2019/2020

Indirizzo e-mail di riferimento:

jasonshuyinta@gmail.com

Componenti

Hajar Berdouzi, 815037

Giovanni Vitale, 838588

Keran Jegasothy, 839151

Tulak Mozammel, 843747

Jason Shuyinta, 831193

JoChat

INDICE

1. PROGETTO

Scopo

Creare una rete peer-to-peer che permetta ai peer di scambiarsi messaggi, sia su ***chat pubblica*** che ***privata***.

Scelta del gruppo

Il progetto è stato realizzato secondo un'architettura decentralizzata ibrida, che prevede la presenza di un server il quale contiene informazioni per facilitare la comunicazione tra peer.

Struttura del progetto

Il nostro progetto è composto da : Nodo.ol, Nodo_service.ol, Server.ol, ServerInterface.iol, MonitorInterface.iol e Monitor.ol.

Il fulcro principale del nostro progetto si concentra sulle entità del Server e del Nodo.

Abbiamo scelto di adottare una decentralizzata ibrida in quanto abbiamo voluto la presenza del server che occorre da tramite per i nodi per reperire le informazioni grazie alle quali i peer, dopo averle recuperate nel Server, potranno comunicare con altri peer.

La comunicazione avviene su chat privata o pubblica.

Nel caso della chat privata il Nodo A riesce a farsi comunicare le informazioni di un altro nodo, con cui vuole comunicare, dal Server, il quale ha tutte le informazioni dei nodi partecipanti alla rete. Il nodo A riesce a inviare il messaggio, utilizzando le informazioni reperite dal Server, al nodo B, impostando dinamicamente la porta di output (peer-to-peer). Il nodo_service del nodo B otterrà il messaggio in chat privata. Nell'invio il messaggio viene criptato, utilizzando l'algoritmo RSA (crittografia asimmetrica); per criptare il messaggio viene utilizzata la chiave pubblica, fornitasi dal Server, del nodo destinatario. Una volta arrivato il messaggio al nodo_service del destinatario esso utilizzerà il metodo di decriptaggio utilizzando la sua chiave privata.

Per quanto riguarda la chat pubblica si è deciso di strutturare il tutto in un Albero, che prevede come padre un elemento void. Nel 1* livello sono presenti i nomi della varie chat pubbliche presenti nel sistema. Nell'ultimo livello, invece, sono presenti le porte dei nodi che partecipano a una determinata chat pubblica. *(L'implementazione di tale albero è presente nel Server, per ulteriori dettagli andare nella sezione apposita Server.ol pagina 13).*

Motivo scelta implementativa

Analizzando le 3 architetture della rete P2P, abbiamo deciso di utilizzare la Decentralizzata Ibrida, in quanto ci sembrava sin da subito la soluzione più adatta ad un contesto reale. Non abbiamo attuato la Decentralizzata Pura in quanto abbiamo riscontrato delle difficoltà nell'implementare un servizio Broadcast di comunicazione tra i vari nodi. In seguito abbiamo preferito la presenza del Server, in quanto come specifichiamo nella documentazione, esso conterrà le informazioni necessarie ai nodi per comunicare.

2.1 INTERFACCE

MonitorInterface.iol

```
type Nodi : void {
    .nomeNodo: string
    .numeroPorta: string
}
```

Tipo Nodi :

- .nomeNodo: è il nome del nodo in formato string
- .numeroPorta: numero della Porta occupata dal nodo in formato string

```
interface MonitorInterface {
    OneWay:
    log(string )

}
```

OneWay: log -> invia un messaggio senza ottenere un messaggio.

- Invia un messaggio in formato stringa

serverInterface.iol

```
type Nodi:void{
.numeroPorta: string
.nomeNodo: string
}
```

Tipo Nodi :

- .nomeNodo: è il nome del nodo in formato string
- .numeroPorta: numero della Porta occupata dal nodo in formato string

```
type ListaChat: Chat
```

Tipo ListaChat: insieme di chat attive nel sistema

```
type Chat: void {
    .nomeChat: string
}
```

Tipo Chat: nome della chat restituito in stringa

```
type joinRequest: void {
    .nomeNodo: string
    .numeroPorta: string
    .chiavePub: string
}
```

Tipo joinRequest:

