-Rossi Alessandro ha implementato la comunicazione divisa in chat Privata e chat di Gruppo e tutto ciò che ne consegue.

-Di Matteo Carlo si è occupato del monitor, delle eccezioni e della documentazione.

9.2 -> Problemi principali riscontrati nel'implementazione

9.2.1 -> Problemi generali

Il primo problema che abbiamo riscontrato è stato l'impossibilità di far partire una comunicazione dalla **PowerShell di Windows**, quando invece partiva dal prompt dei comandi sempre del medesimo sistema operativo.

Abbiamo riscontrato un problema simile anche su **MacOS** nel quale non era possibile iniziare una comunicazione che invece partiva negli altri sistemi operativi.

Si sono verificati alcuni problemi nella lettura di un file di testo (txt): nel confrontare dei numeri sotto forma di stringa in Jolie e nel conteggio dei caratteri che ci risultava diverso da un sistema operativo a un altro.

9.2.2 -> Problemi specifici

Ulteriori problemi riguardanti il codice sono emersi nella parte di crittografia, in particolare nella creazione di un oggetto **Key**. Infatti se i dati di una chiave erano in **String**, poi venivano trasformati in **array di byte**, a quel punto non era possibile assegnare il valore della chiave in una **Key**. Il problema è stato risolto partendo da un valore di **ByteArray**, invece che da un elemento tipo **String**.

Un altro problema riscontrato è stato quello legato all'input da tastiera utilizzando l'operatore **in** che risultava di difficile gestione, perciò abbiamo optato per usare la **Swing_ui**.