- .nomeNodo: il nome del Nodo restituito in stringa
- .numeroPorta : numero della Porta restituito in stringa
- .chiavePub: la chiave pubblica, generata, durante la creazione del nodo, restituita in stringa

```
type Dati:void{
   .numeroPorta: string
   .nomeChat: string
 }
```

- .numeroPorta: il numero della porta restituito in stringa
- .nomeChat: il nome della chat restituito in stringa

```
type Info: void {
    .numeroPorta: string
    .msg: string
    .usernameMittente: string
}
```

- .numeroPorta: il numero della porta restituito in stringa
- .msg: messaggio restituito in stringa
- .usernameMittente: nome del nodo mittente restituito in stringa

```
type Informazioni: void {
    .nomeMittente: string
    .nomeDestinatario: string
    .msg: string
}
```

- .nomeMittente: nome del nodo mittente restituito in stringa
- .nomeDestinatario : nome del nodo destinatario restituito in stringa
- .msg: messaggio restituito in stringa

```
type listaPartecipanti: void {
    .numeroPorta*: string
}
```

- .numeroPorta*: numero delle porte dei nodi restituito in stringa

```
type infoDestinatario: void {
  .numeroPorta: string
  .chiavePub: string
}
```

- numeroPorta: numero della porta restituito in stringa
- .chiavePub: chiave pubblica restituito in stringa

```
type gruppo: void {
   .nome: string
   .porta: string
}
```

- .nome: nome della chat pubblica restituita in stringa
- .porta : numero della porta restituito in stringa

```
type porte: void {
  .numeroPortaGruppo*: string
}
```

- .numeroPortaGruppo: numero della porta restituito in stringa

```
type infoChatGruppo: void {
    .nomeChat: string
    .mittente: string
    .messaggio: string
    .firma: string
    .chiavePub: string
}
```

- .nomeChat: nome della Chat restituito in stringa
- .mittente: nome del nodo mittente restituito in stringa
- .messaggio: messaggio restituito in stringa
- .firma: firma digitale restituito in stringa
- .chiavePub: chiave pubblica restituito in stringa

```
RequestResponse:
    getNodi( string ) ( string ),
    addNameChat( string )( string ),
    checkEsistenzaGruppo( string ) ( string ),
    getInfoDestinatario( string )( infoDestinatario ),
    getChat( void )( string ),
    sendNomeGruppo( string )( listaPartecipanti ),
    richiestaPorteGruppo ( string ) (porte)
```

getNodi: restituisce la lista dei nodi attivi

addNameChat: aggiunge una nuova chat di gruppo alla lista delle chat di gruppo

checkEsistenzaGruppo: controlla l'esistenza del nome del gruppo inserito da input

getChat: restituisce la lista delle chat di gruppo attive

sendNomeGruppo: prende il nome del gruppo e lo aggiunge a una lista

richiestaPorteGruppo: ottieni le porte dei membri partecipanti di un gruppo

```
OneWay:
    join( joinRequest ),
    log(string),
    sendChat( string ),
    invioPrivato( Informazioni ),
    SaveKey( string ),
    creaGruppo( gruppo ),
    gestioneGruppo( gruppo ),
    invioChatGruppo( infoChatGruppo ),
    sendNotifica(string),
    offline(string),
    uscitaGruppo(Dati),
    deleteNodo(Nodi),
    stampaFile(infoChatGruppo)
```

Join: aggiunge un nuovo nodo alla lista dei nodi già esistenti

log: stampa di un messaggio

sendChat: inserisce una nuova chat alla lista delle chat esistenti

invioPrivato: decripta il messaggio e lo scrive sui file dei rispettivi nodi

SaveKey: salva la chiave privata del nodo

creaGruppo: crea un nuovo gruppo con il nome e la porta del nodo creatore del gruppo

gestioneGruppo: aggiunge un nuovo membro ad un gruppo già esistente

invioChatGruppo: verifica la firma digitale e invia il messaggio sul terminale dei nodi destinatari

sendNotifica: invia una notifica di entrata o di uscita di un noto da un gruppo

offline: avviso di uscita di un nodo dalla rete

2.2 Monitor.ol

Nel nostro Sistema il monitor mostra i nodi attivi, mostrando il nome del Nodo e la porta da esso occupato.

```
C:\Users\39380\Desktop\JoChat>jolie monitor.ol
1. Nome del nodo : Hajar - Numero della porta: 9007
2. Nome del nodo : Giovanni - Numero della porta: 9008
3. Nome del nodo : Keran - Numero della porta: 9009
4. Nome del nodo : Jason - Numero della porta: 9010
5. Nome del nodo : Tulak - Numero della porta: 9011
```

Inoltre mostra anche quando un Nodo abbandona la rete.

```
6. L'utente Hajar ha abbandonato la rete!
7. L'utente Giovanni ha abbandonato la rete!
8. L'utente Keran ha abbandonato la rete!
9. L'utente Jason ha abbandonato la rete!
10. L'utente Tulak ha abbandonato la rete!
```

All'interno del main è presente un blocco syncronized che permette di gestire il global.count, ovvero il numero di righe.

2.3 NODO e NODO_SERVICE.OL

La nostra chat messaggistica sfrutta l'embedding di un programma Java per implementare la crittografia e firma digitale.

È necessario, per creare ciò, dedicare una porta riservata all'embedding.

```
//embedding del servizio Java
embedded {
    Java: "lib.Privata" in Privata
}
```

```
outputPort Privata {
    Interfaces: PrivataInterface
}
```

La location della porta per la comunicazione tra nodi viene impostata a run-time tramite i comandi mostrati nel seguente blocco:

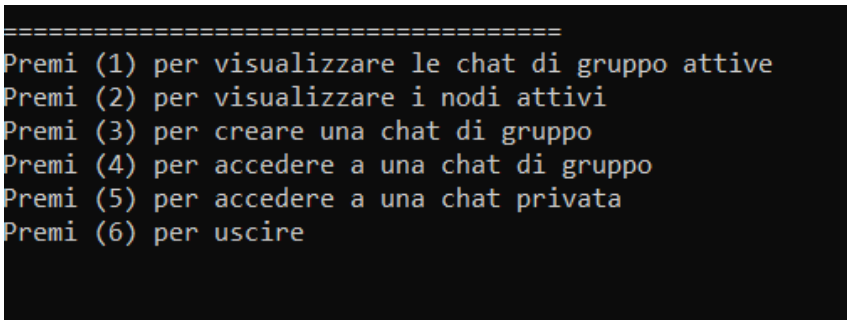
```
//embedding dinamico del servizio nodo_service.ol
socketLocation = "socket://localhost:"+args[1]+"/"
with( emb ) {
  .filepath = "-C LOCATION=\"" + socketLocation + "\" nodo_service.ol";
  .type = "Jolie"
};
loadEmbeddedService@Runtime( emb )()
```

Alla creazione del Nodo se il nome è valido viene creata la sua cartella e vengono generate le due chiavi: Pubblica e Privata, per la crittografia asimmetrica e firma digitale.
Il nome del nodo e la sua porta vengono poi aggiunte a una lista, la quale verrà mandata al Server, tramite servizio join, che sfrutta la porta 9000 riservata alla comunicazione con il Server.

```
main {
    registerForInput@Console(  )(  );
    request.nomeNodo = args[0];
    request.numeroPorta = args[1];
    exists@File( request.nomeNodo )( exists )

    while(exists == true) {
      println@Console( "Il nome inserito non è valido. Prova ad unserire un'altro nome: " )();
      in(request.nomeNodo)
      exists@File( request.nomeNodo )( exists )
    }
    mkdir@File( request.nomeNodo )( response );

    generatoreChiavi@Privata()(coppiaChiavi);
    chiavePub = coppiaChiavi.chiavePubblica;
    chiavePriv = coppiaChiavi.chiavePrivata;
    request.chiavePub=chiavePub;
    Nodo2Nodo.location = "socket://localhost:"+args[1];

    SaveKey@Nodo2Nodo(chiavePriv)
    join@Nodo2Server( request );
    println@Console(  )(  );
```

Ogni nodo è in grado di:

- **Visualizzare le chat di gruppo attive**: le informazioni delle chat di gruppo attive vengono inviate dal Server al Nodo per essere visualizzate.
- **Visualizzare i nodi attivi**: le informazioni dei nodi attivi vengono inviate dal Server al Nodo per essere visualizzate.
- **Creare chat di gruppo**: il Nodo, che crea una chat pubblica, invia al Server il nome del Nodo in modo tale da poter inserirlo all'interno della struttura ad albero della chat di gruppo.
- **Mandare un messaggio in una chat di gruppo:** il Nodo richiede al Server le porte dei nodi partecipanti ad una chat pubblica e invia ad ogni Nodo il messaggio.
- **Mandare un messaggio ad un altro nodo in una chat privata:** il Nodo richiede al Server la porta del Nodo Destinatario, con il quale vuole comunicare.
- **Uscire** : log out dal sistema di messaggistica.



(Il menù che si visualizza sul terminale)

3) Per creare una chat di gruppo:

Una volta inserito il nome corretto per la chat di gruppo essa viene aggiunta ad una lista, presente nel Server, e verrà creato il file sul quale verrà visualizzata la chat pubblica.

Per inviare un messaggio ai componenti del gruppo abbiamo implementato il servizio richiestaPortaGruppo, il quale dando come request il nome del gruppo, otterrà la location delle porte di tutte le componenti del gruppo. Queste location verranno settate dinamicamente alla porta adibita alla comunicazione tra i nodi.

```
//ottieni le porte dei membri del gruppo
richiestaPorteGruppo@Nodo2Server(gruppo.nome)(porte);

//assegna dinamicamente la location della porta dei nodi membri del gruppo
for (i=0, i<#porte.numeroPortaGruppo, i++){
  //controllo per evitare l'auto-invio del messaggio
  if (porte.numeroPortaGruppo[i] != args[1]){
    mySocketLocation = "socket://localhost:"+porte.numeroPortaGruppo[i];
    Nodo2Nodo.location = mySocketLocation;
```

Per l'invio di un messaggio all'interno di una chat pubblica è necessaria una firma digitale; tramite il servizio invioChatGruppo verranno inviati una serie di parametri per criptare il messaggio ed inviare il testo ai vari nodi partecipanti alla chat di gruppo tramite la porta Nodo2Nodo.

```
    //tramite chiave privata e messaggio crea la firma digitale
    pacchettoFirma.messaggio=message;
    creaFirma@Privata(pacchettoFirma)(firmaMessaggio)

    //invia il messaggio, allegando la firma digitale
    infoChatGruppo.nomeChat = gruppo.nome;
    infoChatGruppo.mittente = args[0];
    infoChatGruppo.messaggio = message;
    infoChatGruppo.firma = firmaMessaggio.firmaDigitale;
    infoChatGruppo.chiavePub = chiavePub;
    invioChatGruppo@Nodo2Nodo( infoChatGruppo )
  }
}
```

4) Per accedere a una chat di gruppo:

Questa opzione segue la struttura implementata nell'opzione 3 con l'unica differenza che è necessario che il gruppo, a cui il nodo vuole accedere, sia esistente.

5) Per accedere a una chat privata:

Quando il nodo clicca l'opzione 5 deve specificare il Nodo con cui vuole comunicare in una chat privata, passando al servizio getInfoDestinatario il nome del nodo (nel nostro sistema il nome di ciascun Nodo sono univoci).

Come ritorno avrà il numero di porta del nodo Destinatario e la sua chiave pubblica. *(per ulteriori dettagli riguardo alla crittografia andare nella sezione apposita crittografia pagina 15).*

```
//ottieni le informazioni necessarie per chattare privatamente con un nodo
getInfoDestinatario@Nodo2Server( nomeNodoDestinatario )( infoDestinatario );

//ottieni la chiave pubblica del destinatario per la cifratura del messaggio
cifraMsg.chiavePub=infoDestinatario.chiavePub;
```

La location, appena restituita, verrà impostata dinamicamente alla porta Nodo2Nodo per la comunicazione con il Nodo desiderato.

```
//setta dinamicamente la location della porta del destinatario
mySocketLocation = "socket://localhost:"+infoDestinatario.numeroPorta;
Nodo2Nodo.location = mySocketLocation;
```

Prima dell'invio il messaggio viene criptato tramite metodo encrypt (Java) e insieme al messaggio criptato vengono inviati il nome del Nodo Mittente e il nome del Nodo Destinatario.

```
//cripta il messaggio tramite chiave pubblica
encrypt@Privata(cifraMsg)(messaggioCifrato);

//invio del messaggio criptato
informazioni.nomeMittente = args[0];
informazioni.nomeDestinatario = nomeNodoDestinatario;
informazioni.msg=messaggioCifrato.messaggio;
invioPrivato@Nodo2Nodo( informazioni );
```

In “Nodo_Service.ol” abbiamo implementato il lato Server del Peer, con il quale può effettuare determinate operazioni, come :

1) invioPrivato, ovvero il quale effettua il controllo della crittografia *(per ulteriori dettagli a riguardo andare nella sezione apposita, ovvero Crittografia pagina 15)* e la scrittura all’interno del file txt all’interno delle cartelle dei Nodi Mittente e Destinatario, ovvero i nodi coinvolti nella chat privata.

```
[ invioPrivato( info )] {
  synchronized( token ) {
    decifraMsg.chiavePriv = global.chiaviPrivate;
    decifraMsg.messaggioCifrato = info.msg;
    decrypt@Privata(decifraMsg)(messNormale);

    testo=messNormale.messaggioDecifrato;
```

2) Per l’invio di un messaggio di una chat di gruppo viene innanzitutto verificata la firma digitale del messaggio tramite servizio controllaFirma (Java). Nel caso in cui l’integrità del messaggio non sia verificata allora uscirà in stampa “MESSAGGIO ANOMALO”.

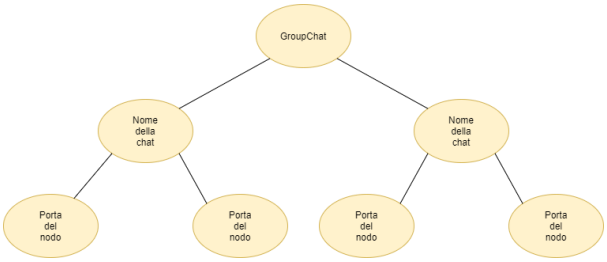
```
//controllo della firma digitale sui messaggi inviati
[ invioChatGruppo( infoChatGruppo ) ]{
  synchronized( token ) {
    controllo.firma=infoChatGruppo.firma;
    controllo.messaggio=infoChatGruppo.messaggio;
    //controllo.messaggio="TEST";
    controllo.keyPub=infoChatGruppo.chiavePub;
    controlloFirma@Privata(controllo)(verifica)
    if(verifica.risultato=="fallito"){
      println@Console("MESSAGGIO ANOMALO")()
    }
```

2.4 SERVER

Il Server viene implementato nel file “Server.ol”. Si ha gestito il Server come luogo per i peer, dove possono reperire le informazioni necessarie. Per esempio, come si mostra nel blocco di codice seguente, si ha il reperimento dei numeri delle porte dei nodi presenti in una determinata chat pubblica.

```
[ richiestaPorteGruppo( nomeChatGruppo ) ( porte ) {
  synchronized( token ) {
    for (i=0, i<#global.groupChat.name, i++){
      if (global.groupChat.name[i] == nomeChatGruppo){
        for (j=0, j<#global.groupChat.name[i].port, j++){
          porte.numeroPortaGruppo[#porte.numeroPortaGruppo] = global.groupChat.name[i].port[j]
        }
      }
    }
  }
}]
```

Un punto fondamentale del progetto che contiene il Server è l’implementazione della struttura ad Albero della chat pubblica.



Ogni qualvolta si crei una chat pubblica viene inserito un figlio all’albero che ha come padre l’elemento void. Inoltre il nodo, che crea il gruppo, inserisce già la sua porta, che si è identificata come figlio della chat di gruppo.

```
[ creaGruppo( gruppo ) ] {
  synchronized( token ) {
    i = #global.groupChat.name;
    global.groupChat.name[i] = gruppo.nome;
    for (j=0, j<#global.groupChat.name, j++){
      if (global.groupChat.name[j] == gruppo.nome){
        global.groupChat.name[j].port[#global.groupChat.name[j].port] = gruppo.porta
      }
    }
  }
}
```

Inoltre, quando un nodo vuole unirsi a una determinata chat pubblica invia al server delle informazioni, ovvero il nome del gruppo a cui vuole unirsi, e il numero di porta da esso occupato. Prima di inserirlo nella struttura ad Albero si effettua un controllo che evita l'inserimento doppio di un nodo all'interno della stessa chat pubblica.

```
[ gestioneGruppo( gruppo ) ] {
  synchronized( token ) {
    esiste = "false"
    for (i=0, i<#global.groupChat.name, i++){
      if (global.groupChat.name[i] == gruppo.nome){
        for (j=0, j<#global.groupChat.name[i].port, j++){
          if (global.groupChat.name[i].port[j] == gruppo.nome){
            esiste = "true"
          }
        }
        if (esiste=="false"){
          global.groupChat.name[i].port[#global.groupChat.name[i].port] = gruppo.porta
        }
      }
    }
  }
}
```

3 Crittografia

Nella chat privata, ovvero uno scambio di messaggi tra nodo mittente e nodo destinatario, il sistema deve assicurare che il messaggio non possa venir letto da terzi, criptando la comunicazione, utilizzando una crittografia **asimmetrica**.

La scelta implementativa del nostro gruppo si è basato sull'**algoritmo RSA,** implementato in un file java, il quale abbiamo embeddato con Jolie.

```
public static KeyPair generateRSAKkeyPair() throws Exception
{
  SecureRandom secureRandom = new SecureRandom();

  KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

  keyPairGenerator.initialize( 2048, secureRandom);

  return keyPairGenerator.generateKeyPair();

  }

  public static Value generatoreChiavi() {
    Value v= Value.create();

    try{
    KeyPair k= generateRSAKkeyPair();
    byte[] encodedBytesPub = k.getPublic().getEncoded();
    byte[] encodedBytesPriv = k.getPrivate().getEncoded();
    String chiavePubblica = Base64.getEncoder().encodeToString(encodedBytesPub);
    String chiavePrivata = Base64.getEncoder().encodeToString(encodedBytesPriv);
    v.getFirstChild("chiavePubblica").setValue(chiavePubblica);
    v.getFirstChild("chiavePrivata").setValue(chiavePrivata);

  }catch(Exception e){
    System.out.println(e);
  }
  return v;
  }
```

generatoreChiavi(): metodo che ritorna un oggetto di tipo Value; Value contiene la coppia di chiavi, in formato Stringa, generata dall'elemento k, oggetto del metodo generateRSAKkeyPair(). I passaggi sono: prendere le chiavi, trasformarle in array di byte e convertirle in stringhe grazie alla classe Base64.

generateRSAKkeyPair(): metodo che ritorna un oggetto KeyPair, utilizzando un algoritmo di crittografia RSA, con chiavi generate, di lunghezza 2048 bit, con un numero sicuro (secureRandom).

```
public Value encrypt(Value request)
{
  Value v= Value.create();
  String chiavePubblica = request.getFirstChild("chiavePub").strValue();
  String message = request.getFirstChild("msg").strValue();
  try{
  byte[] chiaveByte= Base64.getDecoder().decode(chiavePubblica.getBytes());
  KeyFactory kf = KeyFactory.getInstance("RSA");
  PublicKey publicKey = kf.generatePublic(new X509EncodedKeySpec(chiaveByte));
  Cipher encryptCipher = Cipher.getInstance("RSA");
  encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);
  byte[] cipherText = encryptCipher.doFinal(message.getBytes());
  v.getFirstChild("messaggio").setValue(Base64.getEncoder().encodeToString(cipherText));
}catch(Exception e){
  System.out.println(e);
}
  return v;
}
```

encrypt(): metodo che ritorna un oggetto di tipo Value; Value contiene il messaggio criptato. I passaggi sono: si prendono la chiave pubblica e il messaggio da input, da un elemento Value, si trasforma in array di byte la chiave, la si rende un oggetto PublicKey, attraverso l'elemento kf di tipo KeyFactory, si genera un elemento encryptCipher di tipo Cipher, con algoritmo RSA, lo si inizializza tramite la publicKey e il Cipher.ENCRYPT_MODE, si crea il messaggio criptato tramite la funzione doFinal() e, infine, lo si restituisce sottoforma di stringa.

```java
public Value decrypt(Value request)
{
  Value v= Value.create();
  String chiavePrivata = request.getFirstChild("chiavePriv").strValue();
  String message = request.getFirstChild("messaggioCifrato").strValue();
  byte[] bytes=null;
  Cipher decriptCipher = null;
 try{
  byte[] chiaveByte= Base64.getDecoder().decode(chiavePrivata.getBytes());
  KeyFactory kf = KeyFactory.getInstance("RSA");
  PrivateKey privateKey = kf.generatePrivate(new PKCS8EncodedKeySpec(chiaveByte));
  bytes = Base64.getDecoder().decode(message);
  decriptCipher = Cipher.getInstance("RSA");
  decriptCipher.init(Cipher.DECRYPT_MODE, privateKey);
  v.getFirstChild("messaggioDecifrato").setValue(new String(decriptCipher.doFinal(bytes)));
}catch(Exception e){
  System.out.println(e);
 }
  return v;
 }
```

decrypt(): metodo che ritorna un oggetto di tipo Value; Value contiene il messaggio decriptato. I passaggi sono: si prendono la chiave privata e il messaggio criptato, da un elemento Value, si trasforma in array di byte la chiave privata, per poi convertirla in un oggetto PrivateKey attraverso l'elemento kf di tipo KeyFactory, si converte in array di byte anche il messaggio criptato, si riprende un elemento decryptCipher di tipo Cipher, con algoritmo RSA e lo si inizializza con la privateKey e con Cipher.DECRYPT_MODE, e, infine, viene ritornato il messaggio decriptato trasformato in stringa.

4 FIRMA DIGITALE

Nella chat pubblica, ovvero dove i nodi pubblicano messaggi su canali, creati dagli stessi peer, il sistema deve garantire sia l'identità del mittente che l'integrità del contenuto dei messaggi al momento della consegna, utilizzando l'algoritmo di hashing SHA-256, che vien cifrato con l'algoritmo RSA, il tutto in un file Java, embeddato poi con Jolie.

```java
public Value creaFirma(Value request){
  Value v= Value.create();
  String chiavePrivata = request.getFirstChild("privKey").strValue();
  String message = request.getFirstChild("messaggio").strValue();
  String firmaDig = null;
  try{
  byte[] chiaveByte= Base64.getDecoder().decode(chiavePrivata.getBytes());
  KeyFactory kf = KeyFactory.getInstance("RSA");
  PrivateKey privateKey = kf.generatePrivate(new PKCS8EncodedKeySpec(chiaveByte));
  Signature sign = Signature.getInstance("SHA256withRSA");
  sign.initSign(privateKey);
  byte[] bytes = message.getBytes();
  sign.update(bytes);
  byte[] signature = sign.sign();
  firmaDig = Base64.getEncoder().encodeToString(signature);
  v.getFirstChild("firmaDigitale").setValue(firmaDig);
}catch(Exception e){
   System.out.println(e);
 }
  return v;
}
```

creaFirma(): metodo che restituisce un elemento Value; Value conterrà la firma digitale. I passaggi sono: da input, trasformiamo le due stringhe (messaggio e chiavePrivata) in array di byte, rispettivamente usando getBytes() e la classe Base64, la chiave privata viene convertita, successivamente, in un oggetto PrivateKey attraverso l'elemento kf di tipo KeyFactory, viene istanziato l'oggetto sign di tipo Signature, utilizzando un hash SHA256withRSA, sign viene inizializzato con la privateKey, viene aggiunto il messaggio alla firma, viene calcolata la firma digitale in un array di byte signature, quest'ultima viene trasformata in stringa e, infine, restituita.

```java
   public Value controlloFirma(Value request){
     Value v= Value.create();
     String result=null;
     String firma = request.getFirstChild("firma").strValue();
     String chiavePubblica = request.getFirstChild("keyPub").strValue();
     String mex = request.getFirstChild("messaggio").strValue();
     try{
     byte[] chiaveByte= Base64.getDecoder().decode(chiavePubblica.getBytes());
     KeyFactory kf = KeyFactory.getInstance("RSA");
     PublicKey publicKey = kf.generatePublic(new X509EncodedKeySpec(chiaveByte));
     byte[] firmaByte= Base64.getDecoder().decode(firma);
     Signature sign = Signature.getInstance("SHA256withRSA");
     sign.initVerify(publicKey);
     sign.update(mex.getBytes());
       //Verifying the signature
     boolean bool = sign.verify(firmaByte);
       if(bool)
         result="verificato";
       else
         result="fallito";
     v.getFirstChild("risultato").setValue(result);
   }catch(Exception e){
        System.out.println(e);
      }
      return v;
 }
```

controlloFirma(): metodo che restituisce un elemento Value; Value conterrà una stringa. I passaggi sono: da input, trasformiamo le tre stringhe (firma, chiavePubblica e messaggio) in array di byte, l'array chiaveByte si converte in un oggetto PublicKey, attraverso l'elemento kf di tipo KeyFactory, viene istanziato l'oggetto sign di tipo Signature, utilizzando un hash SHA256withRSA, sign viene inizializzato con la publicKey, viene aggiunto il messaggio alla firma, viene verificata la firma digitale, dando come risultato una stringa "verificato", nel caso in cui essa abbia un booleano true, oppure "fallito", nel caso in cui essa abbia un booleano false. Infine, viene restituito il risultato.

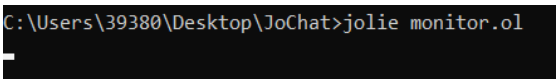
5.DEMO

1) Aprire un terminale e posizionarsi all'interno della cartella "LabSO_JoChat-master" e avviare il server tramite il comando:

<< jolie server.ol >>

2) Aprire un altro terminale nella stessa cartella e avviare il monitor con il comando:

<< jolie monitor.ol >>



3) Aprire un altro terminale nella stessa cartella e avviare un nodo tramite il comando:

<< jolie nodo.ol <nomeNodo> <numeroPorta> >>

(Per creare N nodi, aprire N terminali e ripetere N volte il comando sopraccitato.)

6.STRATEGIA DI DIVISIONE DEL LAVORO

A causa di questo periodo il gruppo non si è mai potuto incontrare dal vivo e tutte le nostre sessioni di lavoro sono state svolte utilizzando la piattaforma Teams e/o Zoom.

Abbiamo deciso di organizzare il lavoro dividendo le giornate in dei momenti: nel mattino i 5 componenti del gruppo si incontravano distribuendo i compiti e confrontando il lavoro svolto mentre nel pomeriggio ci si divideva in gruppo da 2 e/o 3 persone, in base alla disponibilità dei componenti.

In linea generale la distribuzione dei compiti ha seguito questo schema, tenendo conto sempre degli incontri mattutini dove erano presenti tutti i componenti del gruppo.

Principale divisione del lavoro:

- Hajar e Keran: gestione della chat di gruppo
- Hajar e Tulak: Ricerca informazioni riguardante crittografia, firma digitale ed embedding
- Jason e Giovanni : Crittografia, Firma Digitale
- Jason e Tulak: Gestione della concorrenza
- Giovanni e Jason : Gestione della Chat Privata
- Hajar e Keran: Stesura Documentazione

7. PROBLEMATICHE RISCONTRATE E LE RELATIVE SOLUZIONI

Un problema importante riscontrato è stato durante l'implementazione della rete. Inizialmente abbiamo sviluppato una rete client-server per quanto riguarda la chat di gruppo in quanto era il server stesso che mandava il messaggio di un nodo, ovvero il procedimento era: nodo- invio messaggio al server, il quale a sua volta mandava quel messaggio alla chat di gruppo. Durante l'implementazione e negli incontri di gruppo ci siamo resi conto dell'errore che stavamo facendo abbiamo modificato la rete client-server, in una rete p2p, ovvero la comunicazione avviene direttamente da nodo a nodi e il server ci occorre per determinare funzionalità per facilitare la comunicazione, nella quale però non viene coinvolto.

Un ulteriore problema che abbiamo riscontrato è stata nella procedura di embedding del file Java; inizialmente non siamo riusciti, come gruppo, nel comprendere la procedura per l'utilizzo dell'embedding e del file e dopo aver approfondito di più la documentazione Jolie presente su internet siamo riusciti a risolvere.

Infine un ultimo problema è stato durante la creazione del file di Java per quanto riguarda la grandezza dei numeri ammissibili nel calcolo delle chiavi, in quanto provavamo una conversione da Stringa a Long, e poi da Long a BigInteger. Tutto ciò ci dava errore di NumberFormatException. Per risolvere questo problema, abbiamo convertito da BigInteger a sequenza di bit. In questo modo, siamo riusciti a passare al nodo (Jolie) le chiavi. Per riottenere la chiave di tipo BigInteger, abbiamo fatto una conversione da sequenza di bit a BigInteger. In questo modo, abbiamo evitato il problema di perdere la precisione del valore della chiave.

Bibliografia e Sitografia

Documentazione Jolie : https://jolielang.gitbook.io/docs/

Crittografia e Decrittografia: https://www.devglan.com/java8/rsa-encryption-decryption-java

Firma Digitale : https://www.javaboss.it/crittografia-in-java/

Forum di Jolie: https://groups.google.com/d/msgid/infoman-so/CAP023roN88FWGgFU9SYHiTRhVJJkPnvJY%3D9OdSHRvAij%3DFGmSg%40mail.gmail.com